Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges

Danny Weyns

Abstract

Modern software systems are expected to operate under uncertain conditions, without interruption. Possible causes of uncertainties include changes in the operational environment, dynamics in the availability of resources, and variations of user goals. The aim of self-adaptation it to let the system monitor itself and based on its goals reconfigure or adjust itself to satisfy the changing conditions, or if necessary degrade gracefully. In this chapter, we provide a particular perspective on the evolution of the field of self-adaptation in six waves. These waves put complementary aspects of engineering self-adaptive systems in focus that synergistically have contributed to the current knowledge in the field. From the presented perspective on the field, we outline a number of challenges for future research in self-adaptation, both in a short and long term.

1 Introduction

Back in 1968, at the NATO Software Engineering Conference in Brussels, the term "software crisis" was coined, referring to the manageability problems of software projects and software that was not delivering its objectives [36]. One of the key identified causes at that time was the growing gap between the rapidly increasing power of computing systems and the ability of programmers to effectively exploit the capabilities of these systems. This crisis triggered the development of novel programming paradigms, methods and processes to assure software quality. While today large and complex software projects remain vulnerable to unanticipated problems, the causes that underlay this first software crisis are now relatively well under control of project managers and software engineers.

Thirty five years later, in 2003, IBM released a manifesto that referred to another "looming software complexity crisis" this time caused by the increasing complexity of installing, configuring, tuning, and maintaining computing systems [17]. New emerging computing systems at that time went beyond company boundaries into the Internet, introducing new levels of complexity that could hardly be managed, even by the most skilled system administrators. The complexity resulted from various internal and external factors, causing uncertainties that are difficult to anticipate before deployment. Examples are the scale of the

system, inherent distribution of the software system that may span administrative domains, dynamics in the availability of resources and services, system faults that may be difficult to predict, and changes in user goals during operation. In a seminal paper, Kephart and Chess put forward self-management as the only viable option to tackle the problems that underlie this complexity crisis [30]. Self-management refers to computing systems that can adapt autonomously to achieve their goals based on high-level objectives. Such computing systems are usually called *self-adaptive systems*.

As already stated by Kephart and Chess, realising the full potential of self-adaptive system will take "a concerted, longterm, and worldwide effort by researchers in a diversity of fields." Over the past two decades, researchers and engineers from different fields have put extensive efforts in the realisation of self-adaptive systems. In this chapter, we provide a particular perspective on the engineering of self-adaptive systems in six waves. Rather than providing a set of distinct approaches for engineering self-adaptive systems that have been developed over time, the waves put *complementary aspects of engineering self-adaptive systems* in focus that synergistically have contributed to the current body of knowledge in the field. Each wave highlights a trend of interest in the research community. Some of the (earlier) waves have stabilised now and resulted in common knowledge in the community. Other (more recent) waves are still very active and subject of debate; the knowledge of these waves has not been consolidated yet.

The first wave, *Automating Tasks*, stresses the role of self-management as a means to free system administrators and other stakeholders from the details of installing, configuring, tuning, and maintaining computing systems that have to run autonomously 24/7. The second wave, *Architecture-Based Adaptation* emphasises the central role of architecture in engineering self-adaptive systems, in particular the role architecture plays in separating the concerns of the regular functionality of the system from the concerns that are subject of the adaptation. The first two waves put the focus on the primary drivers for self-adaptation and the the fundamental principles to engineer self-adaptive systems.

The third wave, *Runtime Models* stresses the importance of adaptation mechanisms that leverage software models at runtime to reason about the system and its goals. In particular, the idea is to extend the applicability of models produced in traditional model-driven engineering approaches to the runtime context. The fourth wave, *Goal Driven Adaptation* put the emphasis on the requirements that need to be solved by the managing system and how they drive the design of a self-adaptive system and can be exploited at runtime to drive the self-adaptation proces. These two waves put the focus on key elements for the concrete realisation of self-adaptive systems.

The fifth wave, *Guarantees Under Uncertainties* stress the fundamental role of uncertainties as first-class concerns of self-adaptive systems, i.e., the lack of complete knowledge of the system and its executing conditions before deployment, and how these uncertainties can be resolved at runtime. Finally, the sixth wave, *Control-Based Approaches*, stress the solid mathematical foundation of control theory as a basis to design self-adaptive systems that have to

operate under a wide range of disturbances. The last two waves put the focus on uncertainties as key drivers of self-adaptive systems and how to tame them.

The remainder of this chapter is structured as follows. In Section 2, we explain the basic principles and concepts of self-adaptation. Section 3 presents the six waves in detail. Finally, we discuss a number of future challenges in Section 4, both in a short and long term.

2 Concepts and Principles

In this section, we explain what is a self-adaptive system. To that end, we define two basic principles that determine the notion of self-adaptation. These principles allow us to determine the scope of this chapter. From the two principles we derive a conceptual model of a self-adaptive system that defines the basic elements of such a system. The principles and the conceptual model provide the basis for the perspective on the engineering of self-adaptive systems in six waves that we present in the next section.

2.1 Basic Principles of Self-Adaptation

The term *self-adaptation* is not precisely defined in the literature. Cheng et al. refer to a self-adaptive system as a system that "is able to adjust its behaviour in response to their perception of the environment and the system itself" [14]. Brun et al. add that "the *self* prefix indicates that the system decides autonomously (i.e., without or with minimal interference) how to adapt or organise to accommodate changes in its context and environment" [10]. These interpretations take the stance of the external observer and look at a self-adaptive system as one that can handle changing external conditions, resources, workloads, demands, and failures.

Garlan et al. contrast traditional mechanisms that support self-adaptation, such as exceptions in programming languages and fault-tolerant protocols, with mechanisms that are realised by means of a closed feedback loop to achieve various goals by monitoring and adapting system behaviour at runtime [25]. Andersson et al. refer in this context to "disciplined split" as a basic principle of a self-adaptive system, referring to an explicit separation between a part of the system that deals with the domain concerns and a part that deals the adaptation concerns [2]. Domain concerns relate to the goals for which the system is built; adaptation concerns relate to the system itself, i.e., the way the system realises its goals under changing conditions. These interpretations take the stance of the engineer of the system and look at self-adaptation from the point of view how the system is conceived.

Hence, we introduce *two basic principles* that complement one another and determine what is a self-adaptive system:

1. **External principle**: A self-adaptive system is a system that can handle changes in its environment, the system itself and its goals autonomously (i.e., without or with minimal human interference).

2. **Internal principle**: A self-adaptive system comprises two distinct parts: the first part interacts with the environment and is responsible for the domain concerns (concerns for which the system is built); the second part interacts with the first part and is responsible for the adaptation concerns (concerns about the domain concerns).

In contrast to self-adaptive systems that comprise of two distinct parts compliant with the internal principle, adaptation can also be realised in other ways. In self-organising systems, components apply local rules to adapt their interactions in response to changing conditions and cooperatively realise adaptation. This approach often involves emergent behaviour [19]. Another related approach is context-awareness [3], where the emphasis is on handling relevant elements in the physical environment as a first-class citizen in system design and management. Context-aware systems typically have a layered architecture, where a context manager or a dedicated middleware is responsible for sensing and dealing with context changes. While self-organisation or context-awareness can be applied independently or can be combined with self-adaptation, the primary scope of this chapter is on self-adaptation as a property of a computing system that is compliant with the two basic principles of self-adaptation.

Furthermore, self-adaptation can be applied to different levels of the technology stack of computing systems, from the underlying hardware to low-level computing infrastructure, from middleware services to the application software. The challenges of self-adaptation at these different levels are different. For example, the design space for the adaptation of higher-level software entities is often multi-dimensional and software qualities and adaption objectives usually have a complex interplay [1, 10, 24]. These characteristics are less applicable to the adaptation of lower-level resources and hardware entities. The scope of this chapter is primarily on self-adaptation used to manage higher-level software elements of computing systems.

Prominent communities that have actively been involved in the research on self-adaptive systems and the waves presented in this article are the communities of Software Engineering of Adaptive and Self-Managing Systems (SEAMS)[1], Autonomic Computing (ICAC)[2], and Self-Adaptive and Self-Organising Systems (SASO)[3]. Research results on self-adaptation are regularly presented at the top software engineering conferences, including the International Conference on Software Engineering(ICSE)[4] and the International Symposium on the Foundations of Software Engineering (FSE).[5]

[1] https://www.hpi.uni-potsdam.de/giese/public/selfadapt/seams/
[2] http://nsfcac.rutgers.edu/conferences/ac2004/index.html
[3] http://www.saso-conference.org/
[4] http://2016.icse.cs.txstate.edu/
[5] https://www.cs.ucdavis.edu/fse2016/

2.2 Conceptual Model of a Self-Adaptive System

We now describe a conceptual model of a self-adaptive system. The model describes a set of concepts and the relationship between them. The concepts that correspond to the basic elements of a self-adaptive system are kept abstract and general, but they comply with the two basic principles of self-adaptation. The conceptual model introduces a basic vocabulary for the field of self-adaptation and serves as a guidance for organising and focusing the knowledge of the field. Figure 1 shows the conceptual model of a self-adaptive system.

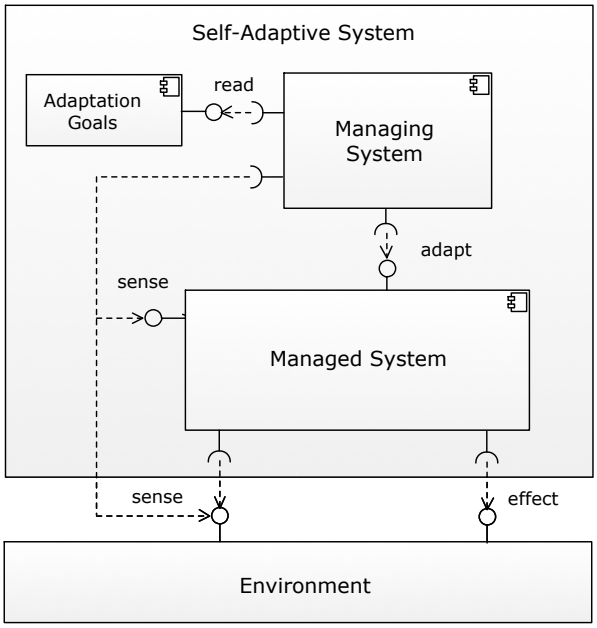

Figure 1: Conceptual model of a self-adaptive system

The conceptual model comprises *four basic elements*: environment, managed system, adaptation goals, and managing system.

Environment. The environment refers to the part of the external world with which the self-adaptive system interacts and in which the effects of the system will be observed and evaluated [29]. The environment can include both physical and virtual entities. For example, the environment of a robotic system includes physical entities like obstacles on the robot's path and other robots, as well as external cameras and corresponding software drivers. The distinction between the environment and the self-adaptive system is made based on the extent of control. For instance, in the robotic system, the self-adaptive system may interface with the mountable camera sensor, but since it does not manage (adapt) its functionality, the camera is considered to be part of the environment.

The environment can be sensed and effected through sensors and effectors respectively. However, as the environment is not under the control of the software engineer of the system, there may be uncertainty in terms of what is sensed by the sensors or what the outcomes will be of effecting the effectors.

Managed System. The managed system comprises the application code that realises the system's domain functionality. Hence, the concerns of the managed system are concerns over the domain, i.e. the environment. For instance, in the case of robots, navigation of a robot and transporting loads is performed by the managed system. To realise its functionality, the managed system senses and effects the environment. To support adaptations, the managed system has to equipped with sensors to enable monitoring and actuators to execute adaptations. Safely executing adaptations requires that the adaptation actions do not interfere with the regular system activity for which the system has to be in a quiescent state [32]. Different terms are used in the literature for the concept of managed system in the context of self-adaptation. For example, Kephart and Chess refer to it as the managed element [30], the Rainbow framework [25] calls it the system layer, Salehie and Tahvildari use core function [42], in the FORMS reference model, the managed system corresponds to the base-level subsystem [50], and Filieri et al. refer to it as controllable plant [23].

Adaptation Goals. The adaptation goals are concerns of the managing system over the managed system; they usually relate to the software qualities of the managed system [47]. Four principle types of high-level adaptation goals can be distinguished: self-configuration (i.e., systems that configure themselves automatically), self-optimisation (systems that continually seek ways to improve their performance or cost), self-healing (systems that detect, diagnose, and repair problems resulting from bugs or failures), and self-protection (systems that defend themselves from malicious attacks or cascading failures) [30]. As an example, a self-optimisation goal of a robot may be to ensure that a particular number of tasks are achieved within a certain time window under changing operation conditions, e.g., dynamic task loads or reduced bandwidth for communication. Adaptation goals are often expressed in terms of the uncertainty they have to deal with. Example approaches are the specification of quality of service goals using probabilistic temporal logics [13], the specification of fuzzy goals, whose satisfaction is represented through fuzzy constraints [5], and adding flexibility to the specification of goals by specifying the goals declaratively, rather than by enumeration [15]. We elaborate on goal modelling in the next section.

Managing System: The managing system manages the managed system. To that end, the managing system comprises the adaptation logic that deals with one or more adaption goals. For instance, a robot may be equipped with a managing system that allows the robot to adapt its navigation strategy to ensure that a certain number of tasks are performed within a given time window under changing operation conditions. To realise the adaptation goals, the managing system monitors the environment and the managed system and adapts the latter when necessary. Conceptually, the managing system may consist of multiple levels where higher-level adaptation subsystems manage underlying

subsystems. For instance, consider a robot that not only has the ability to adapt its navigation strategy, but also adapt the way such adaptation decisions are made, e.g., based on the energy level of the battery. Different terms are used in the literature for the concept of managing system. Examples are: autonomic manager [30], architecture layer [25], adaptation engine [42], reflective subsystem [50], and controller [23].

It is important to note that the conceptual model for self-adaptive systems abstracts away from distribution, i.e., the deployment of the software to hardware. Whereas a distributed self-adaptive system consists of multiple software components that are deployed on multiple nodes connected via some network, from a conceptual point of view such system can be represented as a managed system (that deals with the domain concerns) and a managing system (that deals with concerns of the managed system represented by the adaptation goals). The conceptual model also abstracts away from how adaptation decisions in a self-adaptive are made and potentially coordinated among different components. Such coordination may potentially involve human interventions, such as in socio-technical and cyber-physical systems. The conceptual model is invariant to self-adaptive systems where the adaptation functions are made by a single centralised entity or by multiple coordinating entities. Obviously, the distribution of the components of a self-adaptive system to hardware and the degree of decentralisation of decision making of adaptation will have a deep impact on how concrete self-adaptive systems are engineered.

3 An Organised Tour in Six Waves

In the previous section, the focus was on *what* is a self-adaptive system. We have explained the basic principles of self-adaptation and outlined a conceptual model that describes the basic elements of self-adaptive systems compliant with the basic principles. We direct our focus now on *how* self-adaptive systems are engineered. Specifically, we provide a concise but in-depth introduction to software engineering of self-adaptation. Rather than outlining distinct and comprehensive approaches for engineering self-adaptive systems that have been studied and applied over time, we take a different stance on the field and put different aspects of engineering self-adaptive systems in focus. These aspects are structured in six waves that emerged over time, often triggered by insights derived from other waves as indicated by the arrows in Figure 2.

The waves have contributed *complementary layers of knowledge* on engineering self-adaptive systems that synergistically have shaped the state of the art in the field. Waves highlight trends of interest in the research community. The knowledge consolidated in each wave is important for understanding the concept of self-adaptation and the principles that underlie the engineering of self-adaptive systems. Some waves are stabilised now and have produced knowledge that is generally acknowledged in the community, while other waves are still very active and the knowledge produced

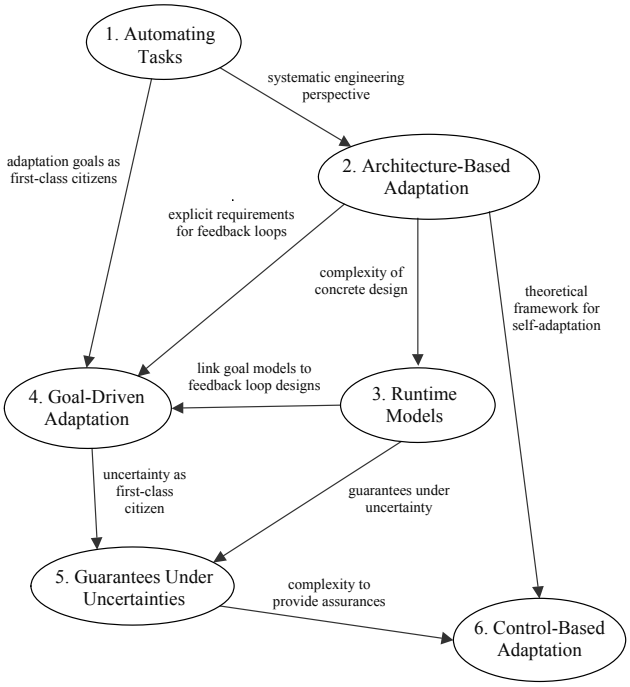

Figure 2: Six waves of research in self-adaptive systems; arrows indicate how waves have triggered new waves

in these waves has not been consolidated yet.

Figure 2 gives a schematic overview of the six waves. The first wave *Automating Tasks* is concerned with delegating complex and error-prone management tasks from human operators to the machine. The second wave *Architecture-based Adaptation* that is triggered by the need for a systematic engineering approach (from the first wave) is concerned with applying the principles of abstraction and separation of concerns to identify the foundations of engineering self-adaptive systems.

The third wave, *Runtime Models* that is triggered by the problem of managing the complexity of concrete designs of self-adaptive systems (from the second wave) is concerned with exploiting first-class runtime representations of the key elements of a self-adaptive system to support decision making at runtime. The fourth wave, *Goal-Driven Adaptation* is triggered by the need to consider requirements of self-adaptive systems as first-class citizens (from waves one and two) and link the goal models to feedback loop designs (from wave three). The fourth wave puts the emphasis on the requirements that need to be solved by the managing system and how they drive its design.

The fifth wave, *Guarantees under Uncertainty* is triggered by the need to deal with uncertainty as first-class citizen in engineering self-adaptive systems

Table 1: Summary of state-of-the-art before each wave with motivation, topic of the wave, and contributions enabled by each of the waves

Wave	SOTA before wave	Topic of wave	(To be) enabled by wave
W1	System management done by human operators is a complex and error prone process	Automation of management tasks	System manages itself autonomously based on high-level objectives
W2	Motivation for self-adaptation acknowledged, need for a principled engineering perspective	Architecture perspective on self-adaptation	Separation between change management (deal with change) and goal management (adaptation objectives)
W3	Architecture principles of self-adaptive systems understood, concrete realisation is complex	Model-driven approach to realise self-adaptive systems	Runtime models as key elements to engineer self-adaptive systems
W4	Design of feedback loops well understood, but requirements problem they intent to solve is implicit	Requirements for feedback loops	Languages and formalisms to specify requirements for self-adaptive systems
W5	Mature solutions for engineering self-adaptive systems, but uncertainty handled in ad-hoc manner	The role of uncertainty in self-adaptive systems and how to tame it	Formal techniques to guarantee adaptation goals under uncertainty
W6	Engineering of MAPE-based self-adaption well understood, but solutions are often complex	Applying principles from control-theory to realise self-adaptation	Theoretical framework for (particular types of) self-adaptive systems

(from wave four) and how to mitigate the uncertainty (from wave three). The fifth wave is concerned with providing trustworthiness for self-adaptive systems that need to operate under uncertainties. Finally, the sixth wave *Control-Based Adaptation* is triggered by the complexity to provide assurances (from wave five) and the need for a theoretical framework for self-adaptation (from wave two). The sixth wave is concerned with exploiting the mathematical basis of control theory for analysing and guaranteeing key properties of self-adaptive systems.

Table 1 provides a short summary with the state-of-the-art before each wave and a motivation, the topics that are studied in the different waves, and the contributions that are enabled by each of the waves. We discuss the waves now in detail based on a selection of highly relevant work.

3.1 Wave I. Automating Tasks

The first wave focusses on the automation of management tasks, from human administrators to machines. In the seminal paper [30], Kephart and Chess elaborate on the problem that the computing industry experienced from the early 2000s and that underlies the need for self-adaptation: the difficulty of

managing the complexity of interconnected computing systems. Management problems include installing, configuring, operating, optimising, and maintaining heterogeneous computing systems that typically span multiple administrative domains.

To deal with this difficult problem, the authors outline a new vision on engineering complex computing system that they coin as *autonomic computing*. The principle idea of autonomic computing is to free administrators from system operation and maintenance by letting computing systems manage themselves given high-level objectives from the administrators. This idea is inspired by the autonomic nervous system that seamlessly governs our body temperature, hearth beat, breathing, etc. Four essential types of self-management problems can be distinguished as shown in Table 2.

Table 2: Types of self-management

Type	Example Problem	Example Solution
Self-configuration	New elements need to be integrated in a large Internet-of-Things application. Installing, configuring, and integrating heterogeneous elements is time consuming and error prone.	Automated integration and configuration of new elements following high-level policies. The rest of the network adapts automatically and seamlessly.
Self-optimisation	A web service infrastructure wants to provide customers a particular quality of service, but the owner wants to reduce costs by minimising the number of active servers.	The infrastructure continually seeks opportunities to improve quality of service and reduce costs by (de-)activating services and change the allocation of tasks to servers dynamically.
Self-healing	A large-scale e-health system provides various remote services to elderly people. Determining problems in such heterogeneous system is complex.	The system automatically detects anomalies, diagnoses the problem, and repairs local faults or adapts the configuration to solve the problem.
Self-protection	A web e-commerce application is vulnerable to attacks, such as illegal communications. Manually detecting and recovering from such attacks is hard.	The system automatically anticipates and defends against attacks, anticipating cascading system failures.

An autonomic computing system supports a continuous process, i.e., the system continuously monitors itself and based on a set of high-level goals adapts itself to realise the goals. The primary building block of an autonomic system is an *autonomic manager*, which corresponds to the managing system in the conceptual model of a self-adaptive system. Figure 3 shows the basic elements of an autonomic manager. The four elements: Monitor, Analyse, Plan, and Execute realise the basic functions of any self-adaptive system. These elements share common Knowledge, hence the model of an autonomic manager is often referred to as the MAPE-K model.

The *Monitor* element acquires data from the managed element and its en-

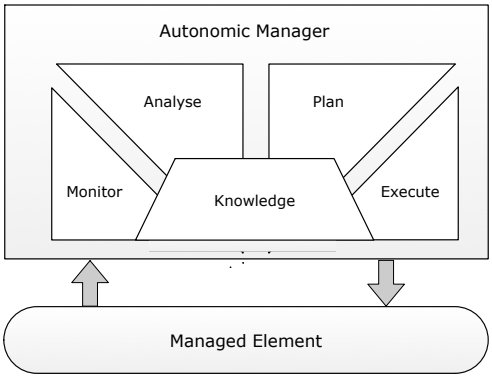

Figure 3: Structure of autonomic manager (based on [30])

vironment, and processes this data to update the content of the *Knowledge* element accordingly. The *Analyse* element uses the up-to-date knowledge to determine whether there is a need for adaptation of the managed element. To that end, the analyse element uses representations of the adaptation goals that are available in the knowledge element. If adaptation is required, the *Plan* element puts together a plan that consists of one or more adaptation actions. The adaptation plan is then executed by the *Execute* element that adapts the managed element as needed. MAPE-K provides a reference model for an managing system. MAPE-K's power is its intuitive structure of the different functions that are involved in realising the feedback control loop in a self-adaptive system.

While the distinct functions of a managing system are intuitive, the concrete realisation of these functions offers significant scientific and engineering challenges. We illustrate some of these challenges with a Web-based client-server system, borrowed from the paper that introduces the Rainbow framework [25]. Figure 4 shows the setup.

The system consists of a set of Web clients that make stateless requests of content to server groups. Each server group consists of one or more servers. Clients connected to a server group send requests to the group's shared request queue, and servers that belong to the group take requests from the queue. The adaptation goal is to keep the perceived response time of each client (self.responseTime) below a predefined maximum (maxResponseTime).

The managing system (Architecture Layer) connects to the managing system (Client-Server System) through probes and effectors. The Model Manager (Monitor) uses probes to maintain an up-to-date architectural model of the executing system, i.e., a graph of interacting components with properties (i.e., clients and servers). Server load (ServerT.load) and available bandwidth (ServerT.bandwidth) are two properties that affect the response time (responseTime). The Constraint Evaluator (Analyse) checks the model periodically and triggers the Adaptation Engine (Plan) if the maximum response time is violated. If the adaptation goal is violated, the managing system executes an adaptation

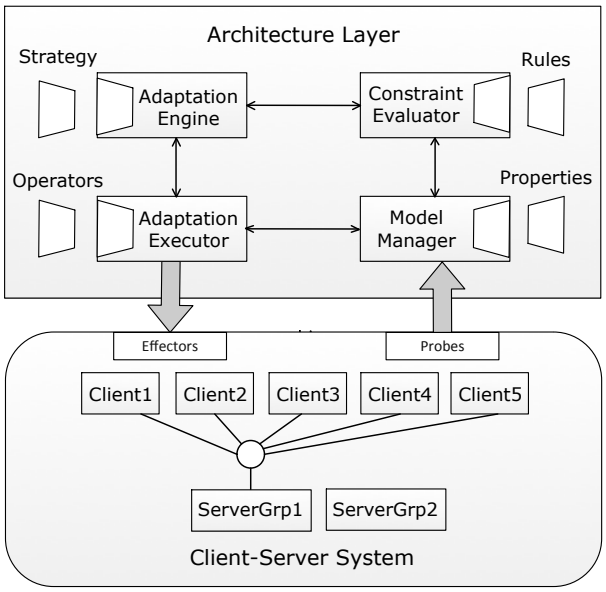

Figure 4: Web-based client-server system (based on [25])

strategy (responseTimeStrategy). This strategy works in two steps: if the load of the current server group exceeds a predefined threshold, it adds a server to the group decreasing the response time; if the available bandwidth between the client and the current server group drops too low, the client is moved to a group with higher available bandwidth lowering the response time. Finally, the Adaptation Executor (Execute) uses the operator ServerGroupT.addServer() to add an a ServerT to a ServerGroupT to increase the capacity, and the operator ClientT.move(from, toGroup) reconnects ClientT to another group (toGroup).

In the Rainbow paper [25], Garlan and his colleagues claim that external control mechanisms that from a closed control loop provide a more effective engineering solution than internal mechanisms. The motivation for this claim is based on the observation that external mechanisms localise the concerns of problem detection and resolution in separate modules that can be analysed, modified, extended, and reused across different systems. However, it took 10 years before the first empirical evidence was produced that underpins this claim [49].

Table 3 summarises the key insights derived from Wave I.

3.2 Wave II. Architecture-Based Adaptation

The second wave directs the focus from the basic motivation for self-adaption to the foundational principles to engineer self-adaptive systems. The pioneering approaches described in the first wave specify solutions at a higher level of abstraction, for examplee, the MAPE-K model. However, these approaches do

Table 3: Key insights of Wave I: Automating Tasks

• Automating tasks is a key driver for self-adaptation. This driver originates from the difficulty of managing the complexity of interconnected computing systems.
• The four essential types of self-management problems are self-configuration, self-optimisation, self-healing, and self-protection.
• Monitor, Analyse, Plan, Execute + Knowledge, MAPE-K in short, provides a reference model for an managing system.
• The MAPE-K functions are intuitive, however, their concrete realisation offers significant scientific and engineering challenges.

not provide an integrated perspective on how to engineer self-adaptive systems. In the second wave researchers apply fundamental design principles, in particular abstraction and separation of concerns, to identify the key concerns of self-adaptation. Understanding these concerns is essential for designers to manage the complexity of engineering self-adaptive systems and consolidate knowledge that can be applied to future designs.

Already in 1998, Oreizy et al. [37] stressed the need for a systematic, principled approach to support runtime change. These authors argued that software architecture can provide a foundation to support systematic runtime change. Central to this is an explicit architecture model of the system that is subject to adaptation, which is deployed at runtime together with the system and is used by a feedback control mechanism as a basis for change.

In their FOSE'07 paper [31], Kramer and Magee argue for an architecture-based approach to engineer self-adaptive software systems. Such an approach offers various benefits, including: *generality* of concepts and principles that apply to a wide range of domains, an appropriate *level of abstraction* to describe dynamic change of a system, the *potential for scalability* as architecture supports composition and hiding techniques, *leverage on existing work* of languages and notations that provide a rigorous basis to support reasoning at runtime, and the *potential for an integrated approach* as specifications at the architecture level typically support configuration, deployment and reconfiguration. Inspired by the flexibility and responsiveness of sense-plan-act types of architectures used in robotics, Kramer and Magee propose a simple yet powerful three-layer architecture model for self-adaptation, as shown in Figure 5.

The bottom layer, *Component Control*, consists of the interconnected components that provide the functionalities of the system. Hence, this layer corresponds to the managed system as described in the conceptual model of a self-adaptive system (see Figure1). This layer may contain internal mechanisms to adjust the system behaviour. However, to realise self-adaptation, component control needs to be instrumented with mechanisms to report the current status of the system to higher layers as well as mechanisms to support runtime modification, such as component addition, deletion, and reconnection.

The middle layer, *Change Management*, consist of a set of pre-specified plans. The middle layer reacts to status changes of bottom layer by executing plans

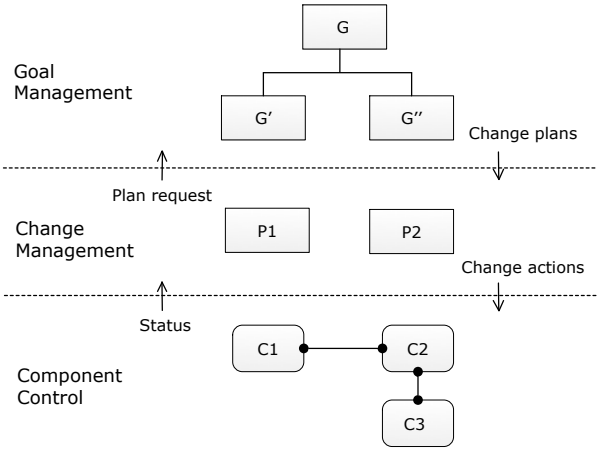

Figure 5: Three-layer architecture model for self-adapation (based on [31])

with change actions that adapt the component configuration of the bottom layer. The middle layer is also responsible for effecting changes to the underlying managed system in response to new objectives introduced from the layer above. Change management can adjust operation parameters of components, remove failed components, add new components, and change interconnections between components. If a condition is reported that cannot be handled by the available plans, the middle layer invokes the services of the top layer.

The top layer, *Goal Management*, comprises a specification of high-level goals. This layer produces change management plans in response to requests for plans from the layer beneath. Such a request will trigger goal management to identify alternative goals based on the current status of the system and generate plans to achieve these alternative goals. The new plans are then delegated to the change management layer. Goal management can also be triggered by stakeholders that introduce new goals. Representing high-level goals and automatically synthesising change management plans is a complex and often time consuming task.

The pioneering models shown in Figures 3, 4 and 5 capture foundational facets of self-adaptation. However, these models lack precision to reason about key architectural characteristics of self-adaptive systems, such as the responsibilities allocated to different parts of a self-adaptive system, the processes that realise adaptation together with the models they operate on, and the coordination between feedback loops in a distributed setting. A precise vocabulary for such characteristics is essential to compare and evaluate design alternatives. Furthermore, these models take a particular stance but lack an encompassing perspective of the different concerns on self-adaption. FORMS (FOrmal Reference Model for Self-adaptation) provides a reference model that targets these issues [50]. FORMS defines the essential primitives that enable software engi-

neers to rigorously describe and reason about the architectural characteristics of distributed self-adaptive systems. The reference model builds on established principles of self-adaptation. In particular, FORMS unifies three perspectives that represent three common but different aspects of self-adaptive systems: reflective computation, distributed coordination, and MAPE-K.

Figure 6 shows the reflection perspective in UML notation. For the formal representation of the three perspectives in Z notation we refer to [50]. To illustrate the FORMS model, we use a robotics application [21] shown in Figure 7. This application comprises a base station and a robot follower that trails a leader. Self-adaption in this system is used to deal with failures and to support dynamic updates.

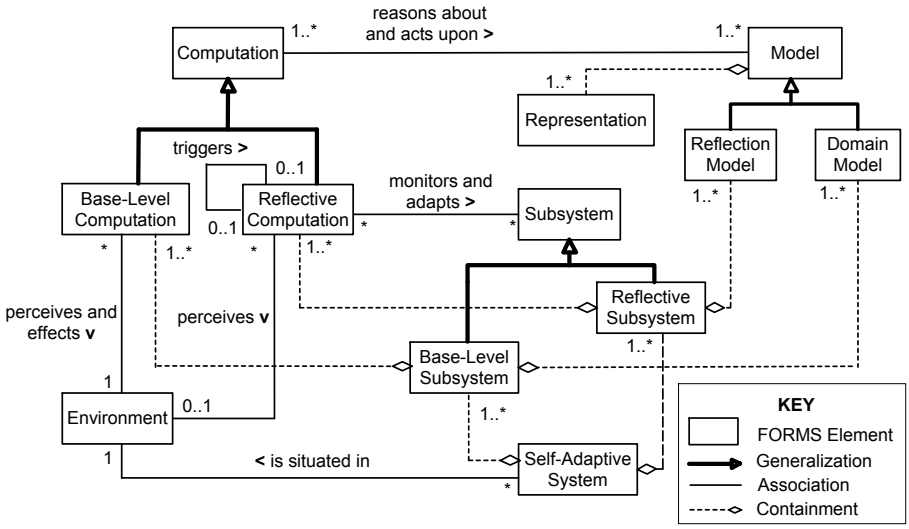

Figure 6: FORMS primitives for the reflection perspective [50]

As shown in Figure 6, a self-adaptive system is situated in an environment and comprises one or more base-level and reflective subsystems. The environment in the robotic application includes the area where the robots can move with lines that mark the paths the robots have to follow, the location of obstacles, and external sensors and cameras with the corresponding software drivers.

A base-level subsystem (i.e., managed system) provides the system's domain functionality; it comprises a set of domain models and a set of base-level computations, inline with principles of computational reflection. A domain model represents a domain of interest for the application logic (i.e., system's main functionality). A base-level computation perceives the environment, reasons about and acts upon a domain model, and effects the environment.

The base-level subsystem of the robots consists of two parts corresponding to the behaviours that realise the mission of the robots. The domain models incorporate a variety of information: a map of the terrain, locations of obsta-

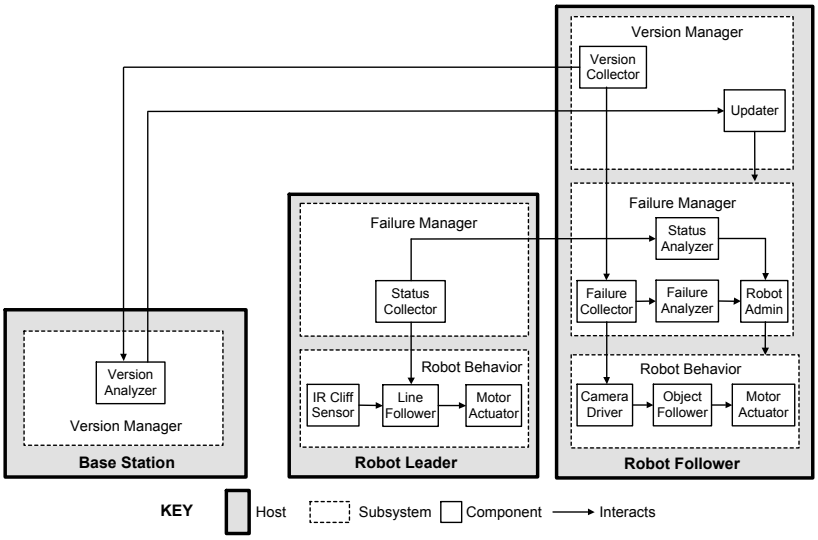

Figure 7: Robotics architecture presented [21]

cles and the other robot, etc. The base-level computation of the robot leader decides how to move the vehicle along a line, avoiding obstacles. The base-level subsystem of the follower moves the vehicle by tracking and following the leader.

A reflective subsystem (i.e. a managing system) manages another subsystem, which can be either a base-level or a reflective subsystem. A reflective subsystem consists of reflection models and reflective computations. A reflection model represents the entities (e.g., subsystem elements, environment attributes) needed for reasoning about adaptation. The reflection model often corresponds to the system's architectural models. A reflective computation reasons about and acts upon reflection models. A reflective computation also monitors the environment to determine when/if adaptations are necessary. However, unlike the base-level computation, a reflective computation does not have the ability to effect changes on the environment directly. The rationale is separation of concerns: reflective computations are concerned with a base-level subsystem, base-level computations are concerned with a domain.

The robot application comprises a reflective subsystem to deal with failures of the robot follower. This subsystem consists of failure managers deployed on the two robots that, based on the collected data, detect and resolve failures of the robotic behaviour of the follower. Reflection models include a runtime system architecture of the robot behaviour, adaptation policies, and plans (the models are not shown in Figure 7). Examples of reflective computations are the failure collector that monitors the camera driver and reports failures to failure analyzer that in turn determines the best replacement component for the camera based on adaptation policies. The failure manager layer is subject to additional version manager layer, which replaces the failure collector components on robot

follower nodes whenever new versions are available.

For the integration of the distributed coordination and MAPE-K perspective with the reflection perspective, and several examples that show how FORMS supports reasoning on the architecture of self-adaptive systems, we refer the interested reader to [50].

Table 4 summarises the key insights derived from Wave II.

Table 4: Key insights of Wave II: Architecture-Based Adaptation

• Architecture provides a foundation to support systematic runtime change and manage the complexity of engineering self-adaptive systems.
• An architecture perspective on self-adaptation provides: generality of concepts and principles, an appropriate level of abstraction, scalability, leverage on existing work, and an integrated approach.
• Two fundamental architectural concerns of self-adaptive systems are change management (i.e., manage adaptation using plans) and goal management (generate plans based on high-level goals).
• Three primary but interrelated aspects of self-adaptive systems are: reflective computation, MAPE-K, and distributed coordination.

3.3 Wave III. Models at Runtime

The second wave clarified the architecture principles that underlie self-adaptive systems. However, the concrete realisation of self-adaptation is complex. The third wave puts the concrete realisation of runtime adaptation mechanisms in focus. In an influential article, Blair et al. elaborate on the role of software models at runtime as an extension of model driven engineering techniques to the runtime context [7]. A model at runtime is defined as "a causally connected self-representation of the associated system that emphasises the structure, behaviour, or goals of the system from a problem space perspective."

The basic underlying motivation for runtime models is the need for managing the complexity that arises from the large amounts of information that can be associated with runtime phenomena. Compared to traditional computational reflection, runtime models of adaptive systems are typically at a higher level of abstraction and the models are causally connected to the problem space (in contrast to the computation space in reflection). The causal connection is important for two reasons: (1) the model should provide up-to-date information about the system to drive adaptations, and (2) if the models are causally connected to the system, adaptations can be made at the model level rather than at the system level. Runtime models provide abstractions of the system and its goals serving as a driver and enabler for automatic reasoning about system adaptations during operation.

Models at runtime can be classified along four key dimensions as shown in Table 5.

Building upon the notion of models at runtime, Morin et al. define a self-adaptive system as a set of configurations that are determined by a space of

Table 5: Dimensions of models at runtime (based on [7])

Type	Example Problem
Structural versus behavioural	Structural models represent how the system or parts of it are organised, composed, or arranged together; behaviour models represent facets of the execution of the system, or observable activities of the system such as the response to internal or external stimuli.
Procedural versus declarative	Procedural models emphasis the *how*, i.e., they reflect the actual organisation or execution of the system; declarative models emphasis the *what*, i.e., they reflect the purpose of adaptation, e.g., in the form or explicitly represented requirements or goals.
Functional versus non-functional	Functional models reflect functions of the underlying system; non-functional models reflect quality properties of the system related to some functionality; e.g., a model keeps track of the reliability of a service.
Formal versus non-formal	Formal models specify the system or parts of it using a mathematical language, supporting automated reasoning; informal models reflect the system using e.g., a programming or domain modelling language.

variation points [35]. Depending on changing conditions (changes in the context, errors, etc.), the system dynamically chooses suitable variants to realise the variation points, changing it from one configuration to another.

Consider as an example a dynamic customer relationship management system that provides accurate client-related information depending on the context. When a user is working in his office, the system can notify him by e-mail via a rich Web-based client. When the user is driving a car to visit a client, messages received by a mobile or smart phone should notify only client-related or critical issues. If the user is using a mobile phone, he or she can be notified via the short message service or audio/voice. In the case the user uses a smart phone the system can use a lightweight Web client.

As these examples illustrate, the variants may provide better quality of service, offer new services that were not relevant under previous conditions, or discard services that are no longer useful. It is essential that transitions between configurations follow a safe migration path. Figure 8 shows the primary elements of a model-oriented architecture that realises this perspective.

The model-oriented architecture that corresponds with the managing system of the conceptual model of a self-adaptive systems consists of three layers. The top layer *Online Model Space* is a platform-independent layer that only manipulates models. The middle layer *Causal Connection* is platform-specific and links the model space to the runtime space. Finally, the bottom layer *Business Application* contains the application logic and is equipped with sensors that track runtime events from the application and its environment, and factories that can instantiate new component instances.

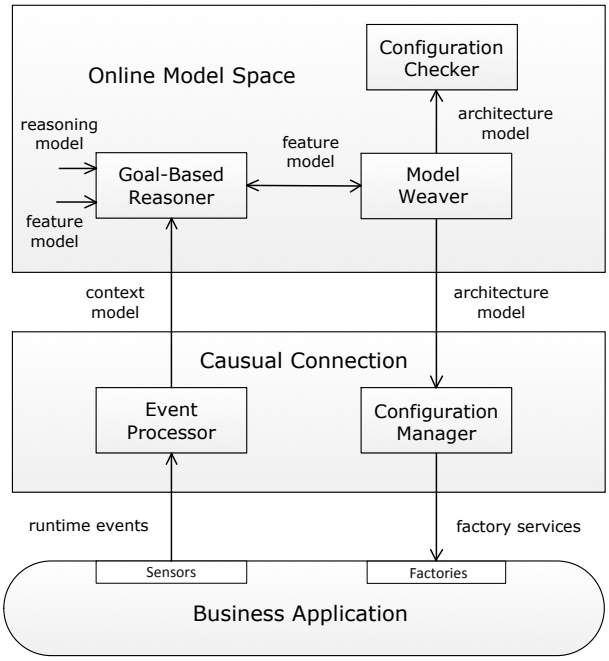

Figure 8: Model-oriented architecture for self-adaptive systems (based on [35])

The five components of the model-oriented architecture interact by exchanging four types of runtime models. The *feature model* describes the variability of the system, including mandatory, optional, and alternative, and constraints among features (requires, excludes). Features refer to architectural fragments that realise the features using a particular naming convention. The *context model* specifies relevant variables of the environment in which the system executes. Context variables are kept up to date at runtime based on sensor data. The *reasoning model* associates sets of features with particular context. One possible instantiation of a reasoning model is a set of event-condition-action rules. The event specifies the signal that triggers the invocation of the rule, e.g. a particular service fails. The condition part provides a logical expression to test whether the rule applies or not, e.g. the functionality of the failed service is required in the current context. The action part consists of update actions that are invoked if the rule applies, e.g. unbind the failed service and bind a new alternative service. Finally, the *architecture model* specifies the component composition of the application. The architecture model refines each leaf feature of the feature model into a concrete architectural fragment.

The *Event Processor* observes runtime events from the system and its context to update a context model of the system. Complex event processing entities can be used to aggregate data, remove noise, etc. When the *Goal-Based Reasoner*

receives an updated context model, it uses the feature model and reasoning model to derive a specific feature model with mandatory features and selected optional features aligned with the current context. The *Model Weaver* uses the specific feature model to compose an updated architecture model of the system configuration. The *Configuration Checker* checks the consistency of the configuration at runtime, which includes checking generic and user-defined application-specific invariants. If the configuration is valid, the model weaver sends it to the *Configuration Manager* that will reconfigure the architecture of the business application accordingly. Such a configuration includes deducing a safe sequence of reconfiguration actions such as removing, adding and binding components.

The model-oriented architecture for self-adaptive systems emphasises the central role of runtime models in the realisation of a self-adaptive systems. The modularity provided by the models at runtime allows to manage potentially large design spaces in an efficient manner.

Table 6 summarises the key insights derived from Wave III.

Table 6: Key insights of Wave III: Models at Runtime

• A model at runtime is a causally connected self-representation of the structure, behaviour, or goals of the associated system.
• Runtime models enable managing the complexity that arises from the large amounts of information that can be associated with runtime phenomena.
• Four key dimensions of runtime models are; structural versus behavioural, procedural versus declarative, functional versus non-functional, and formal versus non-formal.
• From a runtime model viewpoint, a self-adaptive system can be defined as a set of configurations that are determined by a space of variation points. Self-adaptation then boils down to choosing suitable variants to realise the variation points, providing better quality of service for the changing context.

3.4 Wave IV. Goal Driven Adaptation

The fourth wave turns the focus of research from the design of the managing system to the *requirements* for self-adaptive systems. When designing feedback loops, it is essential to understand to requirements problem they intent to solve. A pioneering approach for the specification of requirements for self-adaptive systems is RELAX [52]. RELAX is a language that includes explicit constructs for specifying and dealing with uncertainties. In particular, the RELAX vocabulary includes operators that define constraints on how a requirement may be relaxed at runtime. The grammar provides clauses such as "AS CLOSE AS POSSIBLE TO" and "AS FEW AS POSSIBLE." As an example, the requirement "The system SHALL ensure a minimum of liquid intake" can be relaxed to "The system SHALL ensure AS CLOSE AS POSSIBLE TO a minimum of liquid intake; the system SHALL ensure minimum liquid intake EVENTUALLY." The relaxed requirement tolerates the system temporally not to monitor a person's intake of liquid, but makes sure that it is eventually satisfied not to jeopardise the person's

health. A related approach is FLAGS [4] that is based on KAOS, a goal-oriented approach for modelling requirements. FLAGS distinguishes between crisp goals, whose satisfaction is boolean, and fuzzy goals, whose satisfaction is represented through fuzzy constraints.

Cheng et al. unite the RELAX language with goal-based modelling, explicitly targeting environmental uncertainty factors that may impact the requirements of a self-adaptive system [15]. Figure 9 shows excerpts that illustrates two mechanisms to mitigate uncertainties.

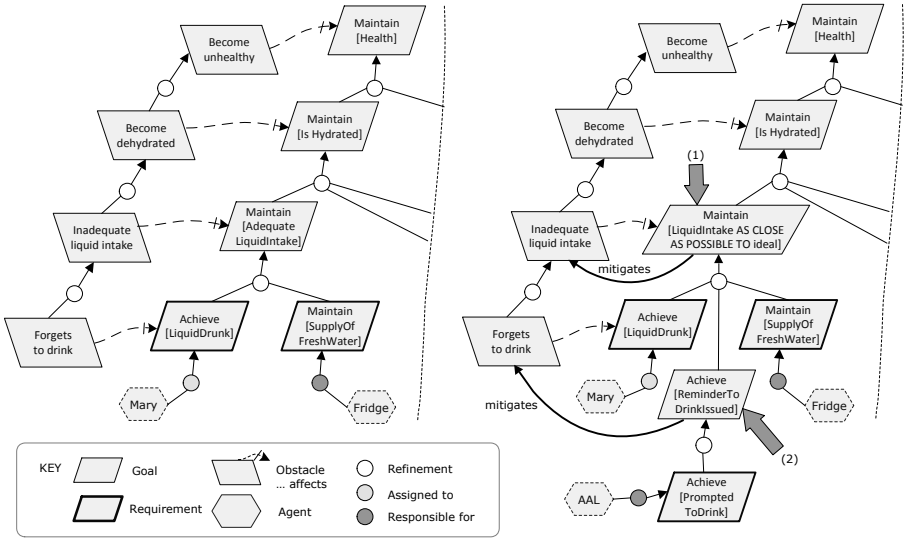

Figure 9: Left: original goal model. Right: goal model with two types of uncertainty mitigations: (1) relaxing a goal; (2); adding a subgoal (based on [15])

The first mechanism to mitigate uncertainty is relaxing a goal. For example, if the goal Maintain[AdequateLiquidIntake] cannot be guaranteed in all circumstances, e.g. based on uncertainties of Mary's behaviour, this uncertainty may be tolerated. To that end, RELAX is applied to the original goal resulting in Maintain[LiquidIntake AS CLOSE AS POSSIBLE TO ideal]. The arc pointing to the obstacle "Inadequate liquid intake" indicates a partial mitigation.

The second mechanism to mitigate uncertainty factors is adding a subgoal. The uncertainty whether Mary will drink enough is mitigated by adding the new sub-goal Achieve[ReminderToDrinkIssued]. This new goal is combined with the expectation that Mary drinks and that the fridge supplies fresh water. The reminders to drink are realised by an AAL system that is responsible for prompting Mary to drink based, i.e. requirement Achieve[PromptedToDrink].

Another mechanism to mitigate uncertainties is adding a new high-level goal for the target system. The interested reader is referred to [15] for a detailed discussion of this mitigation mechanism.

Approaches such as RELAX and FLAGS contribute with languages and notations to specify the goals for self-adaptive systems. Other researchers approach the problem of *requirements* for self-adaptive systems from a different angle and look at requirements as drivers for the design of the managing system. Souza et al. phrase it as "if feedback loops constitute an (architectural) solution for self-adaption, what is the requirements problem this solution is intended to solve?" [44]. The conclusion is that requirements of feedback loops (i.e. the concerns of the managing system in the conceptual model) are requirements about the runtime success/failure/quality-of-service of other requirements (i.e. the requirements of the managed system). These requirements are called *awareness requirements*. Table 7 shows different types of awareness requirements. The illustrative examples are from an ambulance dispatching system.

Table 7: Types of awareness requirements (based on [44])

Type	Illustrative example
Regular	AR1: Input emergency information should never fail.
Aggregate	AR2: Search call database should have a 95% success rate over one week periods.
Trend	AR3: The success rate of the number of unnecessary extra ambulances for a month should not decrease, compared to the previous month, two times consecutively.
Delta	AR4: Update arrival at site should be successfully executed within 10 minutes of the successful execution of Inform driver, for the same emergency call.
Meta	AR5: AR2 should have 75% success rate over one month periods.

A regular awareness requirement refers to another requirement that should never fail. An aggregate awareness requirement refers to another requirement and imposes constraints on their success/failure rate. AR3 is a trend awareness requirement that compares the success rates over a number of periods. A delta awareness requirement specifies acceptable thresholds for the fulfilment of requirements, such as achievement time. Finally, meta awareness requirements make statements about other awareness requirements. The constraints awareness requirements place are on instances of other requirements.

Awareness requirements can be graphically represented as illustrated in Figure 10. The figures shows an excerpt of a goal model for an ambulance dispatching system with awareness requirements AR1, AR2, and AR5.

In order to reason about awareness requirements they need to be rigorously specified and become first class citizens that can be referred to. The following excerpt shows how example requirement AR2 in Table 7 can be specified in the Object Constraint Language extended with temporal operators and other constructs such as scopes and timeouts:

```
context Goal-SearchCallDataBase
  def: all : Goal-SearchCallDataBase.allInstances()
  def: week: all -> select(...)
```

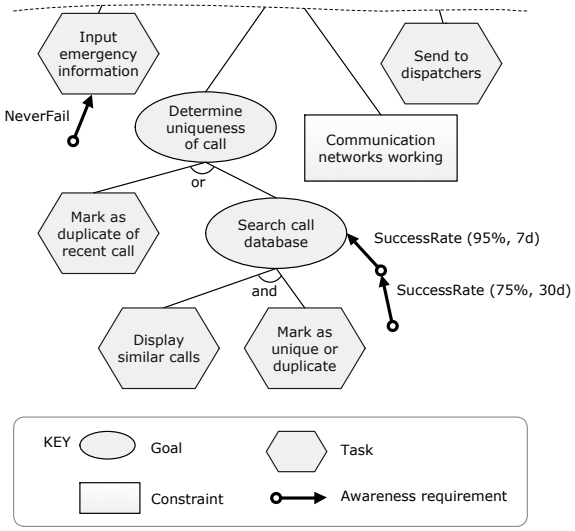

Figure 10: Graphical representation of awareness requirements (based on [44])

```
def: success : week -> select(...)
inv AR2: always(success -> size() / week -> size() >= 0.95)
```

The first line states that for AR2, all instances of the goal Goal-SearchCallDataBase are collected in a set. The next two lines use the *select()* operator to separate the subset of instances per week and the subset of these instances that succeeded. Finally, the sizes of these two sets are compared to assert that 95% of the instances are successful at all times (always).

Souza et al. [44] demonstrate how awareness requirements can be monitored at runtime using a monitoring framework. Monitoring of awareness requirements enables analysis of the behaviour of the system during operation, supporting the decision making for adaptation at runtime. In complementary work [45], the authors introduce the notion of evolution requirements that are modeled as condition-action rules, where the actions involve changing (strengthening, weakening, abandoning, ...) other requirements.

Table 8 summarises the key insights derived from Wave IV.

3.5 Wave V. Guarantees Under Uncertainties

In the fourth wave, uncertainty emerged as an important concern that self-adaptive systems need to deal with. The fifth wave puts the emphasis on taming uncertainty, i.e., providing *guarantees* for the compliance of the adaption goals of self-adaptive systems that operate under uncertainty. The fifth wave introduces a shift in the motivation for self-adaptation: uncertainty becomes the central driver for self-adaptation.

Table 8: Key insights of Wave IV: Goal Driven Adaptation

• Goal driven adaptation has two sides: (i) how to specify the requirements of a system that is exposed to uncertainties, and (2) if feedback loops constitute a solution for adaptation, what is the requirements problem this solution is intended to solve?
• Specifying goals of self-adaptive systems requires taking into account the uncertainties to which the system is exposed to.
• Defining constraints on how requirements may be relaxed at runtime enables to handle uncertainties.
• Requirements of feedback loops (i.e. the concerns of the managing system) are requirements about the runtime success/failure/quality-of-service of other requirements (i.e. the requirements of the managed system).
• Making requirements first class citizens at runtime enables analysis of the behaviour of the system during operation, supporting the decision making for self-adaptation.

Researchers and engineers observe that modern software systems are increasingly embedded in an open world that is constantly evolving, because of changes in the requirements, the surrounding environment, and the behaviour of users. These changes are difficult to anticipate at development time, the applications themselves need to change [4]. Consequently, in self-adaptive systems change activities are shifted from development time to runtime, and the responsibility for these activities is shifted from software engineers or system administrators to the system itself. Multiple researchers have pointed out that the primary underlying cause for this shift stems from uncertainty [22, 40, 48].

Different sources of uncertainty in self-adaptive systems have been identified as shown in Table 9. This table classifies the sources in four groups: sources of uncertainty related to the system itself, uncertainty related to the system goals, uncertainty in the execution context, and uncertainty related to human aspects.

Exposing self-adaptive systems – in particular systems with strict goals – to uncertainty introduces a paradoxical challenge: how can one provide guarantees for the goals of a system that is exposed to continuous uncertainty?

A pioneering approach that deals with this challenge is quantitative verification at runtime. Calinescu et al. apply this approach in the context of managing the quality of service in service-based systems [13].

Figure 11 shows the architecture of the approach that is called QoSMOS (Quality of Service Management and Optimisation of Service-based systems). The service based system offers clients remote access to a composition of Web-services through a workflow engine. To that end, the workflow engine executes services in a workflow. The functionality of each service may be provided by multiple service instances but with different qualities, e.g. reliability, response time, cost, etc. The aim of the system is to provide users the functionality of the composite service with particular qualities.

The adaptation problem is to select concrete services that compose a QoSMOS service and allocate resources to concrete services such that the required qualities are guaranteed. Given that the system is subject to several uncertainties, such as fluctuations in the availability of concrete services, changes in the

Table 9: Sources of uncertainty (based on [34])

Group	Source of uncertainty	Explanation
System	Simplifying assumptions	Refers to modelling abstractions that introduce perse some degree of uncertainty.
	Model drift	Misalignment between elements of the system and their representations.
	Incompleteness	Some parts of the system or its model are missing that may be added at runtime.
	Future parameters value	Uncertainty of values in the future that are relevant for decision making.
	Automatic learning	Learning with imperfect and limited data, or randomness in the model and analysis.
	Adaptation functions	Imperfect monitoring, decision making, and executing functions for realising adaption.
	Decentralisation	Lack of accurate knowledge of the entire system state by distributed parts of it.
Goals	Requirements elicitation	Elicitation of requirements is known to be problematic in practice.
	Specification of goals	Difficulty to accurately specify the preferences of stakeholders.
	Future goal changes	Changes in goals due to new customers needs, new regulations or new market rules.
Context	Execution context	Context model based on monitoring mechanisms that might not be able to accurately determine the context and its evolution.
	Noise in sensing	Sensors/probes are not ideal devices and they can provide (slightly) inaccurate data.
	Different sources of information	Inaccuracy due composing and integrating data originating from different sources.
Humans	Human in the loop	Human behaviour is intrinsically uncertain; it can diverge from the expected behaviour.
	Multiple ownership	The exact nature and behaviour of parts of the system provided by different stakeholders may be partly unknown when composed.

quality properties of services, etc. the requirements are necessarily expressed with probabilities. An example is R_0 :"the probability that an alarm failure ever occurs during the lifetime of the system is less than P = 0.13."

The core of the QoSMOS architecture is an *Autonomic Manager* that interacts with the service-based system through *sensors* and *effectors*. The autonomic manager comprises of a classic MAPE loop that exploits a set of runtime models to make adaptation decisions.

The *Monitor* tracks: (1) quality properties, such as the performance (e.g. response time) and reliability (e.g. failure rate) of the services, and (2) the resources allocated to the individual services (CPU, memory, etc.) together with their workload. This information is used to update the *operational model*. The types of operational models supported by QoSMOS are different types of Markovian models. Figure 12 shows an excerpt of a Discrete Time Markov

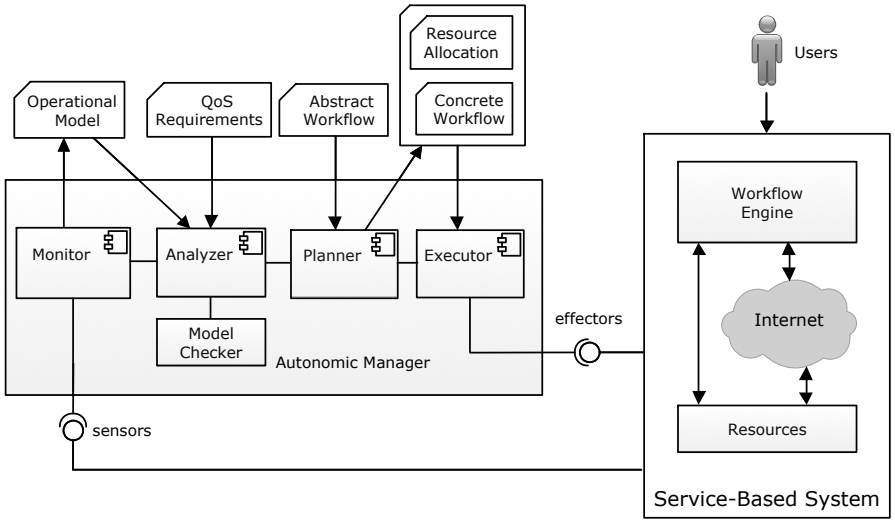

Figure 11: QoSMOS architecture (based on [13])

Chain (DTMC) model for a tele-assistance application. In particular, the model shows a part the workflow of actions with probabilities assigned to branches. The initial estimates of these probability values are based on input from domain experts. The monitor updates the values at runtime, based on observations of the real behaviour. Failure probabilities to service invocations (e.g., c in the model) are modeled as variables because these values depend on the concrete service selected by the MAPE loop.

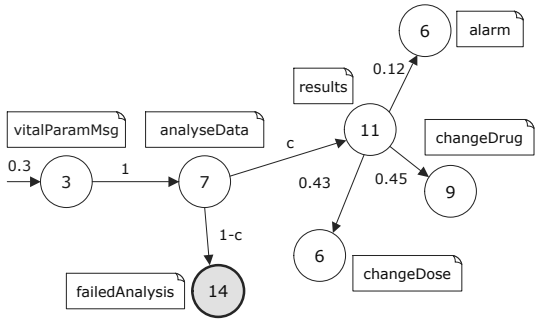

Figure 12: Excerpt of DTMC model for Tele-assistance system (based on [13])

The *Analyzer* component employs the parameterised operational model to identify the service configurations that satisfy the quality of service requirements. To that end, the analyser employs a *model checker*. The model checker requires that the stakeholder requirements are translated from a format in high-

level natural language to a formal expression in the language supported by the model checker (*QoS requirements*). For example, for a DTMC model as shown in Figure 12, requirements can be expressed in Probabilistic Computation Tree Logic (PCTL). The example requirement given above would translate to: $R_0 : P_{\leq 0.13}[\Diamond \text{“}failedAlarm\text{”}]$. PRISM [33] is a model checker that supports the analysis of DTMC models for goals expressed in PCTL expressions. The analyser automatically carries out the analysis of a range of possible configurations of the service based system by instantiating the parameters of the operational model. The result of the analysis is a ranking of the configurations based on the required QoS requirements.

The *Planner* uses the analysis results to build a plan for adapting the configuration of the service-based system. The plan consists of adaptation actions that can be mapping one (or multiple) concrete services with suitable quality properties to an abstract service. Finally, the *Executor* replaces the concrete workflow used by the workflow engine with the new concrete workflow realising the functionality of the QoSMOS service with the required quality of service.

The focus of quantitative verification at runtime as applied in [13] is on providing guarantees for the adaptation goals (see Figure 1). Guaranteeing that the managing system realises its objectives also requires functional correctness of the adaptation components themselves, i.e., the components that realise the MAPE functions. For example, important properties of a self-healing system may be: does the analysis component correctly identify errors based on the monitored data, or does the execute component execute the actions to repair the managed system in the correct order? Lack of such guarantees may ruin the adaptation capabilities. Such guarantees are typically provided by means of design-time modelling, verification, and implementation of the managing system. ActivFORMS [28] (Active FORmal Models for Self-adaptation) is an alternative approach to provide functional correctness of the managing system that is based on executable formal models. Figure 13 shows the basic architecture of ActivFORMS.

The architecture conforms to the three-layer model of Kramer and Magee [31]. A virtual machine enables direct execution of the verified MAPE loop models to realise adaptation at runtime. The approach relies on formally specified templates that can be used to design and verify executable formal models of MAPE loops [26]. ActivFORMS eliminates the need to generate controller code and provide additional assurances for it. Furthermore, the approach supports on-the-fly changes of the running models using the Goal Management interface, which is crucial to support runtime changes of adaptation goals.

Table 10 summarises the key insights derived from Wave V.

3.6 Wave VI. Control-Based Approaches

Engineering self-adaptive systems is often a complex endeavour. In particular, ensuring compliance with the adaptation goals of systems that operate under uncertainty is challenging. In the sixth wave researchers explore the application of control theory as a principle approach to realise runtime adaptation. Control

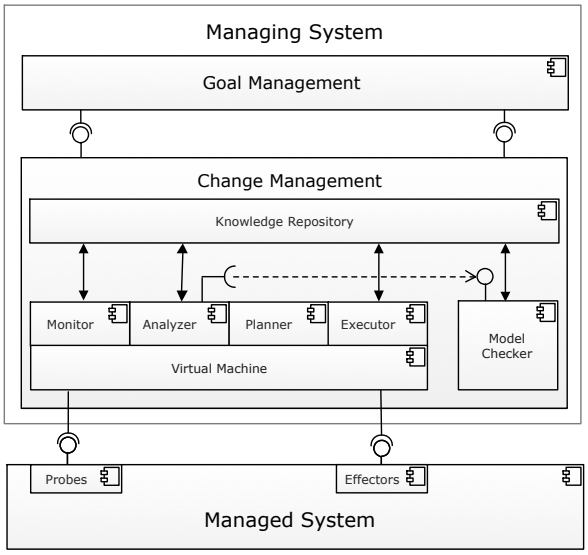

Figure 13: ActivFORMS architecture (based on [28])

Table 10: Key insights of Wave V: Guarantees Under Uncertainties

• Uncertainty is a key driver for self-adaptation.
• Four sources of uncertainties are: uncertainty related to the system itself, the system goals, the execution context, and uncertainty related to human aspects.
• Guarantees for an managing system includes guarantees for the adaptation goals (qualities) and the functional correctness of the adaptation components themselves.
• Quantitative verification at runtime tackles the paradoxical challenge on providing guarantees for the goals of a system that is exposed to continuous uncertainty.
• Executable formal models of feedback loops eliminate the need to generate controller code and provide assurances for it, and support on-the-fly changes of the deployed models, which is crucial for changing adaptation goals during operation.

theory is a mathematically-founded discipline that provides techniques and tools to design and formally analyse systems. Pioneering work on the application of control theory to computing systems is documented in [20, 27]. Figure 14 shows a typical control-based feedback loop.

A control-based computing system consists of two parts: a target system (or plant) that is subject to adaptation and a controller that implements a particular control algorithm or strategy to adapt the computing system. The setpoint represents a stakeholder requirement expressed as a value to be achieved by the system. The target system produces an output that serves as a source of feedback for the controller. The controller adapts the target system by applying a control signal that is based on the difference between the previous system

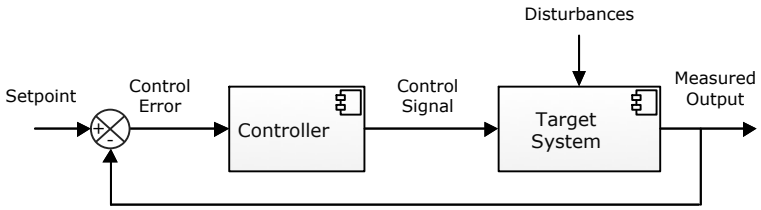

Figure 14: A typical control-based feedback loop

output and the setpoint. The task of the controller is to ensure that the output of the system corresponds to the setpoint while reducing the effects of uncertainty that appear as disturbances or noise in variables or imperfections in the models of the environment used to design the controller.

Different types of controllers exist that can be applied for self-adaptation; the most commonly used type in practice (in general) is the Proportional-Integral-Derivative (PID) controller. Particularly interesting for controlling computing systems is adaptive control that adds an additional control loop that allows adjusting the controller itself, typically to cope with slowly occurring changes of the controlled system [10]. For example, the main feedback loop, which controls a web server farm, reacts rapidly to bursts of Internet load to manage quality of service. A second slow-reacting feedback loop may adjust the controller algorithm to accommodate or take advantage of anomalies emerging over time.

Besides the specific structure of the feedback loop, a key feature of control-based adaptation is the way the target system is modeled, e.g., with difference equations (discrete time) or differential equations (continuous time). Such models allow to mathematically analyse and verify a number of key properties of computing systems. These properties are illustrated in Figure 15.

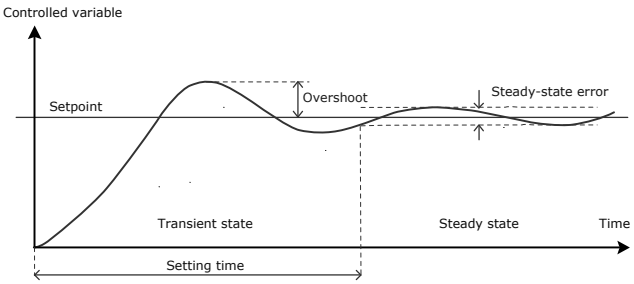

Figure 15: Properties of control-based adaptation

Overshoot is the maximum value by which the system output surpasses the setpoint during the transient phase. *Settling time* is the time required to converge the controlled variable to the setpoint. The amplitude of oscillations of the system output around the setpoint during steady state is called the *steady-*

state error. In addition, *stability* refers to the ability of the system to converge to the setpoint, while *robustness* refers to the amount of disturbance the system can withstand while remaining in a stable state. These control properties can be mapped to software qualities. For example, overshoot or settling time may influence the performance or availability of the application.

Historically, the application of control to computing systems has primarily targeted the adaptation of lower level elements of computing systems, such as the number of CPU cores, network bandwidth, and the number of virtual machines [39]. The sixth wave manifested itself through an increasing focus on the application of control theory to design self-adaptive *software* systems. A prominent example is the Push-Button Methodology (PBM) [23]. PBM works in two phases as illustrated in Figure 16.

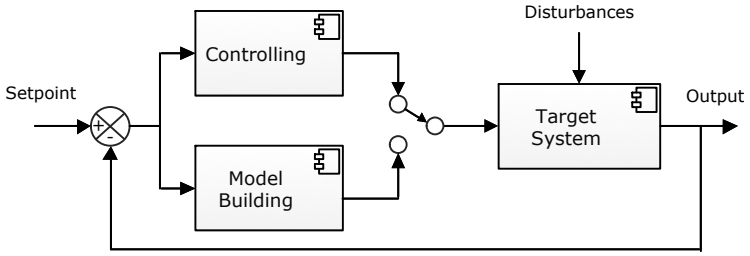

Figure 16: Two phases of PBM (based on [23])

In the *model building phase* a linear model of the software is automatically constructed. The model is identified by running on-the-fly experiments on the software. In particular, the system tests a set of sampled values of the control variable and measures the effects on specified non-functional requirement. The result is a mapping of variable settings to measured feedback. For example, model building measures response time for different number of servers of a Web-based system. In the *controller synthesis phase* a PI-controller uses the synthesised model to adapt the software automatically. For example, in the Web-based system, the controller selects the number of servers that need to be allocated to the service.

To deal with possible errors of the model, the model parameters are updated at runtime according to the system behaviour. For example, if one of the servers in the Web-based system starts to slow down the system response due to overheating, an additional server will be allocated. In case of radical changes, such a failure of a number of servers, a rebuilding of the model is triggered.

A major benefit of a control-theoretic approach such as PBM is that it can provide formal guarantees for system stability, absence of overshoot, settling time, and robustness. The two latter guarantees depend on the so called controller pole (a parameter of the controller that can be set by the designer). Higher pole values improve robustness but lead to higher settling times, while smaller pole values reduce robustness but improve settling time. In other words,

the pole allows to trade-off system responsiveness to change with the ability to withstand disturbances of high amplitude.

PBM is a foundational approach that realises self-adaptation based on principles of control theory. However, basic PBM only works for a single setpoint goal. Examples of follow-up research that can deal with multiple requirements are AMOCS [24] and SimCA [43].

Table 11 summarises the key insights derived from Wave VI.

Table 11: Key insights of Wave V: Control-Based Approaches

• Control theory offers a mathematical foundation to design and formally analyse self-adaptive systems.
• Adaptive controllers that are able to adjust the controller strategy at runtime are particularly interesting to control computing systems.
• Control theory allows providing analytical guarantees for stability of self-adaptive systems, absence of overshoot, settling time, and robustness.
• Linear models combined with online updating mechanisms have demonstrated to be very useful for control-based self-adaptive systems.

4 Future Challenges

Now, we peak into the future of the field and propose a number of research challenges for the next five to ten years to come. But before zooming into these challenges, we first analyse how the field has matured over time.

4.1 Analysis of the Maturity of the Field

According to a study of Redwine and Riddle [41] it typical takes 15 to 20 years for a technology to mature and get widely used. Six common phases can be distinguished as shown in Figure 17. In the first phase, *basic research*, the basic ideas and principles of the technology are developed. Research in the ICAC community[6] has made significant contributions to the development of the basic ideas and principles of self-adaptation. Particularly relevant in this development were also the two editions of the Workshop on Self-Healing Systems.[7] In the second phase, *concept formulation*, a community is formed around a set of compatible concepts ideas and solutions are formulated on specific subproblems. The SEAMS symposium[8] and in particular, the series of Dagstuhl seminars[9] on engineering self-adaptive systems have significantly contributed to the maturation in this phase. In the third phase, *development and extension*, the concepts and principles are further developed and the technology is applied to various

[6] http://nsfcac.rutgers.edu/conferences/ac2004/index.html
[7] http://dblp2.uni-trier.de/db/conf/woss/
[8] www.hpi.uni-potsdam.de/giese/public/selfadapt/seams/
[9] www.hpi.uni-potsdam.de/giese/public/selfadapt/dagstuhl-seminars/

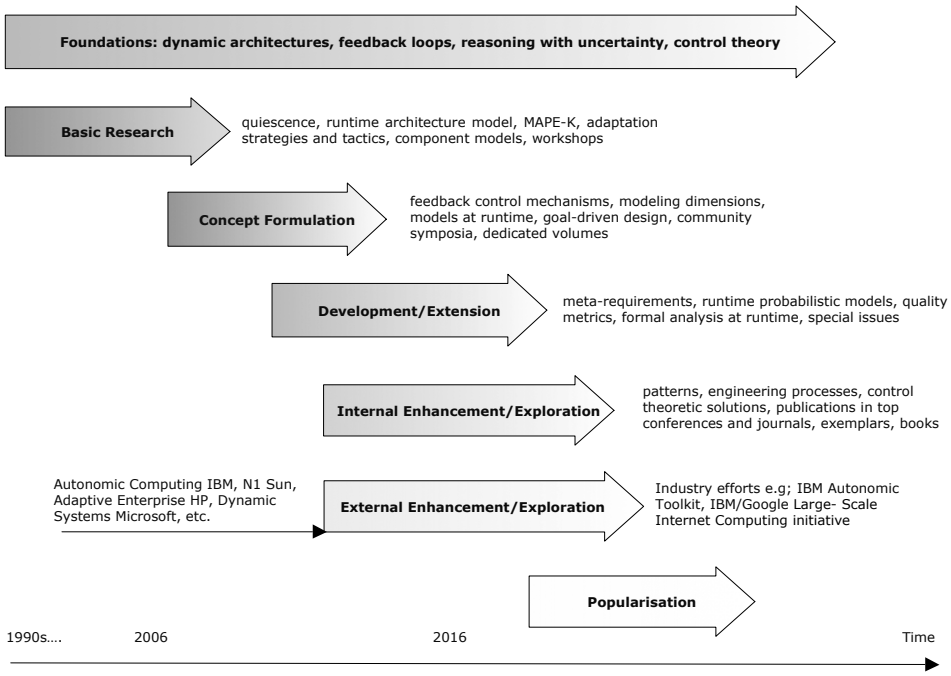

Figure 17: Maturation of the field of self-adaptation. Grey shades indicate the degree the field has reached maturity in that phase (phases based on [41])

applications leading to a generalisation of the approach. Phase four, *internal enhancement and exploration*, the technology is applied to concrete real problems, and training is established. The establishment of exemplars[10] is currently playing an important role to the further maturation of the field. Phase five, *external enhancement and exploration*, involving a broader community to show evidence of value and applicability of the technology, is still in its early stage. Although various prominent ICT companies have invested significantly in the study and application of self-adaptation [9], the effect in practice so far remains relatively low [47]. Finally, the last phase, *popularisation*, where production-quality technology is developed and commercialised has yet to start for self-adaptation. In conclusion, after a relatively slow start, research in the field of self-adaptation has taken up significantly from 2006 onwards and is now following the regular path of maturation. It is currently in the phases of internal and external enhancement and exploration. The application of self-adaptation to practical applications will be of critical importance for the field to reach full maturity.

[10] www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/

4.2 Challenges

After the brief maturity analysis of the field, we look now at challenges that may be worth focusing at in the years to come.

Predicting the future is obviously a difficult and risky task. The community has produced several roadmap papers in the past years, in particular [14] and [18]. These roadmap papers provide a wealth of research challenges structured along different aspects of engineering self-adaptive systems. Here we take a different stance and present open research challenges by speculating how the field may evolve in the future based on the six waves the field went through in the past. We start with a number of short term challenges within current waves. Then we look at challenges in a long term that go beyond the current waves.

4.2.1 Challenges Within the Current Waves

Adaptation in decentralised settings. A principle insight of the first wave is that MAPE represents the essential functions of any self-adaptive system. MAPE takes in principle a centralised perspective on realising self-adaptation. When systems are large and complex, a single centralised MAPE loop may not be sufficient for managing all adaptation in a system. A number or researchers have investigated decentralisation of the adaptation functions; recent examples are [51] where the authors describe a set of patterns in which the functions from multiple MAPE loops are coordinated in different ways, and [12] that presents a formal approach where MAPE loops coordinate with one another to provide guarantees for the adaptation decisions they make. A challenge for future research is to study principled solutions to decentralised self-adaptation. Crucial aspects to this challenge are coordination mechanisms and interaction protocols that MAPE loops require to realise different types of adaptation goals.

Deal with changing goals. One of the key insights of the second wave is that the two basic aspects of self-adaptive systems are change management (i.e., manage adaptation) and goal management (manage high-level goals). The focus of research so far has primarily be on change management. Goal management is basically limited to runtime representations of goals that support the decision making of adaptation under uncertainty. A typical example is [6], where goal realisation strategies are associated with decision alternatives and reasoning about partial satisfaction of goals is supported using probabilities. A challenge for future research is to support changing goals at runtime, including removing goals and adding new goals. Changing goals is particularly challenging. First a solution to this challenge requires goal models that provide first class support for change. Current goal modelling approaches (wave four) take into account uncertainty, but these approaches are not particularly open for changing goals dynamically. Second, a solution requires automatic support for synthesising new plans that comply with the changing goals. An example approach in this direction is ActivFORMS that supports on-the-fly updates of goals and the corresponding MAPE functions [28]. However, this approach requires the engineer to design and verify the updated models before they are

deployed. The full power of dealing with changing goals would be a solution that enables the system itself to synthesis and verify new models.

Domain specific modelling languages. Wave three has made clear that models play a central role in the realisation of self-adaptive systems. A number of modelling languages have been proposed that support the design of self-adaptive systems, but often these languages have a specific focus. An example is Stitch, a language for representing repair strategies within the context of architecture-based self-adaptation [16]. However, current research primarily relies on general purpose modelling paradigms. A challenge for future research is to define domain specific modelling languages that provide first-class support for engineering self-adaptive systems effectively. Contrary to traditional systems, where models are primarily design-time artifacts, in self-adaptive systems models are runtime artifacts. Hence, it will be crucial for modelling languages that they seamlessly integrate design time modelling (human-driven) with runtime use of models (machine-driven). An example approach in this direction is EUREMA that supports the explicit design of feedback loops, with runtime execution and adaptation [46].

Deal with complex types of uncertainties. Wave five made clear that handling uncertainty is one of the "raisons d'être" for self-adaptation. The focus of research in self-adaptation so far has primarily be on parametric uncertainties, i.e., the uncertainties related to the values of model elements that are unknown. A typical example is a Markov model where uncertainties are expressed as probabilities of transitions between states (Figure 12 shows an example). A challenge for future research is to support self-adaptation for complex types of uncertainties. One example is structural uncertainties, i.e. uncertainties related to the inability to accurately model real-life phenomena. Structural uncertainties may manifest themselves as model inadequacy, model bias, model discrepancy, etc. To tackle this problem, techniques from other fields may provide a starting point. E.g., in health economics, techniques such as model averaging and discrepancy modelling have been used to deal with structural uncertainties [8].

Empirical evidence for the value of self-adaptation. Self-adaptation is widely considered as one of the key approaches to deal with the challenging problem of uncertainty. However, as pointed out in a survey of a few years ago, the validation of research contributions is often limited to simple example applications [47]. An important challenge that crosscuts the different waves will be to develop robust approaches and demonstrate their applicability and value in practice. Essential to that will be empirical evidence based on rigorous methods, in particular controlled experiments and case studies. Initially, such studies can be set up with advanced master students (one of the few examples is [49]). However, to demonstrate the true value of self-adaptation, it will be essential to involve practitioners in such validation efforts.

Align with emerging technologies. A variety of new technologies are emerging that will have a deep impact on the field self-adaptation. Among these are 5G, Internet of Things, Cyber Physical Systems, and Big Data. On the one

hand, these technologies can serve as enablers for progress in self-adaptation; e.g., 5G has the promise of offering extremely low latency. On the other hand, they can serve as new consumers of self-adaptation, e.g., self-adaptation as a support for auto-configuration in large-scale Internet of Things applications. A challenge for future research is to align self-adaptation with emerging technologies. Such an alignment will be crucial to demonstrate practical value for future applications. An initial effort in this direction is [38] where the authors explore the use of runtime variability in feature models to address the problem of dynamic changes in (families of) sensor networks. [11] outlines an interesting set of challenges for self-adaption in the domain of Cyber Physical Systems.

4.2.2 Challenges Beyond the Current Waves

To conclude, we speculate on two challenges in the long term that may trigger new waves of research in the field of self-adaptation.

Dealing with unanticipated change. Software is a product of human efforts. Ultimately, a computing machine will only be able to execute what humans have designed for and programmed. Nevertheless, recent advances have demonstrated that machines equipped with software can be incredible capable, examples are machines participating in complex strategic games such as chess and self-driving cars. Such examples raise the intriguing question to what extent we can develop software that can handle conditions that were not anticipated at the time when the software was developed. From the point of view of self-adaptation, an interesting research problem is how to deal with unanticipated change. One possible perspective on tackling this problem is to seamlessly integrate adaptation (i.e., the continuous machine-driven process of self-adaptation to deal with known unknowns) with evolution (i.e., the continuous human-driven process of updating the system to deal with unknown unknowns). This idea goes back to the pioneering work of Oreizy et al. on integrating adaptation and evolution [37]. Realising this idea will require to bridge the fields of self-adaptation and software evolution.

Control theory as a scientific foundation for self-adaptation. Although researchers in the field of self-adaptation have established solid principles, such as quiescence, MAPE, meta-requirements, and runtime models, there is currently no comprehensive theory that underpins self-adaptation. An interesting research challenge is to investigate whether control theory can provide such a theoretical foundation for self-adaptation. Control theory comes with a solid mathematical basis and similar to self-adaptation deals with the behaviour of dynamical systems and how their behaviour is modified by feedback. Nevertheless, there are various hurdles that need to be tackled to turn control theory into the foundation of self-adaptation of software systems. One of the hurdles is the principle differences in paradigms. Software engineers have systematic methods for the design, development, implementation, testing and maintenance of software. Engineering based on control theory on the other hand offers another paradigm where mathematical principles play a central role, principles

that may not be easily accessible to typical software engineers. Another more concrete hurdle is the discrepancy between the types of adaptation goals that self-adaptive software systems deal with (i.e. software qualities such as reliability and performance), and the types of goals that controller deal with (i.e., typically setpoint centred). Another hurdle is the discrepancy between the types of guarantees that self-adaptive software systems require (i.e. guarantees on software qualities) and the types of guarantees that controller provide (settling time, overshoot, stability, etc.). These and other hurdles need to be overcome to turn control theory into a scientific foundation for self-adaptation.

5 Conclusions

In a world where computing systems rapidly converge into large open ecosystems, uncertainty is becoming a defacto element of most systems we build today, and it will be a dominating element of any system we will build in the future. The challenges software engineers face to tame uncertainty are huge. Self-adaptation has an enormous potential to tackle many of these challenges. The field has gone a long way and a substantial body of knowledge has been developed over the past two decades. A key challenge is now to build upon establised foundations, consolidate the knowledge, turn results into robust and repeatable solutions, to move the field forward and propagate the technology throughout a broad community of users in practice. Tackling this challenge is not without risk as it requires researchers to leave their zone of comfort and expose the research results to the complexity of practical systems. However, taking this risk will propel research, open new opportunities, and pave the way towards reaching full maturity as a field.

References

[1] J. Andersson, R. De Lemos, S. Malek, and D. Weyns. Modelling Dimensions of Self-adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*. Springer, 2009.

[2] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Reflecting on Self-adaptive Software Systems. In *Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '09. IEEE Computer Society, 2009.

[3] M. Baldauf, S. Dustdar, and F. Rosenberg. A Survey on Context-aware Systems. *International Journal on Ad Hoc and Ubiquitous Computing*, 2(4):263–277, June 2007.

[4] L. Baresi and C. Ghezzi. The Disappearing Boundary Between Development-time and Run-time. In *Workshop on Future of Software Engineering Research*, FoSER '10. ACM, 2010.

[5] L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy Goals for Requirements-Driven Adaptation. In *International Requirements Engineering Conference*, RE '10. IEEE Computer Society, 2010.

[6] N. Bencomo and A. Belaggoun. Supporting decision-making for self-adaptive systems: From goal models to dynamic decision networks. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, REFSQ '13. Springer Berlin Heidelberg, 2013.

[7] G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[8] L. Bojke, K. Claxton, M. Sculpher, and Palmer S. Characterizing Structural Uncertainty in Decision Analytic Models: A Review and Application of Methods. *Value Health*, 12(5):739–749, 2009.

[9] Y. Brun. Improving Impact of Self-adaptation and Self-management Research Through Evaluation Methodology. In *Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10. ACM, 2010.

[10] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Software Engineering for Self-Adaptive Systems. chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, 2009.

[11] T. Bures, D. Weyns, C. Berger, S. Biffl, M. Daun, T. Gabor, D. Garlan, I. Gerostathopoulos, C. Julien, F. Krikava, R. Mordinyi, and N. Pronios. Software Engineering for Smart Cyber-Physical Systems – Towards a Research Agenda. *SIGSOFT Software Engineering Notes*, 40(6):28–32, 2015.

[12] R. Calinescu, S. Gerasimou, and A. Banks. Self-adaptive software with decentralised control loops. In *International Conference on Fundamental Approaches to Software Engineering*, FASE '15. Springer, 2015.

[13] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, 2011.

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

[15] B. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A Goal-Based Modelling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In *International Conference on Model Driven Engineering Languages and Systems*, MODELS '09. Springer-Verlag, 2009.

[16] S. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860–2875, 2012.

[17] IBM Corporation. An Architectural Blueprint for Autonomic Computing. *IBM White Paper*, 2003. http://www-03.ibm.com/autonomic/pdfs/AC Blueprint White Paper V7.pdf (last accessed: 1/2017).

[18] R. de Lemos, H. Giese, H. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. Smith, J. Sousa, L. Tahvildari, K. Wong, and J. Wuttke. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*. Springer Heidelberg Berlin, Lecture Notes in Computer Science vol. 7475, 2013.

[19] T. De Wolf and T. Holvoet. Emergence Versus Self-Organisation: Different Concepts but Promising When Combined. In *Engineering Self-Organising Systems: Methodologies and Applications*, pages 1–15. Springer Berlin Heidelberg, 2005.

[20] Pedro C. Diniz and Martin C. Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. In *Conference on Programming Language Design and Implementation*, PLDI '97. ACM, 1997.

[21] G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, G. Sukhatme, and B. Petrus. Architecture-driven Self-adaptation and Self-management in Robotics Systems. In *Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '09. IEEE, 2009.

[22] N. Esfahani and S. Malek. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 214–238. Springer Berlin Heidelberg, 2013.

[23] A. Filieri, H. Hoffmann, and M. Maggio. Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees. In *International Conference on Software Engineering*, ICSE '14. ACM, 2014.

[24] A. Filieri, H. Hoffmann, and M. Maggio. Automated Multi-objective Control for Self-adaptive Software Design. In *Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15. ACM, 2015.

[25] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

[26] D. Gil and D. Weyns. MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems*, 10(3):15:1–15:31, 2015.

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

[28] U. Iftikhar and D. Weyns. ActivFORMS: Active Formal Models for Self-adaptation. In *Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '14. ACM, 2014.

[29] M. Jackson. The Meaning of Requirements. *Annals of Software Engineering*, 3:5–21, 1997.

[30] J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[31] J. Kramer and J. Magee. Self-Managed Systems: An Architectural Challenge. In *Future of Software Engineering*, FOSE '07. IEEE Computer Society, 2007.

[32] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[33] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with prism: A hybrid approach. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02. Springer Berlin Heidelberg, 2002.

[34] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns. A Classification of Current Architecture-based Approaches Tackling Uncertainty in Self-Adaptive Systems with Multiple Requirements. In *Managing Trade-offs in Adaptable Software Architectures*. Elsevier, 2016.

[35] B. Morin, O. Barais, J.M. Jezequel, F. Fleurey, and A. Solberg. Models at Runtime to Support Dynamic Adaptation. *IEEE Computer*, 42(10):44–51, 2009.

[36] P. Naur and B. Randell. Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. *Brussels, Scientific Affairs Division, NATO*, 1968.

[37] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based Runtime Software Evolution. In *International Conference on Software Engineering*, ICSE '98. IEEE Computer Society, 1998.

[38] Ó. Ortiz, A. B. García, R. Capilla, J. Bosch, and M. Hinchey. Runtime Variability for Dynamic Reconfiguration in Wireless Sensor Network Product Lines. In *16th International Software Product Line Conference - Volume 2*. ACM, 2012.

[39] T. Patikirikorala, A. Colman, J. Han, and Liuping W. A Systematic Survey on the Design of Self-adaptive Software Systems using Control Engineering Approaches. In *Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '12, 2012.

[40] D. Perez-Palacin and R. Mirandola. Uncertainties in the Modelling of Self-adaptive Systems: A Taxonomy and an Example of Availability Evaluation. In *International Conference on Performance Engineering*, ICPE '14, 2014.

[41] S. Redwine and W. Riddle. Software Technology Maturation. In *International Conference on Software Engineering*, ICSE '85. IEEE Computer Society Press, 1985.

[42] M. Salehie and L. Tahvildari. Self-adaptive Software: Landscape and Research Challenges. *Transactions on Autonomous and Adaptive Systems*, 4:14:1–14:42, 2009.

[43] S. Shevtsov and D. Weyns. Keep it SIMPLEX: Satisfying Multiple Goals with Guarantees in Control-Based Self-Adaptive Systems. In *International Symposium on the Foundations of Software Engineering*, FSE '16, 2016.

[44] V. Silva Souza, A. Lapouchnian, W.. Robinson, and J. Mylopoulos. Awareness Requirements for Adaptive Systems. In *Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11. ACM, 2011.

[45] V. Silva Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos. Requirements-driven software evolution. *Computer Science - Research and Development*, 28(4):311–329, 2013.

[46] T. Vogel and H. Giese. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems*, 8(4):18:1–18:33, 2014.

[47] D. Weyns and T. Ahmad. *Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review*, pages 249–265. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[48] D. Weyns, N. Bencomo, R. Calinescu, J. Cámara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.M. Jezequel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli. Perpetual Assurances in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems*, volume 9640 of *Lecture Notes in Computer Science*. Springer, 2016.

[49] D. Weyns, U. Iftikhar, and J. Söderlund. Do External Feedback Loops Improve the Design of Self-Adaptive Systems? A Controlled Experiment. In *International Symposium on Software Engineering of Self-Managing and Adaptive Systems*, SEAMS '13, 2013.

[50] D. Weyns, S. Malek, and J. Andersson. FORMS: Unifying Reference Model for Formal Specification of Distributed Self-adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):8:1–8:61, 2012.

[51] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Göschka. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 76–107. Springer, 2013.

[52] J. Whittle, P. Sawyer, N. Bencomo, B. Cheng, and J.M. Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *IEEE International Requirements Engineering Conference*, RE '09. IEEE Computer Society, 2009.