Collective Autonomic Systems: Towards Engineering Principles and Their Foundations

Lenz Belzner, Matthias Hölzl, Nora Koch, and Martin Wirsing(✉)

Ludwig-Maximilians-Universität München, Munich, Germany
wirsing@lmu.de

Abstract. Collective autonomic systems (CAS) are adaptive, open-ended, highly parallel, interactive and distributed software systems. They consist of many collaborative entities that manage their own knowledge and processes. CAS present many engineering challenges, such as awareness of the environmental situation, performing suitable and adequate adaptations in response to environmental changes, or preserving adaptations over system updates and modifications. Recent research has proposed initial solutions to some of these challenges, but many of the difficult questions remain unanswered and will open up a rich field of future research.

In an attempt to initiate a discussion about the structure of this emerging research area, we present eight engineering principles that we consider promising candidates for relevant future research, and shortly address their possible foundations. Our discussion is based on a development life cycle (EDLC) for autonomic systems. Going beyond the traditional iterative development process, the EDLC proposes three control loops for system design, runtime adaptation, as well as feedback between design- and runtime. Some of our principles concern the whole development process, while others focus on a particular control loop.

Keywords: Autonomic systems · Awareness · Adaptation · System development life cycle · Control loops · Engineering principles

1 Introduction

Software increasingly models, controls and monitors massively distributed dynamic systems. Systems often operate in highly variable, even unpredictable, open-ended environments. They are based on dynamically forming *ensembles*: sets of parallel, interactive and distributed components able to form groups dynamically and on-demand while pursuing specific goals in changing environments. Each component manages its knowledge and processes (see Fig. 1). Changing requirements, technologies or environmental conditions motivate ensembles that can autonomously adapt without requiring redeployment or interruption of system operation. We call ensembles with these abilities *Collective Autonomic Systems (CAS)*.

This work has been sponsored by the EU project ASCENS IP 257414 (FP7).

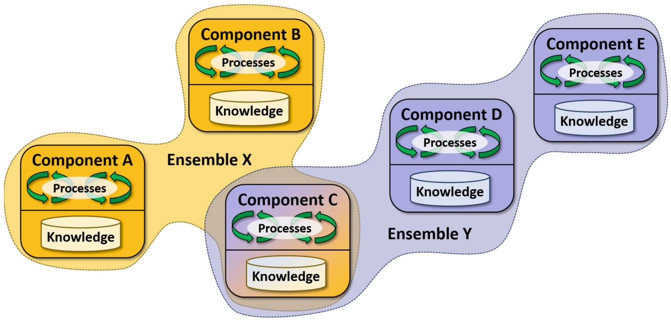

Fig. 1. Two overlapping ensembles [44]

An early vision of autonomic computing was published by Kephart and Chess in 2003 [42]. This work focuses on the self-* properties of autonomic systems, such as self-management, self-optimization and self-protection. The authors discuss a set of scientific challenges, such as learning and optimization theory, and automated statistical modeling.

In 2009 a Dagstuhl seminar group [16] presented the state-of-the-art and research challenges for engineering self-adaptive software systems. Specifically, the roadmap focus on development methods, techniques, and tools that are required to support the systematic development of complex software systems with dynamic self-adaptive behaviour. An updated version of the roadmap was published in 2011 [2].

Recent research has proposed initial solutions to some of these challenges from the engineering point of view, e.g. [66]. There are, however, still many open questions like the interplay of static and dynamic knowledge and the building of societies of systems that require well-founded research in the near future.

In this work we present eight engineering principles that we consider promising candidates for relevant future research, and shortly address their possible foundations. These principles are:

P1 Use probabilistic goal- and utility-based techniques
P2 Characterize the adaptation space and design the awareness mechanism
P3 Exploit interaction
P4 Perform reasoning, learning and planning
P5 Consider the interplay of static and dynamic knowledge
P6 Enable evolutionary feedback
P7 Perform simulation and analysis throughout the system life cycle
P8 Consider societies of systems

We base our discussion on a development life cycle for autonomic systems (EDLC) [32]. In addition to traditional iterative development, the EDLC proposes a control loop for runtime adaptation, as well as feedback between design- and runtime. Some of our principles focus on a particular control loop while others concern the whole system life cycle.

The paper is organized as follows: In Sect. 2 we present a running example and in Sect. 3 we provide an overview of the EDLC. In Sects. 4 to 7 we discuss the proposed engineering principles. Section 8 concludes.

2 The Robot Rescue Scenario

Collective autonomic systems have many potential application areas stressing different aspects of collective adaptation and autonomicity. One domain that places particularly high demands on systems to adapt autonomously to changes in their environment, to continuously adjust task priorities, and even to fulfill requirements that were unknown when the system was deployed is the area of disaster relief and rescue operations.

We will illustrate the principles we propose with the following disaster relief example which is presented in more detail in [30]: An industrial complex has been damaged; workers have been trapped in several buildings and need to be rescued and spills of various chemicals need to be contained and cleaned up. It is expected that some damaged parts of the complex will collapse while the rescue operation is in progress; the rescue team has to take the effects of additional deterioration of the environment into account. In particular, the rescuers should avoid actions that damage the environment in a way that might impair the success of the rescue mission and, if this is beneficial for the progress of the rescue mission, stabilize parts of the environment that are in danger of collapsing.

Having a swarm of autonomous robots capable of performing these kinds of rescue missions would allow humans to stay clear of the dangerous parts of the environment. In some cases it is possible to have remotely controlled robots perform part of the work, but this is often not feasible since buildings, chemicals or radiation may interfere with both wired and wireless remote controls. Therefore, robots that can perform autonomously would be the most suitable solution.

This scenario illustrates many of the complexities of building collective autonomic systems: The robot rescue swarm is a hybrid system; control has to be distributed between the individual agents, it is often unclear which tasks the ensemble of agents should perform and what actions individual agents should take to further progress of the overall system. The environment is continuous, stochastic, only partially observable, and highly dynamic. The presence of multiple adaptive agents complicates many issues, since the effects of actions depend on the adaptations of other agents in addition to the stochastic nature of the environment, and since multiple agents may compete for the same resources, even if they try to cooperate on an overall goal. Since agents may be damaged in the cause of the rescue operation, the system also has to deal with “bad agents”.

3 The Development Life Cycle of Collective Autonomic Systems

The development of collective autonomic systems goes beyond addressing the classical phases of the software development life cycle like requirements elicitation, modeling, implementation and deployment. Engineering these complex systems has also to tackle aspects like awareness and self-adaptation, which have to be considered from the beginning of the development process. This has already been recognized by several authors, for example in the MAPE-K architecture [20] or the life cycles proposed by Inverardi and Mori [36] or Brun et al. [11].

Influenced by these approaches, in previous work we proposed the ensemble development life cycle (EDLC) for autonomic systems, which is based on control loops for the design, runtime and evolution [32]. It can graphically be represented as shown in Fig. 2. The left cycle represents the *design* loop and the right one represents the *runtime* loop. Both loops are connected by a third *evolutionary* loop, consisting of system *deployment* to runtime and *feedback* from runtime into system design.

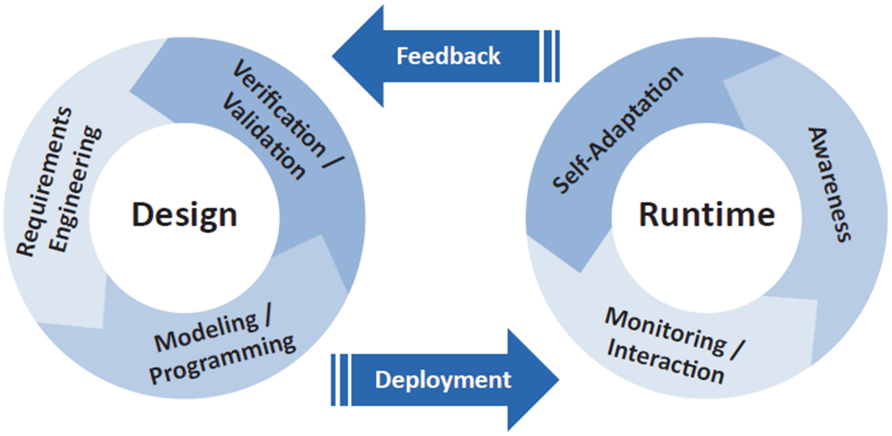

Fig. 2. Ensemble Development Life Cycle (EDLC)

During the design process, work at multiple levels of abstractions and details has to be undertaken simultaneously, providing the basis for the runtime application, which has to be able to monitor the own behaviour and the behaviour of the environment, to reason on these observations, to trigger the learning process, and to adapt to the changes according to the results of the carried out reasoning. System design benefits from the runtime process, too, as feedback can be used to reassess the (adaptation) requirements and the corresponding models.

In summary, we have identified three control loops and propose a software development life cycle for collective autonomic systems based on these three types of control loops: design, runtime and evolution.

Design. The offline phases comprise *requirements engineering*, *modeling and programming*, and *verification and validation*. Models are usually built on top of the elicited requirements, mainly following an iterative process, in which validation and verification in early phases of the development are highly recommended, in order to mitigate the impact of design errors. A relevant issue is then the use of modeling and implementation techniques for adaptive and awareness features. Our aim is to focus on these distinguishing characteristics of collective autonomic systems along the whole development cycle.

We emphasize the relevance of mathematically founded approaches to validate and verify the properties of the CAS and predict the behaviour of such complex software. This closes the cycle, providing feedback for checking the requirements identified so far or improving the model or code.

Runtime. The online phases comprise *monitoring and interaction*, *awareness* and *self-adaptation*. They consist of observing the running system and the environment, reasoning on such observations, and using the results of the analysis for adapting the system and computing data that can be used for feedback in the offline activities.

Evolution. Transitions between online and offline phases can be performed as often as needed throughout the system's evolution feedback control loop, i.e. data acquired during the runtime cycle are fed back (*feedback*) to the design cycle to provide information for system redesign, verification and redeployment (*deployment*).

EDLC and Engineering Principles. We outline the relation of our engineering principles for CAS and the EDLC.

– Design loop: Principles P1 and P2 (use of goal- and utility-based techniques, and characterization and design of adaptation and awareness mechanisms) are mainly applicable in the design loop. They are discussed in Sect. 4.
– Runtime loop: Principles P3 and P4 (the interaction between agents, the system and the environment, as well as reasoning, learning, and planning) are typical for the runtime loop. They are discussed in Sect. 5.
– Evolution loop: Principles P5 and P6 (interplay of static and dynamic knowledge, and the evolutionary feedback) support the evolution loop. They are discussed in Sect. 6.
– All loops: Principles P7 and P8 (simulation and analysis throughout the system life cycle, and consideration of societies of systems) are applicable to all phases of the EDLC. They are discussed in Sect. 7.

The Engineering Principles in the Running Example. We briefly illustrate here how the principles apply to our running example.

We start in the design loop with the specification of requirements such as localization and transport of victims using a goal-oriented approach [1,43]: Goals are identified and subsequently refined. In the robot rescue scenario, initial goals might state the number of victims that the robot swarm should be able to rescue depending on properties of the high-level location graph; later this goal might be detailed in terms of the detailed topography of the underlying map. It is often not possible to guarantee that the system will reach its goals, therefore the designers have to resort to probabilistic characterization. This leads to principle P1: *Use probabilistic goal- and utility-based techniques.*

During the requirements engineering phase, we also need to decide in which environments the system can operate, and what kind of performance we want to achieve in different environments. For example, we need be clear whether the robot swarm is tailored to one particular site or whether it should be able to operate in any site not exceeding a certain size; similarly we need to detail with which kinds of obstacles, damage or environmental danger the swarm can deal, whether it can operate in the presence of other robot rescue swarms or human rescue personnel, etc. In more technical terms, we need to define the adaptation space and the adaptation requirements, as expressed in principle P2: *Characterize the adaptation space and design the awareness mechanism.*

Since in the rescue scenario a large number of robots may act simultaneously, direct communication is often not a feasible strategy. Instead the robots could leave marks in the environment as sources of information for other agents, resulting in stigmergic communication. This interaction with the environment motivates principle P3: *Exploit interaction.*

In the dynamically changing environment in which the robots operate, it is difficult to specify beforehand the precise behavior of the robots. For example, victims will act on their own, continuously creating new situations for robots to cope with. This yields principle P4: *Perform reasoning, learning and planning.*

Nevertheless, complex tasks require modeling before deployment, acquiring runtime information and feeding back this information for further offline improvement of the model. In fact, runtime machine learning can be very useful, but its effectiveness is highly dependent on the quality of the model. However, the quality of the model can only be assessed and improved with the use of runtime data [30]. This is expressed in principle P5: *Consider the interplay of static and dynamic knowledge.*

These models and behaviors learned while operating should be transfered back to the design loop to be used in future iterations. This mutual influence of design and runtime is captured in principle P6: *Enable evolutionary feedback.*

The principles motivated so far yield systems with highly emergent behavior. Thus we propose principle P7: *Perform simulation and analysis throughout the system life cycle.* This supports developers in better understanding the system's dynamics and error conditions. Simulation data can also be used to train adaptive components of the system.

The open-ended environment and possible interaction with other systems leads to principle P8: *Consider societies of systems.* For example, rescue teams of multiple organizations may be deployed to the same disaster site without previous knowledge of each other.

4 Design Loop

Two principles address design-time concerns that support runtime adaptation and evolution: probabilistic goal- and utility-based techniques (P1), and the characterization of the adaptation space and the design of the awareness mechanism (P2).

4.1 Use Probabilistic Goal- and Utility-Based Techniques

Motivation. Collective autonomic systems are typically built to satisfy complex needs that cannot easily be addressed by more static, monolithic systems. Their requirements consist of hard constraints that must not be violated by the system and soft constraints that describe behaviors that should be optimized.

Hard constraints can be expressed as formulas in an appropriate logic that describe properties of the system that should be maintained, avoided or reached (c.f. goal-directed requirements acquisition [22]). A maintain constraint for a robot in the rescue swarm might be to never run out of battery power; an avoid constraint to never injure a rescue worker. Soft constraints are commonly expressed as functions that the system should optimize (see e.g. [23, p. 365]), for example the number of victims rescued by a robot. Most of the time, the soft constraints in CAS describe multicriteria optimization problems. In our scenario, rescuing as many victims as quickly as possible and avoiding damage to itself are competing optimization objectives for individual robots.

Having multiple competing objectives is typical for any requirements specification, since trade-offs between, e.g., possible features, system performance and size, and development time are common for most development tasks. But for traditional systems these trade-offs can be resolved during design time and, while the choices made to resolve them shape the resulting system, they don't have to be considered while the system is operating.

For CAS it is important to be more explicit about the competing requirements for several reasons, such as: 1. A flexible system requires the possibility to weight different criteria according to situation and/or current requirements. This weighting is enabled by explicit distinction of the different optimization criteria. 2. Goals and activities of different agents are intertwined and may lead to emerging phenomena that cannot be derived from properties of the individual agents but only from the system as a whole. Consideration of these kinds of effects by the agents themselves will become increasingly important as systems become more autonomic and adaptive; doing so without an explicit representation of the desired system properties is unlikely to yield favorable results.

We are typically concerned with the behavior of the whole system and not just the software controlling the system, hence most constraints have to take into account physical properties of system components and failure probabilites. Purely goal-based specification can only establish very weak properties. A more likely specification for a CAS would define a required quality of service, e.g., the swarm has to rescue 80 % of the victims with a probability of 95 %.

Therefore, goal and utility-based techniques that explicitly represent the constraints and optimization choices a system faces are more important for CAS than for traditional systems. Given these observations, the first principle we propose is:

Principle P1. Requirements specifications for CAS should be expressed in terms of probabilistic goals and utility functions.

Foundations. Goal-based approaches to requirements analysis, such as KAOS [46], have been widely used in traditional software engineering, and more recently probabilistic variants of these approaches have been proposed [14]. Some goal-based approaches that are concerned with modeling CAS, such as GEM [34], allow modelers to directly express optimization goals in terms of expected utilities.

Most techniques for utility-based modeling and analysis belong to the area of operations research [28]. In particular, expected utility theory [9] and the more general prospect theory [4,40] form the theoretical basis for basing system analysis and design on utility functions. Multicriteria optimization is extensively discussed in [41]. Optimization of probabilistic decisions is often treated in the special context of Markov Decision Processes (MDPs) [56]; stochastic games [49] generalize MDPs and repeated games and are therefore often a more appropriate setting for probabilistic decision problems with multiple agents. Solution techniques for MDPs (and generalizations of MDPs) based on reinforcement learning will be discussed in Sect. 5.2.

Using goal- and utility-based techniques raises the question 1. how system-level goals or utility functions can be decomposed into goals for individual agents, 2. how high-level individual goals or utility functions can be refined into lower-level goals/utilities and eventually individual tasks, and 3. how run-time decision making itself is distributed among the agents in a system. In particular for goal-based techniques there exist a number of multi-agent oriented programming languages and agent-oriented software engineering techniques that address these questions, but most of them are tailored towards deterministic problems and don not address the probabilistic case. The collection [65] contains several relevant summary articles.

4.2 Characterize and Design Adaptation and Awareness

Motivation. The reason for developing Collective Autonomic Systems is often that we need systems that can adjust to many situations, that are resilient to partial failures, and that can easily be changed and enhanced while they are operating. Following [34], we call the range of environments in which a system can operate together with the goals that the system should be able to satisfy in each environment the *adaptation space.* It is important to note that typically the goals and utility functions of a system depend on features of the environment in which it is operating: The more difficult the environment is for the system, the lower the expected utility we can expect it to achieve.

As in [33], we call the parts of the system that are responsible for maintaining information about the environment in which it is operating its *awareness mechanism.* To ensure that the system achieves its desired performance, the awareness mechanism has to provide enough information in each of the possible environments in the adaptation space so that the system can satisfy the goals for that environment.

We often discover new requirements and environmental conditions that influence the capability of a CAS to satisfy its goals in the runtime cycle. When this happens, we need to be able to determine whether the system can still successfully operate under the newly discovered conditions, and, if this is not the case, which changes to the system can restore its capabilities most economically. This is easiest to achieve when the adaptation space is explicitly specified and the awareness mechanism is designed to recognize the different types of environments and adjust the system's behavior accordingly:

Principle P2. Characterize the adaptation space and design the system's awareness mechanism.

Foundations. The notions of adaptation space and awareness mechanism are introduced in [34] and [33], respectively. The connection between the so-called "black-box" view of adaptation that is expressed via adaptation spaces and the "white-box" view of adaptation that is concerned with mechanisms to achieve adaptation is explored in [12].

Bruni et al. [13] presented a control-data-based conceptual framework for adaptivity. They provide a formal model for the framework based on a labelled transition system (LTS). In addition, they provide an analysis of adaptivity from the control data point of view in different computational paradigms, such as context-oriented and declarative programming.

Techniques for developing awareness mechanisms are varied and partially domain dependent. For the robot case study, probabilistic state estimation and filtering [62] are particularly relevant. Techniques for reasoning, planning and learning that can be used in a wide variety of awareness mechanisms are described as part of principle P4 in Sect. 5.2.

5 Runtime Loop

Two principles are mainly rooted in the runtime loop: the exploitation of the interaction between agents or the system and its environment (P3) and the use of reasoning, learning and planning techniques at runtime (P4).

5.1 Exploit Interaction

Motivation. Traditionally, the interaction between agents or between a system and its environment is described in terms of interfaces with protocols that may, e.g., be expressed as state machines. In CAS these kinds of interactions still exist, but often agents also have to interact in different ways with the environment or each other. For example, in the rescue scenario victims may not be able to communicate actively with the rescue robots, instead the rescuers may have to analyze their sensor data (i.e., perform a probabilistic state estimation) to detect the presence of victims and how to rescue them. Similarly, it may not be possible for rescue robots to directly communicate with all other rescue workers in their vicinity; they may again have to rely on state estimation to infer their current activities or intentions. Estimating the internal state of other actors is difficult, so we may simplify the task of the observer by *signaling* the intent or activities of agents, i.e., by performing the actions in such a way that they are easy to recognize. In the simplest case this might just consist in turning on a colored light whenever performing a certain kind of task. For example, a rescue robot might display a green light when it is searching for victims, a red light while it is picking up a victim and a blue light when it is transporting a victim to the rescue zone. More complex behavioral clues are possible and widely observed in animals.

An example of exploiting the interactions between system and environment is *stigmergy*, the process of indirect coordination via manipulation of the environment. For example, if a robot locates a victim but has currently no capacity to rescue the victim, it might change the environment (e.g. by scratching a mark on the ground) to make it simpler for other agents to locate the victim.

Principle P3. Exploit interactions between agents or between the system and its environment to improve system reliability and performance.

Foundations. Many of the foundations for exploiting interactions, such as speech-act theory, signaling, mechanism design, auctions, negotiations or arguments have been studied extensively in the literature on multi-agent systems, see [60, 65, 67] for overviews. The theory of signaling has been extensively studied in economics and evolutionary biology [19, 61]. Mechanism design is concerned with the development of mechanisms that cause utility-maximizing agents to behave according to a desired utility function [8, 35].

Many bio-inspired or swarm-based approaches to computation exploit interactions to enable groups of agents to perform tasks that are beyond the capabilities of the individual member of the group [7, 68]. Stigmergy can frequently be observed in natural systems [18] and it is an important feature of ant algorithms [25].

5.2 Perform Reasoning, Learning and Planning

Motivation. One approach to cope with complex dynamics and change is to synthesize system knowledge and/or behavior at runtime. This ensures that the system can react flexibly to situations that actually occur and focus its computational effort on concretely encountered problems. Engineering of CAS will require a deep understanding of the requirements and implications of the algorithms and frameworks involved in this process. In general, three different aspects of system synthesis at runtime are *abstraction*, *learning* and *reasoning*. Abstraction means that a system is able to meaningfully condense or filter perceptions and thus concentrate cognitive effort on relevant portions of low-level data. For example, the visual system of a rescue robot might condense the signal of its video cameras into a short list of relevant objects and their spatial locations.

Based on given abstractions, learning is concerned with the compilation of runtime data to general knowledge about the environment, e.g. in form of causal relationships or probabilistic prediction of dynamics. Learning may be incremental: Already available models may be refined by a learning process accounting for currently available information, like a rescue robot updating its maps based on data gathered while navigating the rescue area.

Reasoning is concerned with the exploitation of available knowledge to generate new knowledge: For example, a formerly unidentified general causal relation is deduced from some already available knowledge. Decision making in general, and planning in particular, are reasoning processes that compile knowledge about valuable system behavior from existing knowledge. Based on this knowledge, concrete system behavior is selected and executed in order to satisfy system goals as much as possible.

Principle P4. Perform reasoning, learning and planning to enable CAS to autonomously cope with change and unexpected events.

Foundations. Efficiency of system synthesis through learning and planning is highly correlating with the representation (i.e. the abstraction) of the problem at hand. Recently, techniques for representation learning have been successfully applied to learning abstractions from low-level data (digit recognition [29], speech synthesis and recognition [24], etc.), yielding systems that closely reach human performance for particular tasks [45]. Representation learning effectively allows to identify meaningful patterns in raw data. Performing this process iteratively yields different levels of abstraction. Final system learning and reasoning can then be based on learned abstractions. As CAS are potentially designed to be deployed to unknown and/or changing environments, the ability of autonomous abstraction from low-level data is expected to be highly valuable.

In the context of CAS, learning is a tool to account for uncertainty in specification. Domains as the exemplary rescue scenario tend to be highly complex and hard to be modeled completely accurate. Allowing a system to learn models about these domains at runtime enables them to potentially recover from misconceptions or errors in the specification made at design time (e.g. via decision forests [21], Gaussian processes [57] or other machine learning techniques [6]). CAS also may be expected to operate in environments where external, uncontrolled entities interact with them, with potentially conflicting or even adversarial interests. In these cases, learning models of these external agents is essential to ensure system stability and robustness (see also principle P8 in Sect. 7.2).

Single agent decision making can be driven by model-based reinforcement learning with open-loop planning, showing promising results in numerous application areas [10,64]. These techniques particularly suit the highly dynamic and probabilistic environments that CAS are typically designed for. Collective learning and emergent optimization can be achieved with Hierarchical Lenient Multi-Agent Reinforcement Learning [31]. Here, a special emphasis is given to scalability issues in the context of CAS.

While representation learning, supervised machine learning and reasoning each provide powerful tools for their respective problem areas, CAS will require

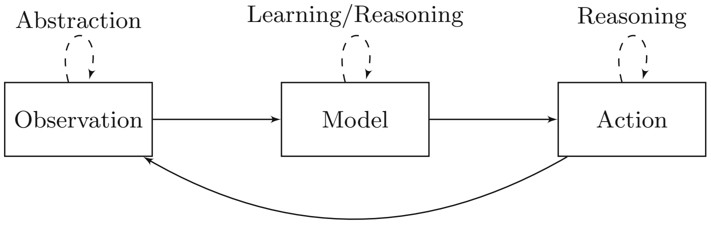

Fig. 3. Interplay of abstraction, learning and reasoning

a deep integration of the aspects involved. Figure 3 illustrates the subtle interplay of these fields. The subfields of system autonomy are clearly dependent on each other: For example, abstraction influences what is learned, but how does a learned model influence possible changes in abstraction? What ways of combining learned models and decision making are there? Answering questions of this kind will be essential for engineering, analyzing and understanding CAS. For recent results combining representation learning and decision making the context of autonomous game players see [17,51]. Further investigation in this direction seems a promising venue for research in the field of CAS.

6 Evolution Loop

Our principles for the evolution loop concern the interplay of static and dynamic knowledge (P5), and a system architecture that enables evolutionary feedback and management of the system's evolution (P6).

6.1 Consider Interplay of Static and Dynamic Knowledge

Motivation. In classical engineering, perceptual abstractions and causal models about system dynamics are usually specified in some particular formalism at design time. We refer to this kind of specification as *static knowledge*. CAS are typically equipped with abilities for gathering and compiling information about their current execution environment. This enables CAS to build perceptual abstractions and causal models at runtime and to optimize their behavior accordingly (see Sect. 5.2). For engineering CAS this raises the question how to deal with and integrate these different sources of knowledge.

For example, a CAS in the rescue domain could learn about the influence of victims' weights on the success of a transportation task and adjust its behavior accordingly. While dynamic optimization of this kind seems a valuable property, the interplay of static specification and runtime optimization is not obvious. For example, specifying a reward function for providing first aid to an injured victim could lead to robots that cause injuries intentionally to collect the specified reward for subsequently providing first aid.

In order to combine static and dynamic knowledge, we have to ensure compatibility of specified and learned representations. CAS have to incorporate specifications when learning abstractions, and we have to extract useful information from knowledge dynamically synthesized by a system at runtime. This information is fed back to the design cycle, and potentially influences further static knowledge design.

Also, continuous comparison of expectation and observation enables systems to detect discrepancies of system design (e.g. knowledge about the environment) and concrete situation. This may trigger recovery mechanisms by either signaling the discrepancy to system operators, or by activating autonomous recovery mechanisms. For example, a designed environmental model may be autonomously replaced by one learned from observations at runtime in case the latter becomes more accurate. Also, this learned model should be fed back into further iterations of system design and analysis.

Principle P5. Understand, analyze and document the interplay of static and dynamic knowledge over the whole lifecycle of CAS.

Foundations. For concepts and algorithms used to compile knowledge from runtime observations, see Sect. 5.2.

To the best of the authors' knowledge, not much research has yet been dedicated to the relation of specification, and representation and model learning. We argue that researching methods for integrating static and dynamic knowledge is a central challenge for successful development of CAS, in particular the integration of results from representation learning with manual specifications.

6.2 Enable Evolutionary Feedback and Manage Evolution

Motivation. Systems designers are used to the notion that they iteratively adapt and improve systems during the development phase, and that they deploy these improvements as part of a larger system. In this process it is common to move from information-rich analysis and design models to a runtime representation in which many of the concerns that are relevant for the system design are "compiled away," either by manually implementing the design models or by performing model transformations into low-level reprentations.

In the EDLC we assume that the developers cannot exactly predict the environment in which the system is operating and the policies that the system should use. Instead the system dynmically improves the actions it takes in response to various situations and, based on the result of these actions, its model of the environment. It is an important problem in the development of CAS to ensure that the notion of autonomous "improvement" of the system agrees with the goals of the stakeholders, and that learned adaptations can be transferred back to the design cycle so that they can influence future design choices and persist over redeployments of and updates to the system. To this end, larger parts of the design knowledge have to be made explicit and made available to the system at runtime: When manually designing a rescue robot it may not be necessary to provide the robot with the information that further injuring victims is undesirable, since this knowledge is already encoded in the robot's program; when allowing the robot to autonomously adapt its strategy it is imperative that this kind of information is present to inform the robot's choices.

The independent adaptation of the CAS leads to an evolutionary process in which parts of the systems can exploit novel capabilities of other agents in the system to autonomously improve their own performance. For example, a rescue robot might figure out a way to scale walls that allows it to cut down the time it needs to get to the victims. Robots without the wall-scaling capability might then stop rescuing other victims and instead transport medicine to the rescue are to avoid medics running out of medicine.

The resulting evolutionary process may either result in a stable system behavior, or it may result in a system where each robot continuously adjusts its strategy to react to the behavioral changes in other robots. This continuous adaptation may greatly enhance the flexibility and performance of the system, but it may also lead to a system that squanders its resources on permanent adaptation without achieving its goals. But even if the system adapts successfully, its developers now face the challenge that their design loop has to keep up with and exploit the evolutionary process in the system, and that new agents they deploy into the system should manage and improve the evolutionary process, not disrupt it needlessly.

Principle P6. Enable evolutionary feedback and manage the evolutionary process.

Foundations. Evolutionary processes can generate surprisingly complex behavior even when the agents involved in the process follow simple, fixed algorithms. Many algorithms in swarm intelligence [25,68] are based on systems consisting of relatively simple agents that collectively achieve complex behaviors. However, to our knowledge no general, systematic method to derive simple agent rules that result in a desired swarm behavior exists. For many scenarios it seems therefore more tractable to have system designs in which individual agents can perform substantial tasks on their own. Techniques for enabling adaptation of these agents to an evolutionary process are described in the principles in Sects. 4 and 5.

The main tool for understanding and analyzing the system dynamics resulting from continuous adaptation of multiple agents is evolutionary game theory [3,63], and in particular the notions of *evolutionary stable strategy* and *co-evolution*. References [60,65] contain discussions of evolutionary game theory in the context of multi-agent systems.

The foundations mentioned for principle P3 are important to manage the evolutionary process as well. In particular mechanism design [8,35] provides tools to design rules that steer the evolutionary process in the desired direction. In many cases it may be necessary to go beyond utility maximization of individual agents and to institute norms or institutions that regulate parts of the system [38,52,55].

7 Engineering Principles for All Loops

In addition to the engineering principles presented in the previous sections for specific EDLC-loops we present two principles encompassing the whole development process: simulation and analysis throughout the system life cycle (P7), and the consideration of societies of systems (P8).

7.1 Perform Simulation and Analysis

Motivation. Complex requirements documents invariably contain inconsistencies, omissions and errors. In traditional system development these problems often manifest themselves during the design or implementation phases and are resolved before the system is deployed. Since in CAS the requirements also serve as a basis for autonomic adaptation of the system, it is more difficult, and also more important, to ensure that the requirements of the system are complete, correct and actually express the desired properties.

Collective adaptive systems potentially have to operate in large environments with probabilistic dynamics and high branching factors. The system environment typically exposes its own dynamics mostly uncontrolled by the system to be designed. Coordination of numerous components increases the complexity of adaptation mechanisms and their assessment through system engineers. Emergent behavior resulting from the interplay of system and environmental dynamics and from interaction of various system participants has to be tuned to fit a system's original requirements.

Statistical analysis of simulation data enables to quantify system properties. For example, in classical reachability analysis a definite result is possible due to the deterministic nature of systems. In CAS, there will usually be a probabilistic dimension to reachability results, as typically only few of the dynamics of the domain are under direct control of the system. On the other hand, classical verification techniques such as model checking could be used at runtime in order to verify and analyze the actually occurring system configuration. This allows verification without considering all possible adaptations of the CAS.

Principle P7. Perform simulation and analysis throughout the system life cycle. In particular, deploy early to a simulated environment so that feedback from the runtime loop can be used in the design cycle as early as possible.

Foundations. Modern simulation tools allow to access system behavior at various levels of detail by being able to simulate physical dynamics, visual data and audio information with accuracy close to reality [50,53]. Simulation is highly efficient, supported by designated hardware architectures (e.g. graphic processors). High-quality simulation data is especially valuable in situations where algorithms are employed that learn abstractions from low-level data at runtime (e.g. autoencoders [24], principal component analysis [37] or sparse coding [47]).

Simulation provides a way to train and assess systems that employ model or behavior synthesis before actually deploying them (i.e. learning and reasoning systems, see Sect. 5.2). For example, a system that uses some sort of representation learning to compile abstractions from low-level data may use simulated sensory data to start to learn abstractions before being deployed.

Simulation is also the core component of statistical approaches to validation and verification of systems, as e.g. statistical model checking [48,59]. Here, a number of simulated runs of the system is performed and analyzed to provide assertions about system performance with a required statistical confidence. To the best of the authors' knowledge, literature does not provide a generic solution of generating explicit feedback for system design from stochastic simulation data effectively. Identifying relevant parts of simulation data and finding ways to use this information, both at design time and at runtime, are crucial challenges for successfully building and operating CAS.

7.2 Consider Societies of Systems

Motivation. Often multiple ensembles exist that we require to cooperate adaptively, resulting in ensembles that team up to build larger ensembles. While it is possible to define coordinative mechanisms in a centralized manner, it is valuable to provide approaches that allow for emergent control of collaboration to ensure adaptivity and robustness. Various forms of coordination and collaboration can be observed in natural *societies and organizations*: Societal mechanisms of control range from highly centralized (e.g. dictatorial) to extremely distributed (e.g. swarms). Organization forms for managing communication and aggregation of information range from flat to highly hierarchical. Building societies of systems also requires to dynamically define to what extend a member of a society (be it an individual or an ensemble) should weight individual vs. global welfare.

In the robotic rescue example, consider different teams of robots operating in the scene. When an ensemble is built to extinguish a fire in a particular area, the group mainly designed to provide first aid to victims would rather hinder in fulfilling the goal. Also, different ways of organization would be needed for fighting a fire (probably very reactive, with high individual decision impact) and transporting a victim to safety in collaboration (rather centralized, with individuals providing information to a coordinator). Nevertheless it may be the case that some robots have to participate in both tasks.

Conflicting individual goals, openness and the need for collaboration require the ability to deal robustly with potentially adversarial or malfunctioning decision makers. For example, an agent could simply fail without having noticed or it could provide misinformation intentionally. It is valuable for CAS to explicitly maintain a model of reliability of collaborators to base own decisions on this information.

Principle P8. Consider societies of systems and enable CAS to cope adaptively and robustly with the complexity of interaction within and between systems.

Foundations. While it is straightforward to implement social mechanisms adhoc, it is necessary to study their characteristics and implications in order to ensure their support for our design goals. Recent approaches take inspiration from the fields of socio-technical systems [27] and institutional theory [58] to provide formal techniques and languages to capture and express the exact meaning of social institutions and mechanisms [38,55].

Exploiting virtual social institutions to enable adaptive formation and disassembly of coordination collectives yields highly open systems. This openness of CAS disables traditional security solutions. Recent surveys provide an overview of new attacks forms and solution approaches in the changing security landscape of open, autonomous multi-agent systems [5,39]. Research and engineering approaches in the field of CAS have to address the changing security challenges resulting from building societies of systems. Trust- and reputation-based computing provides a principled approach for systems to deal with openness and external decision makers [15,26,54].

8 Conclusions

In this work we presented eight engineering principles that we consider to play a fundamental role in the construction and running of collective autonomous systems (CAS) and that we consider promising areas for future research:

P1 Use probabilistic goal- and utility-based techniques
P2 Characterize the adaptation space and design the awareness mechanism
P3 Exploit interaction
P4 Perform reasoning, learning and planning
P5 Consider the interplay of static and dynamic knowledge
P6 Enable evolutionary feedback
P7 Perform simulation and analysis throughout the system life cycle
P8 Consider societies of systems

The principles are closely related to the ensembles development life cycle (EDLC), which distinguishes a design, a runtime and an evolution loop. Some of our principles concern the whole development process while others are focused on a particular loop (design loop: P1, P2; runtime loop: P3, P4; evolution loop: P5, P6; all loops: P7, P8). We have motivated each principle in the context of CAS and discussed its scientific foundations and challenges.

Our list of principles is far from exhaustive. There are other relevant aspects which could be addressed and analyzed in detail. Some of these address technical properties of CAS, such as safety or security. There are also broader implications of development and deployment of CAS to be considered, for example the ethical and legal concerns raised by systems that autonomously act in environments shared with humans and other autonomous systems. We hope that the principles presented in this paper can serve as guidelines for future research in the area of CAS and foster further discussion and ideas in the field.

References

1. Abeywickrama, D., Bicocchi, N., Zambonelli, F.: SOTA: towards a general model for self-adaptive systems. In: 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 48–53, June 2012
2. Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35813-5_1
3. Alexander, J.M.: Evolutionary game theory. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Stanford Center for the Study of Language and Information, Fall 2009 edn. (2009)
4. Barberis, N.C.: Thirty years of prospect theory in economics: a review and assessment. J. Econ. Perspect. **27**(1), 173–196 (2013). http://www.aeaweb.org/articles.php?doi=10.1257/jep.27.1.173
5. Bijani, S., Robertson, D.: A review of attacks and security approaches in open multi-agent systems. Artif. Intell. Rev. **42**(4), 607–636 (2014)
6. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer, Heidelberg (2006)
7. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence-From Natural to Artificial Systems. Studies in the Sciences of Complexity. Oxford University Press, Oxford (1999). http://ukcatalogue.oup.com/product/9780195131598.do
8. Borgers, T., Krahmer, D., Strausz, R.: An Introduction to the Theory of Mechanism Design. Oxford University Press, Oxford (2015)
9. Briggs, R.: Normative theories of rational choice: expected utility. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Stanford Center for the Study of Language and Information, fall 2014 edn. (2014)
10. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S., et al.: A survey of monte carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
11. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02161-9_3
12. Bruni, R., Corradini, A., Gadducci, F., Hölzl, M., Lafuente, A.L., Vandin, A., Wirsing, M.: Reconciling white-box and black-box perspectives on behavioural self-adaptation. In: Wirsing et al. [68]
13. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 240–254. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28872-2_17
14. Cailliau, A., van Lamsweerde, A.: Assessing requirements-related risks through probabilistic goals and obstacles. Requir. Eng. **18**(2), 129–146 (2013). doi:10.1007/s00766-013-0168-5
15. Castelfranchi, C., Tan, Y.H.: Trust and Deception in Virtual Societies. Springer, Heidelberg (2001)
16. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02161-9_1

17. Clark, C., Storkey, A.: Training deep convolutional neural networks to play go. In: Proceedings of the 32nd International Conference on Machine Learning, pp. 1766–1774 (2015)
18. Special Issue of the Journal for Cognitive Systems Research: Stigmergy 3.0: From Ants to Economics. Elsevier, March 2013
19. Connelly, B.L., Certo, S.T., Ireland, R.D., Reutzel, C.R.: Signaling theory: a review and assessment. J. Manag. **37**(1), 39–67 (2011)
20. IBM Corporation: An architectural blueprint for autonomic computing. Technical report, IBM (2005). http://researchr.org/publication/autonomic-architecture-2005
21. Criminisi, A., Shotton, J., Konukoglu, E.: Decision Forests: A Unified Framework for Classification, Regression, Density Estimation Manifold Learning and Semi-supervised Learning. Now, Breda (2012)
22. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Sci. Comput. Program. **20**(1), 3–50 (1993). http://www.sciencedirect.com/science/article/pii/016764239390021G
23. Dechter, R.: Constraint Processing. Morgan Kaufmann, Burlington (2003)
24. Deng, L., Seltzer, M.L., Yu, D., Acero, A., Mohamed, A.r., Hinton, G.E.: Binary coding of speech spectrograms using a deep auto-encoder. In: Interspeech, pp. 1692–1695. Citeseer (2010)
25. Dorigo, M., Bonabeau, E., Theraulaz, G.: Ant algorithms and stigmergy. Future Gener. Comput. Syst. **16**(9), 851–871 (2000). http://dl.acm.org/citation.cfm?id=348599.348601
26. Falcone, R., Castelfranchi, C.: Social trust: a cognitive approach. In: Castelfranchi, C., Tan, Y.-H. (eds.) Trust and Deception in Virtual Societies, pp. 55–90. Springer, Netherlands (2001)
27. Geels, F.W.: From sectoral systems of innovation to socio-technical systems: insights about dynamics and change from sociology and institutional theory. Res. Policy **33**(67), 897–920 (2004). http://www.sciencedirect.com/science/article/pii/S0048733304000496
28. Hillier, F., Lieberman, G.: Introduction to Operations Research. McGraw-Hill Higher Education, New York (2010). https://books.google.de/books?id=NvE5PgAACAAJ
29. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), 1527–1554 (2006)
30. Hölzl, M., Gabor, T.: Continuous collaboration for changing environments. In: Steffen, B. (ed.) Transactions on FoMaC I. LNCS, vol. 9960, pp. 201–224. Springer, Heidelberg (2016). doi:10.1007/978-3-319-46508-1_11
31. Hölzl, M., Gabor, T.: Reasoning and learning for awareness and adaptation. In: Wirsing et al. [68]
32. Hölzl, M., Koch, N., Puviani, M., Wirsing, M., Zambonelli, F.: The ensemble development life cycle and best practises for collective autonomic systems. In: Wirsing et al. [68]
33. Hölzl, M., Wirsing, M.: Issues in engineering self-aware and self-expressive ensembles. In: Pitt, J. (ed.) The Computer After Me: Awareness and Self-awareness in Autonomic Systems. Imperial College Press, London (2014)
34. Hölzl, M., Wirsing, M.: Towards a system model for ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems: Essays Dedicated to Carolyn Talcott on the Occasion of her 70th Birthday. LNCS, vol. 7000, pp. 241–261. Springer, Heidelberg (2011). doi:10.1007/978-3-642-24933-4_12

35. Hurwicz, L., Reiter, S.: Designing Economic Mechanisms. Cambridge University Press, New York (2006). https://books.google.de/books?id=Mvn8chTLeFwC
36. Inverardi, P., Mori, M.: A software lifecycle process to support consistent evolutions. In: Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 239–264. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35813-5_10
37. Jolliffe, I.: Principal Component Analysis. Wiley Online Library, New York (2002)
38. Jones, A., Artikis, A., Pitt, J.: The design of intelligent socio-technical systems. Artif. Intell. Rev. **39**(1), 5–20 (2013). doi:10.1007/s10462-012-9387-2
39. Jung, Y., Kim, M., Masoumzadeh, A., Joshi, J.B.: A survey of security issue in multi-agent systems. Artif. Intell. Rev. **37**(3), 239–260 (2012)
40. Kahneman, D., Tversky, A.: Prospect theory: an analysis of decision under risk. Econometrica **47**, 263–291 (1979)
41. Keeney, R., Raiffa, H.: Decisions with Multiple Objectives: Preferences and Value Tradeoffs. Wiley, New York (1976)
42. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). doi:10.1109/MC.2003.1160055
43. Keznikl, J., Bures, T., Plasil, F., Gerostathopoulos, I., Hnetynka, P., Hoch, N.: Design of ensemble-based component systems by invariant refinement. In: Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering, CBSE 2013, pp. 91–100. ACM, New York (2013)
44. Koch, N.: ASCENS: autonomic service-component ensembles (brochure), February 2015
45. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Proceedings of Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
46. van Lamsweerde, A.: Requirements engineering in the year 00: a research perspective. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), pp. 5–19. ACM (2000)
47. Lee, H., Battle, A., Raina, R., Ng, A.Y.: Efficient sparse coding algorithms. In: Proceedings of Advances in Neural Information Processing Systems, pp. 801–808 (2006)
48. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16612-9_11
49. Mertens, J.F., Neyman, A.: Stochastic games. Int. J. Game Theor. **10**(2), 53–66 (1981). doi:10.1007/BF01769259
50. Millington, I.: Game Physics Engine Development. Morgan Kaufmann Publishers, Amsterdam (2007)
51. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.A.: Playing atari with deep reinforcement learning. CoRR abs/1312.5602 (2013). http://arXiv.org/abs/1312.5602
52. Ostrom, E.: Governing the Commons: The Evolution of Institutions for Collective Action. Political Economy of Institutions and Decisions. Cambridge University Press, New York (1990). https://books.google.de/books?id=4xg6oUobMz4C
53. Pharr, M., Humphreys, G.: Physically Based Rendering: From Theory to Implementation. Morgan Kaufmann, Burlington (2004)
54. Pinyol, I., Sabater-Mir, J.: Computational trust and reputation models for open multi-agent systems: a review. Artif. Intell. Rev. **40**(1), 1–25 (2013)

55. Pitt, J., Busquets, D., Bourazeri, A., Petruzzi, P.: Collective intelligence and algorithmic governance of socio-technical systems. In: Miorandi, D., Maltese, V., Rovatsos, M., Nijholt, A., Stewart, J. (eds.) Social Collective Intelligence. Computational Social Sciences, pp. 31–50. Springer International Publishing, Switzerland (2014). doi:10.1007/978-3-319-08681-1_2
56. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming, 1st edn. Wiley, New York (1994)
57. Rasmussen, C.E.: Gaussian Processes for Machine Learning. MIT Press, Massachusetts (2006)
58. Scott, W.R.: The adolescence of institutional theory. Adm. Sci. Q. **32**(4), 493–511 (1987)
59. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005). doi:10.1007/11513988_26
60. Shoham, Y., Leyton-Brown, K.: Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations. Cambridge University Press, New York (2008)
61. Spence, M.: Signaling in retrospect and the informational structure of markets. Am. Econ. Rev. **92**(3), 434–459 (2002). http://www.aeaweb.org/articles.php?doi=10.1257/00028280260136200
62. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. MIT Press, Massachusetts (2005)
63. Weibull, J.W.: Evolutionary Game Theory. MIT Press, Cambridge (1995)
64. Weinstein, A., Littman, M.L.: Open-loop planning in large-scale stochastic domains. In: desJardins, M., Littman, M.L. (ed.) Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, 14–18 July 2013, Bellevue, Washington, USA. AAAI Press (2013). http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6341
65. Weiss, G. (ed.): Multiagent Systems, 2nd edn. MIT Press, Massachusetts (2013)
66. Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.): Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998. Springer, Heidelberg (2015)
67. Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley, New York (2009). https://books.google.de/books?id=X3ZQ7yeDn2IC
68. Yang, X.-S. (ed.): Recent Advances in Swarm Intelligence and Evolutionary Computation. SCI, vol. 585. Springer, Heidelberg (2015). doi:10.1007/978-3-319-13826-8