Software Engineering for Self-Adaptive Systems: A Second Research Roadmap (Draft Version of May 20, 2011)

Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw
(Dagstuhl Seminar Organizer Authors)
r.delemos@kent.ac.uk, holger.giese@hpi.uni-potsdam.de, hausi@cs.uvic.ca, mary.shaw@cs.cmu.edu

Jesper Andersson, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Goeschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Marin Litoiu, Antonia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Bradley Schmerl, Dennis B. Smith, João P. Sousa, Gabriel Tamura, Ladan Tahvildari, Norha M. Villegas, Thomas Vogel, Danny Weyns, Kenny Wong, Jochen Wuttke
(Dagstuhl Seminar Participant Authors)

ABSTRACT

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

1. INTRODUCTION

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

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

The goal of this roadmap paper is to summarize the state-of-the-art and identify research challenges when developing, deploying, managing and evolving self-adaptive software systems. Specifically, we focus on development methods, techniques, and tools that we believe are required when dealing with software-intensive systems that are self-adaptive in their nature. In contrast to merely speculative and conjectural visions and ad hoc approaches for systems supporting self-adaptability, the objective of this paper is to establish a roadmap for research and identify the key research challenges. Instead of dealing with a wide range of topics associated with the field, we focus on four essential topics of self-adaptation: design space of adaptive solutions, processes, from centralized to decentralized control, and practical run-

This roadmap paper is a result of the Dagstuhl Seminar 10431 on *Software Engineering for Self-Adaptive Systems* in October 2010.

Dagstuhl Seminar Proceedings 10431
Software Engineering for Self-Adaptive Systems
http://drops.dagstuhl.de/opus/volltexte/2011/3156

time verification and validation. The presentations of each of the topic do not cover all related aspects; instead focused theses are used as a means to identify challenges associated with each topic. The four identified theses are the following.

- *Design space for adaptive solutions* — the need to define what is the design space for adaptive software systems, including the decisions the developer should address.
- *Processes* — the need to define innovative generic processes for the development, deployment, operation, maintenance, and evolution of self-adaptive software systems.
- *From centralized to decentralized control* — the need to define a systematic engineering approach for designing centralized or decentralized control schemes for software adaptation.
- *Practical run-time verification and validation* — the need to investigate verification and validation methods and techniques for obtaining inferential and incremental assessments for the provision of confidence and certifiable trust in self-adaptation.

The intent of this new roadmap paper is not to supersede the previous paper on software engineering self-adaptive systems [11], but rather to complement and extend it with additional topics and challenges. The research challenges identified in the previous paper are still valid. Moreover, it is too early to re-assess the conjectures made in that paper. In order to provide a context for this roadmap, in the following, we summarize the most important challenges identified on the first roadmap paper [11].

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

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

2. DESIGN SPACE FOR ADAPTIVE SOLUTIONS

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

2.1 Design Space Definitions

The *design space* of a system is the set of decisions, together with the possible choices, the developer must make. A *representation of a design space* is a static textual or graphical form of a design space, or a subset of that space. Intuitively, a design space is a Cartesian space with dimensions representing the design decisions and values along those dimensions representing the possible choices. Points in the space represent concrete designs. In practice, most interesting design spaces are too rich to represent in their entirety, so representations of the design space capture only the principal decisions. Typically, the design dimensions are not independent, so making one decision may preclude, or make irrelevant, other decisions [5, 46].

Several partial methodologies for identifying and representing design spaces have emerged. For example, Kramer and Magee [29] outline three tiers of decisions the developer must make — ones that pertain to goal management, change management, and component control. Dobson et al. [12] identify four aspects of self-adaptive systems around which decisions can be organized: collect, analyze, decide, act. Finally, Brun et al. [6] discuss the importance of making the adaptation control loops explicit during the development process and outline several types of control loops that can lead to adaptation. Specific design spaces have also been proposed in the form of taxonomies. For example, Brake et al. [4] introduce (and Ghanbari et al. [17] later refine) a taxonomy for performance monitoring of self-adaptive systems together with a method for discovering parameters in source code. Ionescu et al. [24] formally define controllability and observability for web services and show that controllability can be preserved in composition.

2.2 Key Design Space Dimensions

In this section, we outline a design space for self-adaptive systems with five principal dimensions — clusters of design decisions pertinent to self-adaptive systems. The clusters are: observation, representation, control, identification, and

adaptation mechanism. Each cluster provides additional structure in the form of questions a developer should consider when designing such a system. While we hope our enumeration will help formalize and advance the understanding for self-adaptive system design, it is not intended to be complete and further work on expanding and refining this design space is necessary and appropriate.

2.2.1 Representation

The representation cluster is concerned with design decisions about run-time problem and system representations. To enable adaptation, key information about the problem and system has to be accessible at run-time.

The internal representation of the environment forms one important design decision. Choices include explicit representations — e.g., graph models, formulae, bounds, objective functions, etc. — or implicit representations in code.

The internal representation of the system itself forms another design decision. The choices here are similar to those for the environment representation.

2.2.2 Observation

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

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

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

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

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

2.2.3 Control

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

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

Feedback loops play an integral role in adaptation decisions. Thus, key decisions about a self-adaptive system's control are: "what are the involved control loops?" and "how do those control loops interact?" The choices depend on the structure of the system and the complexity of the adaptation goals. Control loops can be composed in series, parallel, multi-level (hierarchical), nested, or independent patterns. Brun at al. [6] have further discussed the choices and impact of control loops on the design of self-adaptive systems.

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

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

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

2.2.4 Identification

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

For each goal, there is a decision about which instantiations could satisfy that goal. The main concern of this decision is enumerating concrete sets of possible structures, behaviors, states, parameter values, etc. It is likely that not all identified instantiations will be supported at run-time. Selecting those that will be supported is another design decision.

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

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

2.2.5 Adaptation Mechanisms

The choice of adaptation mechanisms the self-adaptive system employs is an important cluster of design decisions.

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

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

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

2.3 Design Space Challenges

The design space described above can help formalize and advance the understanding of self-adaptive system design. However, it is not complete and further exploration and expansion is necessary to aid self-adaptive system developers. A more complete list can help ensure designers avoid leaving out critical decisions.

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

Each cluster we outlined above needs to be further expanded and refined. Further, validation of the decisions against real-world examples can serve as the framework for describing options. Dimensions in the self-adaptive design space are not independent and the interactions between the decisions in those clusters need to be explored. Understanding the decision relationships can narrow the search space and reduce the complexity of the design and of the design process.

An important challenge to consider is bridging the gap between the design and the implementation of self-adaptive systems. Frameworks and middleware (e.g., [13, 35]) can help bridge that gap, providing developers with automatically generated code and reusable models for their specific design decisions. This challenge is even more difficult in the case of reengineering existing non-self-adaptive systems or integrating self-adaptive and non-self-adaptive systems.

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

3. PROCESSES

Traditionally, software engineering (SE) research primarily focuses on principles for developing high quality software, rather than post-deployment activities, such as maintenance or evolution [37]. Meanwhile, it has been commonly accepted in the SE community that software implementing real world applications must continually evolve according to changes; otherwise, the software does not fulfill its ever changing requirements and therefore, will become outdated earlier than expected [32, 33]. This awareness has produced different software process models that address the inherent need for change and evolution by following iterative, incremental and evolutionary approaches to software development rather than strictly separating sequenced phases of requirements engineering, design, implementation, and testing [31, 37].

In the last decade, software evolution and maintenance have emerged as a key research field in SE [37] that separates the time before and the time after the software is delivered, or in other words, between development-time, deployment-time, and run-time in the software lifecycle. Post-delivery changes are typically done by re-entering the development stages of requirements engineering, design, implementation, and testing, which results in a new version of a software product or a patch. This is released to replace or enhance the currently running version [28]. Such releases are usually performed during scheduled down-times of the system compromising the system's availability. Thus, the whole maintenance process has been mainly done off-line guided by human-driven change management activities and decoupled from the running system.

However, such a lifecycle does not meet the requirements of *self-adaptive software* [11] that we are envisioning. A self-adaptive software system operating in a highly dynamic world must adjust its behavior automatically in response to changing environments or requirements while shifting the human role from operational to strategic. Humans define adaptation goals and new application or domain requirements, and the system performs all necessary adaptations autonomously at run-time. Throughout the system's lifecycle including adaptation periods, the system needs to be available and provide functionality to users or other systems keeping acceptable levels of quality of services (QoS).

Different researchers [1, 2, 22, 23] argue that we have to reconceptualize the whole SE process for modern software systems and particularly for the case of self-adaptive systems.

The problem we address is concerned with certain software evolution activities [8], more specifically their timing in the

process. This problem has three dimensions:

1. *Software lifecycle phases* [39] (e.g., development, deployment, operation, maintenance, and evolution).
2. *Software engineering disciplines* [39] (e.g., requirements engineering, design, implementation, validation, verification, etc.), and activities included in the disciplines (e.g., requirements elicitation, prioritization, and validation).
3. *Software evolution activities timeline* [8], that is, when change takes place (development-time, deployment-time, run-time).

The rationale is that in a self-adaptive software system lifecycle evolution activities are not bound to the traditional timeline (e.g., development-time), but are shifted to run-time. However, such shifts may introduce new process requirements in a different phase, for instance, that additional activities are performed during development. One example of changed timing for activities in self-adaptive systems is verification and validation. The dynamic nature of running self-adaptive systems and their environments requires continuous validation and verification to assess the system at run-time, which is traditionally done at development-time and which, however, requires new and efficient techniques for the run-time case (cf. Section 5). The consequence is a different and more dynamic SE process for self-adaptive systems that we want to understand and elaborate.

3.1 Example: Migrating Evolution Activities

To illustrate the specifics of SE processes for self-adaptive software systems and their differences to traditional software development and evolution activities, we compare the traditional approach to fix faults with the *automatic workarounds* approach [9, 10]. Automatic workarounds aim to mask functional faults at run-time by automatically looking for and executing alternative ways to perform actions that lead to failures.

Besides the implementation of new or changing requirements, the evolution of software systems may include corrective maintenance activities [47]. Traditionally, users experience failures and report them to developers who are then in charge of analyzing the failure report, identifying the root cause of the problem, implementing the changes, and releasing the new fixed version of the software. Traditionally these activities are done off-line.

In contrast, the automatic workarounds mechanism exploits the intrinsic redundancy of "equivalent operations" usually offered by software systems for different needs, but for obtaining the same functionality. Consider for example a container component that implements an operation to add a single element, and another operation to add several elements at the same time. To add two elements, it is possible to add one element after the other, or as an equivalent alternative to add them both at the same time. If adding two elements in sequence causes a failure at run-time, the automatic workarounds mechanism tries to execute the equivalent operation instead, as an attempt to avoid the problem. Thus, the automatic workarounds approach partially moves corrective maintenance activities to run-time. Once the user reports a failure, and based on that information, the automatic workarounds mechanism tries to find a workaround. It checks whether the failure has been experienced by other users, and a workaround has already been found. If so, it first tries to execute the known valid workaround. If no workaround is known, or the known workaround no longer works, the mechanism scans the list of equivalent operations, and checks whether they serve as workarounds.

The automatic workaround mechanism illustrates how some activities that were previously performed off-line in the maintenance phase by software developers are now performed at run-time by a self-adaptive mechanism. One such activity is *failure analysis*, were the causes for a failure are analyzed. In traditional maintenance, the failure report is analyzed by the developers while the self-adaptive behavior performs the analysis in the automatic workaround mechanism. In general, compared to traditional SE processes, adding self-adaptive behavior to a system will impact how processes supporting lifecycle phases are defined and connected. The automatic workarounds approach exemplifies three interesting characteristics of a process for self-adaptive systems

- *Migrating activities from one phase to another* — The analyzing failure reports activity is (partially) moved from the development-time (maintenance) phase to run-time. This (partially) delegates the developer's responsibility for this activity to a self-adaptation mechanism in the self-adaptive system.
- *Introducing new activities in other lifecycle phases* — Introducing the automatic workaround mechanism requires that additional activities are performed in the development and maintenance phases. One example is the identification of equivalent operations. Whenever some behavior is "assigned" to the automatic workaround mechanism, equivalent operations for this behavior must be identified.
- *Defining new lifecycle phase inter-dependencies* — The automatic workaround mechanism searches for equivalent operations, executes them, and lets the users evaluate the results. This goes on until the user approves the results, and thus the workaround, or until no more equivalent operations could be found. If this is the case, the mechanism is not able to provide a solution to the problem. T he only fallback available is to generate a failure report and send it to the maintenance organization where it will be dealt within the traditional maintenance activity. This exemplifies how traditional maintenance activities integrate with run-time activities, for instance, as information providers or as fallback activities if run-time activities do not succeed.

3.2 Understanding a Self-Adaptive Software System's Lifecycle

Understanding how software is best developed, operated, and maintained is an ever-present research challenge in the SE field. During the last two decades we have witnessed the development of ultra-large-scale, integrated, embedded,

and pervasive software systems that have introduced new challenges concerned with system operations: dynamic environments may change the goals of the system, the lack of intuitive interfaces makes it difficult for an external party to be responsible for the operation, and finally, the vast number of systems makes the operations task too complex for a single centralized machine or system operator. One answer to these advances is to instrument software systems with functionality that makes them more autonomous. This autonomy means that systems take over some of the responsibilities previously performed by other roles in the software lifecycle, such as sensing failures and automatically recovering from them.

An SE process is a workflow of activities that are performed by roles and that use and produce artifacts in order to develop, operate, and evolve a software system. In general, we conceive two extreme poles of SE processes [22, 23]. One pole corresponds to the traditional process creating a system that is frozen or static with respect to adaptation and evolution at run-time. In contrast, the other pole describes a process with almost all activities performed at run-time, which enables sophisticated self-adaptation capabilities. In practice, a process for a self-adaptive software system is positioned as a trade-off in between these two extreme poles.

Our goal is a generic process engineering framework for self-adaptive software systems that considers influential factors by providing a library of reusable process, activity, role, and artifact definitions. The framework guides and supports an engineer in understanding, specifying, tuning, and enacting an SE process for a concrete self-adaptive system. The framework is based on process models specifying how processes are carried out, because such descriptions materialize how a self-adaptive software system is developed and evolved. In addition they promote discussions, reuse, and even automated analysis [41], which in turn supports understanding the lifecycle of a self-adaptive system.

The key issue is the design of the process engineering framework for self-adaptive software systems, which includes three main components supporting process comprehension, specification, optimization, and enactment. The three components are (1) the library containing definitions of reusable process elements, (2) the specification of a concrete process for a specific self-adaptive software system, and (3) the analysis and tuning of process specifications.

Definitions of process elements for the library as well as process specifications should be based on an existing framework, like the *Software & Systems Process Engineering Metamodel Specification* (SPEM) [39]. SPEM provides a modeling language for process specifications including lifecycle phases, milestones, roles, activities, and work products. We need to identify required extensions to SPEM in order to model specifics of processes for self-adaptive systems, like the phases when process elements are employed and their relationships to other phases. For example, in the automatic workarounds approach, we identified one activity (analyze failure report) that may be performed as part of a regular maintenance phase or at run-time. Another example is to model dependencies between phases, e.g., an activity can only be performed at run-time if another activity has been performed at development-time. Extending SPEM will result in a language for the process engineering framework to define process elements for the library, to model concrete processes for self-adaptive software systems, and to analyze and tune these processes.

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

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

The third framework component explicitly addresses the product/process analysis and tuning to obtain an enactable process specification that appropriately fits the specific product and the product's goals and environments. A typical sensitivity point is the degree of adaptation and evolution support at run-time. Any design decision concerned with self-adaptive behavior must analyze, for instance, the overhead it introduces. Is the overhead acceptable or not? If not, are pre-computed adaptations possible to tune the process by reducing the overhead? As stated in [1] run-time validation and verification may not match the requirements of efficiency to allow the systems to timely react according to changes. This exemplifies that software design and process analysis/tuning are not isolated activities, and it promotes the continuous integration of design decisions and process analysis/tuning throughout a self-adaptive software system's lifecycle.

Finally, it is likely that an engineer uses the three components of the process engineering framework iteratively and concurrently rather than sequentially. For example, while specifying a process, an engineer does not find a suitable process element definition in the library, and thus, new definitions will be created and added to the library. Or during product/process analysis, an engineer identifies the need for process optimization, and searches the library for more

suitable process elements definitions that could be used to tune the process. Like software development processes, the process of using the framework itself is characterized by incremental, iterative, and evolutionary principles.

Another dimension that should be considered from the beginning when setting up a process engineering framework is the degree of automation. The process uses or is based on models throughout the lifecycle. Since the system evolves at run-time, the models have to evolve as well (model evolution) and models need to be accessible at run-time, either online or off-line [2]. The availability of run-time models makes it possible to use them as interfaces for monitoring [51] and adapting [50] a running system, and to perform what-if analysis and consultation [2], e.g., to test adaptations at the level of models before actually adapting the system. In addition, process activities must be based on up-to-date models. Changes in a run-time model allow the dynamic derivation of new capabilities to respond to situations unforeseen in the original design. Not all need to be new, we envisage the use of a library of model transformation strategies [1] to derive system models as well as keeping the process up-to-date with respect to the running system and vice versa. As an initial step, model synchronization techniques have already been applied at run-time to keep multiple system models providing different views on a running system up-to-date and consistent to each other [50, 51].

3.3 Research Challenges

The different problems and dimensions highlighted in the previous sections can be summarized by the following research challenges.

First of all, since dynamic environments may change the goals of the system, we need proper means to fully understand the nature of these systems and the key characteristics of their lifecycles to enhance the comprehension, specification, optimization, and enactment of the software process. More autonomy calls for the capability of self-reacting to anomalous situations. Both probing and reacting must be properly planned, designed, and implemented, and they also require that some activities, which were traditionally performed before releasing the system, be shifted at run-time.

Understanding how software processes change when developing a self-adaptive system also requires that influential factors be identified and understood. Particular factors impose specific self-* capabilities and also the degree of autonomy the system must embed. A clear identification of these factors is essential for designing a suitable process, identifying the required degree of automation, and positioning it between the two aforementioned poles. Some mild capabilities can be guaranteed through slightly static processes, while more advanced, extreme capabilities call for dynamic processes.

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

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

4. FROM CENTRALIZED TO DECENTRALIZED CONTROL

Control loops have been identified as crucial elements to realize the adaptation of software systems [12, 26, 45]. As outlined in the the former road map [11], a single centralized control component may realize the adaptation of a software system, or multiple control components may realize the adaptation of a composite of software systems in a decentralized manner. In a decentralized setting, the overall system behavior emerges from the localized decisions and interactions. These two cases of self-adaptive behavior, in the form centralized and decentralized control of adaptation are two extreme poles. In practice, the line between the two is rather blurred, and development may result in a compromise. We illustrate this with a number of examples.

Adaptation control can be realized by a simple sequence of four activities: monitor, analyze, plan, and execute (MAPE). Together, these activities form a feedback control system from control theory [44]. A prominent example of such adaptation control is realized in the Rainbow framework [14]. Hierarchical control schemes allow management or the complexity of adaptation when multiple concerns (self-healing, self-protection, etc.) have to be taken into account. In this setting, higher level adaptation controllers determine the set values for the subordinated controllers. A prominent example of a hierarchical control schema is the IBM architectural blueprint [20]. In a fully decentralized adaptation control schema, relatively independent system components coordinate with one another and adapt the system when needed. An example of this approach is discussed in [16] in which component managers on different nodes automatically configure the system's components according to the overall architectural specification.

These examples show that a variety of control schemas for self-adaptive systems are available. Our interest in this section is twofold: first, we are interested in understanding the drivers to select a particular control schema for adaptation; and second, we are interested in getting better insight in the possible solutions to control adaptation in self-adaptive

systems. Both the drivers and solutions are important for software engineers of self-adaptive system to choose the right solution concerning centralized or decentralized control. In the remainder of this section, we report on our findings concerning this endeavor and outline some of the major research questions we see to achieve that a systematic engineering approach for designing centralized or decentralized control schemes for software adaptation.

4.1 Distribution versus Decentralization

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

Textbooks on distributed systems (e.g., [48]) typically differentiate between centralized data (in contrast to distributed, partitioned, and replicated data), centralized services (in contrast to distributed, partitioned, and replicated services) and centralized algorithms (in contrast to decentralized algorithms).

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

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

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

The existing self-adaptive literature and research, in particular those with a software engineering perspective, have by and large tackled the problem of managing either local or distributed software systems in a centralized fashion (e.g., [14, 20, 40]). While promising work is emerging in decentralized control of self-adaptive software (e.g., [7, 16, 36, 52, 53]), we believe that there is a dearth of practical and effective techniques to build systems in this fashion.

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

4.2 Drivers for Selecting a Control Schema for Adaptation

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

4.2.1 Domain Characteristics

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

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

4.2.2 Requirements of the Problem at Hand

Stakeholder requirements may exclude particular solutions to realize adaptations.

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

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

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

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

4.3 Patterns for Interacting Control Loops

Ideally, we would like to have a list of problem characteristics/requirements and then match solutions against these. However, in practice, as stakeholders typically have multiple, often conflicting requirements, any solution will imply tradeoffs.

We have identified different solutions in the form of patterns of interacting control loops in self-adaptive systems. Patterns are an established way to capture design knowledge fostering comprehension of complex systems, and serving as the basis for engineering such systems. Each pattern can be considered as a particular way to orchestrate the control loops of complex self-adaptive software systems, as we explained in Section 2.2.3.

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

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

This pattern naturally fits systems with a hierarchical architecture. However, independently of the actual system architecture, hierarchical organization of the control system has been proposed (e.g., in [30]) to get a better separation of concerns among different control levels.

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

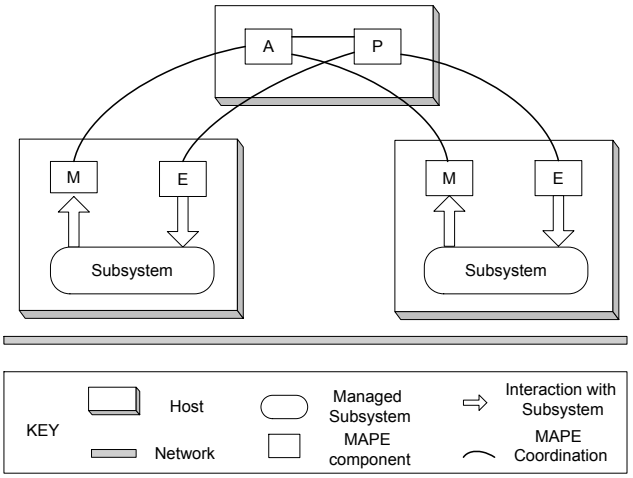

Figure 1: Master-slave pattern

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

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

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

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

planning of adaptations of its region.

Pattern 4: Fully Decentralized In this pattern, each host implements a complete MAPE loop, whose local M, A, P and E components coordinate their operation with corresponding peer components of the other hosts. Ideally, this should lead to a flexible sharing of information about the status of the managed systems, as well as the results of the analysis. The triggering of possible adaptation actions is then agreed on and managed by the local P components, which then activate their local E components to execute the actions to the local managed subsystems. In practice, achieving a globally consistent view on the system status, and reaching a global consensus about suitable adaptation actions is not an easy task. In this case, it could be preferable to limit the interaction among peer control components to get some partial information sharing and some kind of lose coordination. Generally, this may lead to sub-optimal adaptation actions, from the overall system viewpoint. However, depending on the system at hand and the corresponding adaptation goals, even local adaptation actions based on partial knowledge of the global system status may lead to achieve globally optimal goals (TCP adaptive control flow is a typical example of this).

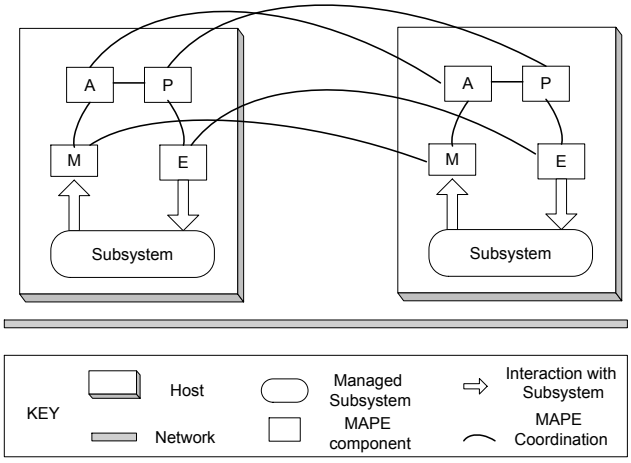

Figure 2: Decentralized pattern

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

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

4.4 Outlook

So far, the research community on self-adaptive and autonomic systems has spent little effort in studying the interactions among components of MAPE loops. Our position of making the control loops explicit underlines the need for a disciplined engineering practice in this area. Besides the consolidation of architecture knowledge in the form of different MAPE configurations as patterns, we also need practical interface definitions (signatures and APIs), message formats, and protocols. The necessity of such definitions has partially already been appreciated in the past, e.g., in [34] the authors standardize the communication from the A to the P component by using standard BPEL (Business Process Execution Language) as the data exchange format, but no comprehensive approach exists so far.

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

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

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

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

5. CONTROL SCIENCE FOR RUN-TIME V&V FOR SELF-ADAPTIVE SYSTEMS

In an impressive research report published in 2010, US Air Force (USAF) chief scientist Werner Dahm identified "control science" as as a top priority for the USAF science and technology research agenda for the next 20 years [54]. Control science develops verification and validation tools and techniques to allow humans to trust decisions made by autonomous or self-adaptive systems. According Dahm, the major barrier preventing the USAF from gaining more capability from autonomous systems is the lack of validation and verification (V&V) methods and tools. In other words run-time V&V methods and tools are critical for the success of autonomous, autonomic, smart, self-adaptive and self-managing systems.

To be able to establish "certifiable trust" when adapting to contextual and environmental changes at run-time we need

powerful and versatile V&V methods, techniques, and tools. One research approach is to adapt traditional development-time V&V methods to work at run-time — model checking is a good candidate in this realm. A complementary research direction is to ease or relax the traditional software engineering approach where we satisfy requirements outright to a more control engineering approach where we regulate functional and particularly non-functional requirements to stay within their viability zones using feedback loops [38]. To reason about viability zones effectively requires rich semantic models for run-time decision-making. The Models@Runtime community is particularly concerned with this research avenue and uses model-driven techniques for validating and monitoring run-time behavior [2].One of the most promising approaches is to develop hybrid approaches that combine these three key strategies. The term control science is an appropriate term to characterize this research realm.

State space explosion has long been a huge challenge in the design and implementation of practical V&V techniques. With the event of software-intensive systems with high levels of adaptability the problems of very large state spaces are greatly exacerbated. However, hybrid approaches that combine the regulation of viability zones, model-based reasoning, and practical V&V techniques are particularly promising in alleviating the large state space problem. These research directions constitute formidable challenges for the software engineering for adaptive and self-managing systems (SEAMS) community.

Software validation and verification (V&V) concerns the quality assessment of software systems throughout their lifecycle. The goal is to ensure that the software system satisfies its functional requirements and meets its expected quality attributes [3, 21]. As mentioned in Section 3, the realization of self-adaptation has gradually blurred the boundaries among the different software lifecycle phases in all of their dimensions. Thus, not only V&V tasks are amenable, or even required, to be performed in different phases (i.e., at run- and load-time instead of just design- and development-time), but also different, more efficient and refined V&V methods are needed in this setting (e.g., regulating the satisfaction of requirements).

While high levels of adaptability and autonomy result in obvious benefits to the stakeholders of software systems, realizing these abilities with confidence is hard. Designing and deploying certifiable V&V methods for highly adaptive systems is one of the major challenges facing the entire field. Understanding the underlying theoretical principles and the specific properties that govern dynamic adaptation and the various ways in which they can be realized may require a large part of this decade, if not more [54].

As mentioned above, one of the key problems related to traditional V&V approaches is the explosion of the state space. The number of possible input states that such systems can assume is so large that not only is it unattainable to check all of them directly but also infeasible to check more than a small fraction of them. Another key problem is the fact that it is impossible to verify a requirements specification, given the evolving nature of software requirements and their context-dependencies in self-adaptive systems [25]. The unpredictable nature of context changes that can affect adaptive software behavior is yet another key problem.

Therefore, the SEAMS community ought to develop a control science involving design-time and run-time V&V methods and techniques for self-adaptive and self-managing systems with inferential, incremental and compositional characteristics to provide adequate confidence levels and certifiable trust in self-adaptation of such systems.

5.1 Adaptation Goals and Requirements: V&V Starting Points

Traditionally, software quality assessment has been realized through a variety of strategies to cover the wide concept of "quality", defined in terms of the degree in which software conforms to its requirements and fulfills its intended goals [3]. These requirements and goals are expressed using different notations and formalisms, which are the actual starting points for V&V tasks. Nonetheless, self-adaptation goals and specific requirements introduce the necessity of classifying requirements into two system levels, according to the separation of concerns implied by the general MAPE-loop model [26]. The first level corresponds to the managed system to be dynamically adapted according to context changes, while the second, to the adaptation mechanism itself. In the first level, requirements express the main reason for the system to be self-adaptive, usually in terms of adaptation goals. These goals can be any of the self-* properties, as well as the continued satisfaction or regulation of functional, quality of service (QoS) or quality of experience (QoE) requirements, such as performance, availability, security, and user satisfaction. In the second level, the adaptation mechanism, the requirements are identified mainly on the preservation of adaptation properties such as robustness, stability, accuracy, settling-time, and overshoot.

For both levels, V&V methods and techniques, which are applied at design-, load- or run-time, must determine whether or not the software system meets its requirements and goals.

5.2 V&V Self-Adaptation Concerns

5.2.1 Uncertainty in Self-Adaptation

Uncertainty in the system operation and maintenance phase is possibly the most challenging concern for validation and verification of self-adaptive software. This implies the central role of dynamic monitoring in V&V methods and techniques for this kind of systems [43, 55]. Conventional V&V techniques have limited applicability, since they assume controlled conditions and enough time and computing resources to provide rigorous and thorough certification of "complete" system computational states before its deployment in production environments. However, self-adaptation targets environments with hard to predict, highly dynamic, and comparatively resource-constrained conditions. Moreover, these resources must be shared in the execution of the actual functional services provided by the managed system. Thus, run-time validation and verification must fulfill efficiency requirements not only to verify the system adaptation while maintaining the capacity of timely reaction to context changes without compromising system availability, but also to maintain the QoS or QoE requirements on the managed system services.

Under these settings, the accurate estimation of the safety operational region where the system operates without reaching undesirable states is not only challenging but crucial for the robustness of self-adaptive systems. The engineering of adaptive controllers for flight control systems provides exemplifying insights into this point. In particular, Schumann and Gupta proposed a V&V tool to calculate safety regions around the current state of an adaptive system, based on a Bayesian statistical approach [43]. With this approach, they can provide a measure of confidence on the probable adequacy of the system's model under particular situations.

5.2.2 Self-Adaptation Properties

V&V concerns for self-adaptation certification can be classified according to the two system levels introduced by self-adaptation's separation of concerns. The first is related to the certification of the managed system, while the second to the certification of the adaptation mechanism, with respect to their corresponding requirements and goals [49].

In this context, motivated by the discussions during the 2010 Dagstuhl Seminar 10431 on *Software Engineering for Self-Adaptive Systems,* and based on an extensive analysis of several self-adaptive approaches, Villegas et al. proposed a framework for evaluating self-adaptive systems [49]. This framework synthesizes the different properties that have been proposed for the managed system (i.e., adaptation goals) and the adaptation mechanisms (i.e., adaptation properties), as well as the adaptation goals and their relationship to quality attributes. Several of the synthesized adaptation properties are gleaned from control theory [19, 25] and reinterpreted for self-adaptive software. Moreover, the framework classifies adaptation properties according to *how* and *where* they are observed (cf. Table 1 [49]). Concerning how they are observed, some properties can be evaluated using static verification techniques while others require dynamic verification and run-time monitoring. With respect to where they are observed, properties can be evaluated on the managed system or on the adaptation controller (i.e., the adaptation mechanism). However, most properties can be observed only on the managed system even when they are used to evaluate the adaptation controller.

Table 1: Classification of adaptation properties according to how and where they are observed.

Adaptation Property	Property Verification Mechanism	Where the Property is Observed
Stability	Dynamic	Managed System
Accuracy	Dynamic	Managed System
Settling Time	Dynamic	Both
Small Overshoot	Dynamic	Managed System
Robustness	Dynamic	Controller
Termination	Static	Controller
Consistency	Both	Managed System
Scalability	Dynamic	Both
Security	Dynamic	Both

5.3 Run-Time V&V Research Challenges

The fundamental problems addressed by run-time V&V for self-adaptive systems can be argued to be identical to those of traditional, design-time V&V [15]. That is, independent of the self-* adaptation goals, context awareness and even uncertainty, V&V fundamentally aims at guaranteeing that a system meets its requirements and expected properties. However, a key differentiating factor between run-time and design-time V&V is that resource constraints such as time and computing power are more critical for run-time V&V. From these constraints, non-trivial challenges arise, and to tackle them we should depart of course from traditional V&V methods and techniques. Nonetheless, a general discussion of different traditional V&V techniques and their applicability to run-time V&V for self-adaptive systems may not be sensible at this level. On the one hand, the decision of whether using one verification technique or another should be made depending on the properties to be verified as well as the criticality of the system (e.g., in terms of safety). On the other hand, although fully automated techniques (e.g., model checking) might be the first choice for run-time V&V, there is no reason for not also using semi-automated or even manual techniques (e.g., theorem proving) and combining them in the same or different lifecycle phases of system development. Hence, instead of systematically analyzing the applicability of traditional V&V techniques to dynamic adaptation, we focus on some properties that these techniques could have, exemplifying them in specific techniques.

5.3.1 V&V Techniques: Desirable Properties

Even though traditional V&V techniques (e.g., testing, model checking, formal verification, static and run-time analysis, or program synthesis) have been used for the properties described in Sect. 5.2.2 (cf. Table 1), an important challenge is their integration into the self-adaptation lifecycle (i.e., at run-time). For this, we introduce yet another kind of properties — properties on V&V techniques including sensitivity, isolation, incrementality, and composability.

According to González et al., sensitivity and isolation refer to the level of run-time testability that an adaptive software system can support [18]. On the one hand, sensibility defines the degree with which run-time testing operations interfere with the running system service delivery. That is, the degree with which run-time V&V may affect the accomplishment of system requirements and adaptation goals. Instances of factors that can affect run-time test sensitivity are (i) component state, not only because run-time tests are influenced by the actual state of the system but because the state of the system could be altered as a result of test invocations; (ii) component interactions, as the run-time testability of a component may depend on the testability of the components it interacts with; (iii) resource limitations, because run-time V&V may affect non-functional requirements such as performance at undesirable levels; and (iv) availability, as run-time validation can be performed depending on whether testing tasks require exclusive usage of components with high availability requirements. On the other hand, González et al. also define isolation as the means to counteract run-time test sensitivity. Instances of techniques for implementing test isolation are (i) state separation (e.g., blocking the component operation while testing has place, performing testing on cloned components); (ii) interaction separation (e.g., blocking component interactions that may be propagated due to results of test invocations); (iii) resource monitoring (e.g.,

indicating that testing must be postponed due to resources unavailability); and (iv) scheduling (e.g., planning testing executions when involved components are less used).

5.3.2 Requirements at Run-Time

Concerning run-time validation, adaptive systems require suitable techniques to validate their conformance with requirements after behavioral and structural changes. In light of this, the application of run-time automatic testing techniques to enable adaptive software systems with self-testing capabilities seems to be a promising approach. An instance of this approach is the self-testing framework for autonomic computing systems proposed by King et al. [27]. This framework dynamically validates change requests in requirements using regression testing and customized tests to assess the behavior of the system under the presence of new added components. For this, autonomic managers designed for testing are integrated into the current workflow of autonomic managers designed for adaptation. Two strategies support the validation process: (i) safe adaptation with validation and (ii) replication with validation. In the first strategy, testing autonomic managers apply an appropriate validation policy during the adaptation process where involved managed resources are blocked until the validation is completed. If the change request is accepted, the corresponding managed resources are adapted. In the second strategy, the framework maintains copies of the managed resources for validation purposes. Thus, changes are implemented on copies, then validated and if they are accepted, the adaptation is performed. Testing policies can be also defined by administrators and loaded into the framework at run-time. This self-testing approach evidences the blurred boundaries among the software lifecycle phases and the many implications for V&V of self-adaptive software systems. Some of these implications constitute challenges that arise from requirements engineering. First, run-time specifications of system requirements are necessary to manage their lifecycle and its implications for run-time V&V. Second, requirements traceability becomes a crucial fact in order to identify incrementally what to validate, the requirements subset that has changed, and when. Moreover, test case priority further contributes to refine this incremental validation. Third, for context-aware requirements, specifications must explicitly define the environmental conditions that must be monitored at run-time. The Requirements@runtime research community provides valuable foundations (e.g., requirements reflection) to support requirements variability from the perspective of run-time V&V of self-adaptive software systems [42].

5.3.3 State Explosion and Unpredictability

Model checking has been used as an automated method for verifying concurrent software systems to overcome the multiple limitations and shortcomings of testing techniques. Given a correctness specification for a software system behavior in temporal logic (or any of its variants), the method uses efficient search procedures to systematically check the finite states graph representation of the system. The well known practical problem of this method is the state explosion that implies the representation of all of the states of system behavior.

In self-adaptive software, this problem is augmented given that, from the structural point of view, the changing nature of self-adaptive software can be abstracted in terms of its architecture (i.e., its elements and their relationships) reconfiguration. Thus, in contrast to the one structural static configuration of traditional software, model checking must be applied to each of the possible new configurations.

Some proposals can be used to guide the way to address the state explosion problem. For instance, [15] observes that software behavior in one state is not generally independent of previous states, for a given (structural) configuration. Therefore, for a given system property of interest, equivalence classes could be formed by grouping sets of a number of states representing the same information with respect to this property, thus reducing the state space to be checked. On the probabilistic side, the already mentioned proposal of Schumann and Gupta [43] could be used as inspiration to reduce this space by calculating the most probable set of next states around the current state of an adaptive system. Furthermore, if these techniques are composable and incremental, they could be used together to pave the way to cope with the problems of state explosion and unpredictability in V&V for self-adaptive software.

5.4 Outlook

The validation and verification of self-adaptive software systems at run-time is an urgent necessity and a huge challenge to be able to establish "certifiable trust" in practical adaptation mechanisms. While development V&V methods are necessary and play an important role in the quest towards achieving effective run-time V&V, but they are not sufficient. To reason effectively and provide assurances about the behavior of self-adaptive systems at run-time we need to resort control theory which deals with dynamical systems. The combination of theories and principles from software engineering and control theory used to establish certifiable trust in highly adaptive software intensive systems is called control science.

In this section, we discussed important challenges and possible roadblocks for run-time validation and verification of self-adaptive systems. First, the traceability of evolving requirements is crucial for the identification of what to validate, when, and the most appropriate V&V method for a particular requirement change. Second, run-time V&V techniques must exhibit desirable properties thus increasing their complexity. Third, dynamic instrumentation such as dynamic monitoring is also required to realize run-time V&V techniques.

The assessment of research approaches on self-adaptive software systems constitutes an important starting point for the development of standardized and objective certification methods. For this, we believe that the evaluation framework proposed by Villegas et al. provides useful guidance and insights [49]. The SEAMS community is ideally positioned to develop control science towards certifiable trust in self-adaptation.

6. LESSONS AND CHALLENGES

In this section, we present the overall conclusions of the research roadmap paper in the context of the 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 topics identified as key to software engineering of self-adaptive systems: design space for self-adaptive solutions, processes, from centralized to decentralized control, and practical run-time verification and validation. We now summarize the most important challenges for each topic.

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

There are several topics related to software engineering of self-adaptive systems that we did not cover, some of which we now mention. First, how to design self-adaptive system to handle change. For example, designs should provide some elasticity in order to be robust against some kinds of change, or they should be able to manage change by generating alternative solutions. Another issue related to system design is whether adaptation should be reactive or proactive. Further, how should competition and cooperation be managed? One of the key activities of feedback control loops in self-adaptive software systems is decision making, and its associated adaptation techniques and criteria for balancing, for example, quality of services, over-provisioning, and cost of ownership. We also did not cover technologies like model-driven development, aspect-oriented programming, and software product lines. These technologies might offer new opportunities and approaches in the development of self-adaptive software systems. Finally, we did not discuss exemplars — canonical problems and accompanying self-adaptive solutions — which are a likely stepping stone to the necessary benchmarks, methods, techniques, and tools to solve the challenges of engineering self-adaptive software systems.

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

7. REFERENCES

[1] L. Baresi and C. Ghezzi. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10)*, pages 17–22, New York, NY, USA, 2010. ACM.

[2] G. Blair, N. Bencomo, and R. B. France. Models@run.time: Guest Editors' Introduction. *IEEE Computer*, 42(10):22–27, 2009.

[3] P. Bourque and R. Dupuis. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society, 2005.

[4] N. Brake, J. R. Cordy, E. Dancy, M. Litoiu, and V. Popescu. Automating discovery of software tuning parameters. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 65–72, New York, NY, USA, 2008. ACM.

[5] F. P. Brooks. *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley Professional, 1st edition, 2010.

[6] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. In B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer Berlin / Heidelberg, 2009.

10.1007/978-3-642-02161-9-3.

[7] 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.

[8] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.

[9] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *FSE'10: Proceedings of the 2010 Foundations of Software Engineering conference*, pages 237–246, New York, NY, USA, 2010. ACM.

[10] A. Carzaniga, A. Gorla, and M. Pezzè. Self-healing by means of automatic workarounds. In *SEAMS'08: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 17–24, New York, NY, USA, 2008. ACM.

[11] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. D. M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[12] 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 on Autonomous and Adaptive Systems (TAAS)*, 1:223–259, December 2006.

[13] A. Elkhodary, N. Esfahani, and S. Malek. FUSION: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, pages 7–16, Santa Fe, NM, USA, 2010.

[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37:46–54, October 2004.

[15] E. Gat. Autonomy software verification and validation might not be as hard as it seems. In *Proceedings 2004 IEEE Aerospace Conference*, pages 3123–3128, 2004.

[16] I. Georgiadis, J. Magee, and J. Kramer. Self-Organising Software Architectures for Distributed Systems. In *1st Workshop on Self-Healing Systems*, New York, 2002. ACM.

[17] H. Ghanbari and M. Litoiu. Identifying implicitly declared self-tuning behavior through dynamic analysis. *Software Engineering for Adaptive and Self-Managing Systems, International Workshop on*, 0:48–57, 2009.

[18] A. González, E. Piel, and H.-G. Gross. A model for the measurement of the runtime testability of component-based systems. In *Proceedings of 2009 International Conference on Software Testing Verification and Validation Workshops*, pages 19–28. IEEE, 2009.

[19] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[20] IBM. An architectural blueprint for autonomic computing. Technical report, IBM, Jan 2006.

[21] IEEE. Industry implementation of international standard ISO/IEC 12207:95, standard for information technology-software life cycle processes. Technical report, IEEE, 1996.

[22] P. Inverardi. Software of the Future Is the Future of Software? In U. Montanari, D. Sannella, and R. Bruni, editors, *Trustworthy Global Computing*, volume 4661 of *Lecture Notes in Computer Science (LNCS)*, pages 69–85. Springer Berlin / Heidelberg, 2007.

[23] P. Inverardi and M. Tivoli. The Future of Software: Adaptation and Dependability. In *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science (LNCS)*, pages 1–31. Springer Berlin / Heidelberg, 2009.

[24] D. Ionescu, B. Solomon, M. Litoiu, and G. Iszlai. Observability and controllability of autonomic computing systems for composed web services. In *6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI 2011)*, 2011.

[25] S. A. Jacklin, M. R. Lowry, J. M. Schumann, P. P. Gupta, J. T. Bosworth, E. Zavala, and J. W. Kelly. Verification, validation, and certification challenges for adaptive flight-critical control system software. In *Proceedings of American Institute of Aeronautics and Astronautics AIAA Guidance Navigation and Control Conference and Exhibit*. American Institute of Aeronautics and Astronautics, 2004.

[26] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[27] T. M. King, A. E. Ramirez, R. Cruz, and P. J. Clarke. An integrated self-testing framework for autonomic computing systems. *Journal of Computers*, 2(9):37–49, 2007.

[28] B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an ontology of software maintenance. *Journal of Software Maintenance: Research and Practice*, 11(6):365–389, 1999.

[29] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering*, pages 259–268, 2007.

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

[31] C. Larman and V. R. Basili. Iterative and Incremental Development: A Brief History. *IEEE Computer*, 36(6):47–56, June 2003.

[32] M. M. Lehman. Software's Future: Managing

Evolution. *IEEE Software*, 15(01):40–44, 1998.

[33] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[34] F. Leymann. Combining Web Services and the Grid: Towards Adaptive Enterprise Applications. In J. Castro and E. Teniente, editors, *First International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05) - CAiSE workshop*, pages 9–21. FEUP Edi cões, June 2005.

[35] S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and G. Sukhatme. An architecture-driven software mobility framework. *Journal of Systems and Software*, 83(6):972–989, June 2010.

[36] S. Malek, M. Mikic-rakic, and N. Medvidovic. A decentralized redeployment algorithm for improving the availability of distributed systems. In *3rd International Conference on Component Deployment*, Grenoble, France, 2005.

[37] T. Mens. *Introduction and Roadmap: History and Challenges of Software Evolution*, chapter 1. Software Evolution. Springer, 2008.

[38] H. Müller, M. Pezzè, and M. Shaw. Visibility of control in adaptive systems. In *Proceedings of Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008*, pages 23–27. ACM/IEEE, 2008.

[39] Object Management Group (OMG). Software & Systems Process Engineering Meta-Model Specification (SPEM), Version 2.0, 2008.

[40] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14:54–62, May 1999.

[41] L. J. Osterweil. Software processes are software too. In *Proceedings of the 9th international conference on Software Engineering (ICSE '87)*, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[42] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems. a research agenda for RE for self-adaptive systems. In *Proceedings of 18th International Requirements Engineering Conference*, pages 95–103. IEEE, 2010.

[43] J. Schumann and P. Gupta. Bayesian verification & validation tools for adaptive systems: Report on principle of operation and prototypical implementation of bayesian envelope tool for neural networks. Technical report, National Aeronautics and Space Administration (NASA), 2006.

[44] D. E. Seborg, T. F. Edgar, D. A. Mellichamp, and F. J. Doyle III. *Process Dynamics and Control*. John Wiley & Sons, 3 edition, 1989.

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

[46] M. Shaw. The role of design spaces in software design. *(Submitted for publication)*, 2011.

[47] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering (ICSE '76)*, pages 492–497. IEEE Computer Society Press, 1976.

[48] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2 edition, 2006.

[49] N. Villegas, H. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *Proceedings of 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011)*. ACM, 2011. To appear.

[50] T. Vogel and H. Giese. Adaptation and Abstract Runtime Models. In *Proceedings of the 5th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010)*, pages 39–48. ACM, 2010.

[51] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Incremental Model Synchronization for Efficient Run-Time Monitoring. In S. Ghosh, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *LNCS*, pages 124–139. Springer Berlin / Heidelberg, 2010.

[52] P. Vromant, D. Weyns, S. Malek, and J. Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS11)*, Honolulu, Hawaii, 2011.

[53] D. Weyns, S. Malek, and J. Andersson. On decentralized self-adaptation: lessons from the trenches and challenges for the future. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 84–93, New York, NY, USA, 2010. ACM.

[54] W.J.A. Dahm United States Air Force Chief Scientist (AF/ST). Technology Horizons a Vision for Air Force Science & Technology During 2010-2030. Technical report, U.S. Air Force, 2010.

[55] S. Yerramalla, Y. Liu, E. Fuller, B. Cukic, and S. Gururajan. An approach to V&V of embedded adaptive systems. In M. Hinchey, J. Rash, W. Truszkowski, and C. Rouff, editors, *Formal Approaches to Agent-Based Systems*, volume 3228 of *LNCS*, pages 173–188. Springer Berlin / Heidelberg, 2005.