Towards Systematically Engineering Autonomous Systems using Reinforcement Learning and Planning

Draft.
Final version in
Pedro López-García, John P. Gallagher, Roberto Giacobazzi (eds.):
Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems 2023: 281-306

Martin Wirsing[1] and Lenz Belzner[2]

[1] Ludwig-Maximilians-Universität München, Munich, Germany `wirsing@lmu.de`
[2] Technische Hochschule Ingolstadt, Ingolstadt, Germany `lenz.belzner@thi.de`

Abstract. Autonomous systems need to be able dynamically adapt to changing requirements and environmental conditions without redeployment and without interruption of the systems functionality. The EU project ASCENS has developed a comprehensive suite of foundational theories and methods for building autonomic systems. In this paper we specialise the EDLC process model of ASCENS to deal with planning and reinforcement learning techniques. We present the "AIDL" life cycle and illustrate it with two case studies: simulation-based online planning and the PSyCo reinforcement learning approach for synthesizing agent policies from hard and soft requirements. Related work and potential avenues for future research are discussed.

Dedicated to Manuel Hermenegildo

1 Introduction

An autonomous system is able to adapt at runtime to uncertain and dynamically changing environments and to new requirements. Autonomous systems can be single autonomous entities or collective ones that consist of several collaborating entities. Classical examples are intelligent agents [67], and autonomic systems [26], more recent are ensembles [64,28], and collective adaptive systems [33].

Reinforcement learning [53] and online planning are methods for automatically computing sequential controllers - so-called policies - of autonomous systems. Given an uncertain probabilistic environment, reinforcement learning is about learning from interactions with the environment. It is an interactive process with the goal to learn a policy that maximises the sum of future rewards. Planning requires a (simulation) model and is a "computational process that takes a model as input and produces or improves a policy for interacting with the modelled environment".

Systematic engineering approaches for intelligent agents and multi-agent systems such as Gaia [68], Tropos [14] support a sequential development process or focus on software architecture such as IBM's MAPE-K architecture [30] for autonomic systems.

The modern industrial agile development approaches MLOps [3] and AIOps [2] aim at machine learning methods for big data applications and IT operations. [22] proposes an engineering process exhibiting the central activities necessary for the successful application of machine learning.

The EU project ASCENS [1,65] has developed a comprehensive suite of foundational theories and methods for building autonomic systems. The ASCENS methods cover system specification and development as well as monitoring and dynamic system adaptation. Also machine-learning approaches have been studied in ASCENS but they were not systematically related to the software development life cycle. In particular, the ASCENS project has proposed the Ensemble Development Life Cycle EDLC [27] for engineering adaptive and autonomous systems. The EDLC is an agile process covering the whole software life cycle including development and runtime phases and provides mechanisms for enabling system changes at runtime.

In this paper we review the EDLC life cycle and specialise it to the construction of autonomous policies using planning and reinforcement learning techniques. We call this life cycle "AIDL" and illustrate it with two existing case studies: simulation-based online planning for autonomously adapting the behaviour of a robot [10] and the PSyCo reinforcement learning approach for synthesizing agent policies from hard and soft requirements [12]. Related work and future directions for research are discussed.

Personal Note. Martin has known Manuel for almost 20 years when they met and contributed to initiatives of the Future Emerging Technologies section of the European Commission, e.g. in 2005 at the "Beyond-the-Horizon" Workshop on Anticipating Future and Emerging Information Society.

In 2007 Manuel invited Martin to become a member of the Scientific Board of IMDEA Software and later in 2011, to be a guest researcher at IMDEA Software for three months. In this way, Martin had the chance in participating in the extra-ordinary raise of IMDEA Software to one of Europe's leading research institutes. Cooperating and discussing with Manuel is a very pleasant experience. He is not only an outstanding scientist and an excellent coordinator of scientific work; he is also a warm-hearted and kind friend and colleague. We are looking forward to many further inspiring exchanges with him.

Outline. In Section 2 we shortly review reinforcement learning, planning, and the EDLC life cycle. In Section 3 we present the AIDL Life Cycle. Section 4 illustrates AIDL by two case studies. Finally, Section 5 presents related work and Section 6 concludes the paper.

2 Preliminaries: Reinforcement Learning, Planning, and the Life Cycle EDLC

2.1 Reinforcement learning and planning

Reinforcement learning and (online-) planning are well-suited methods for computing optimising goals in a probabilistic domain. The standard case is that the domain is given by a Markov decision process (MDP) (for a formal definition see Appendix A) and the goal is to compute a policy which maximises an expected reward.

Model-free and model-based reinforcement learning. Reinforcement learning is an interactive process between an agent and the environment. The goal is to learn a policy for maximising the discounted cumulative return the agent receives over time (for definitions see Appendix A). In each step the agent makes an action and receives an immediate reward. Typically, positive values express good actions, negative values express bad actions.

There is a rich family of reinforcement learning algorithms. For small or middle size state and action spaces classical algorithms such as value iteration, policy iteration, and temporal difference learning are widely used. If the state space is large one can only hope to find approximate solutions and thus uses so-called function approximation methods such as gradient-descent over artificial neural networks (see e.g. [53,54]).

The latter algorithms are model-free in that they do not have any knowledge of the domain and thus start from an arbitrary distribution. Model-based algorithms have access to or learn a (probabilistic) digital twin[3] of the environment which predicts state transitions and rewards. This allows the algorithm to plan its next steps based on a range of possible choices. In many cases this considerably improves the learning efficiency; however, if the model does not faithfully match the reality the algorithm may behave badly in the real environment.

Some modern algorithms combine model-free with model-based learning. E.g. AlphaGo combines a model-free reinforcement learning algorithm with (model-based) Monte Carlo tree search; it was the first program to win the game Go against a human champion [51]. Reinforcement learning algorithms are also combined with evolutionary methods in order to improve stability and quality of the results [20].

[54] gives an overview on classical reinforcement learning algorithms (until 2010). For a taxonomy of reinforcement learning algorithms see [4], an excellent survey on model-based reinforcement learning is given in [40].

Safe learning. In the algorithms above, learning is used for optimizing the system behavior but not for guaranteeing the safety of the system. But in many applications, the system requirements comprise different kinds of goals including achieve goals that optimize behaviours, and maintain goals that restrict the space of feasible solutions.

There are three broad classes for dealing with such situations: shielding [6], safe exploration [24], and reward-shaping methods [47,12]. Shielding ensures at runtime that the chosen action is safe whereas in safe exploration, the learning process is restricted to learn only safe actions. Reward-shaping methods are based on Constrained Markov Decision Processes (CMDP) [7] (see also Appendix A) and try to balance optimization of return and costs incurred by constraint violation. E.g. [12] uses a Lagrangian for transforming the costs of the safety constraints and the rewards into a single optimizing problem. The safety of the solution is ensured by runtime Bayesian model checking.

[3] also called internal model or simulation model in the literature.

Non-stationary environments. In non-stationary environments the probability transition function and/or the reward change over time. Main approaches are transfer learning, and meta-learning (see e.g. [40]).

Transfer learning [55,59] explores the idea that experience gained in learning to perform one task can help improve learning performance in a related, but different, task. Meta-learning [56] is concerned with accumulating experience on the performance of multiple applications of a learning system. Typically, an adaptation space is given by a distribution of environments and a shared common structure that can be exploited for fast learning. For fast online adaptation in dynamic environments, [18] uses an distribution of MDPs whereas [42] meta-trains a global model.

Planning. Planning is a large and longtime established field. In “classical” offline-planning algorithms the policy is constructed for the entire state space before the system is interacting with the environment. This is only feasible for small and mid-size problems. Instead, an online algorithm is computing a near-optimal action for the current state. When interacting with the environment an online-algorithm encounters only a small subset of the entire state space and has to tune its decisions only for a single time step.

The key idea of online planning is to perform planning and execution of an action iteratively at runtime. At each planning step, the agent performs forward search on a digital twin, e.g. by Monte Carlo Tree Search [34,15] (in discrete domain) or by Cross Entropy Open Loop Planning [62] (in continuous state and action spaces). Online planning is suitable for MDPs as well as for partially observable MDPs (POMDPs) (see formal def. in Appendix A). A survey of classical online POMDP methods is given in [49]. For the trade-off between online planning and model-based reinforcement learning see [41].

2.2 The Ensemble Development Life Cycle EDLC

The “Ensemble Development Life Cycle” EDLC [27] is an agile software process model that explicitly deals with autonomous systems, in particular with ensembles and collective adaptive systems. EDLC has been used in the development of several autonomic systems such as swarm robots [66,44], peer-to-peer cloud [38], and e-mobility applications [17,25]. The construction of autonomous systems using EDLC is supported by eight engineering principles [11]. System construction according to EDLC emphasises mathematically well-founded approaches to validate and verify the properties of the collective autonomic system and enable the prediction of the behaviour of such complex software.

The EDLC life cycle is arranged in three cycles (see Fig. 1). In the development cycle Dev (called "design time cycle" in [27]) the classical development phases - requirements engineering, modelling and programming, verification and validation - are iterated; in the operations cycle Ops (called "runtime cycle" in [27]), the entities of the ensemble iterate a "runtime feedback control loop" comprising the monitoring, awareness, and (self-) adaptation mechanisms. 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 providing feedback data that can be used in the development activities for improving the system. The connection between the two cycles is established by a third evolutionary cycle consisting of system deployment or hot update to the operations and providing feedback data from runtime to the development cycle.

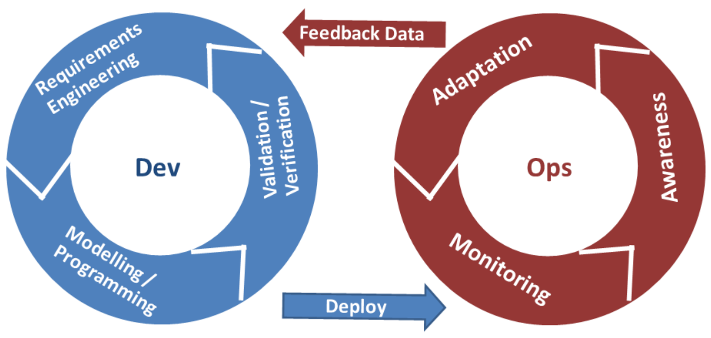

Fig. 1: Ensemble Development Life Cycle EDLC.

Development cycle Dev. The phases of the Dev cycle rely on mathematically well-founded approaches that support the correct construction and the analysis of autonomic systems.

Requirements. EDLC supports two goal-oriented methods - SOTA [5] and ARE [58] - for elicitating and specifying the requirements. In both cases the final requirements specification consists of a model of the domain together with hard goals that the system has to satisfy and soft goals that describe behaviours that should be optimised.

The notion of adaptation domain [66,29] describes the “borders of validity” of an autonomous system. The adaptation domain determines the variety of different environments, goals, and adverse system states the system should be able to tolerate and in which it should be able to continue working “correctly.” In some cases the adaptation space is complemented by so-called “resilience goals” which determine those environments and system states outside the adaptation space the system should be able to recover from and return back into the adaptation space.

Modelling and programming. For this task, the EDLC relies on well-known methods for modelling and implementing adaptation and autonomy. This ranges from a method for stepwise refinement and the development of high-level modelling languages (such as SCEL [19]) to classical adaptation techniques (such as programming using modes and dynamic reconfiguration) as well as AI adaptation techniques (such as swarm algorithms as well as planning, learning, and reasoning).

A main ingredient are pattern catalogues [63,21,45] to help developers to make appropriate design choices for models and implementations. For example, architectural patterns such as “knowledge-equipped component” describe the architecture of a system or a component, and adaptation patterns such as “centralised autonomic manager” are concerned with adaptation mechanisms [27,45].

Validation and verification. Analysis techniques for adaptive and autonomous systems have to cover the “normal” system behaviour as well as essential aspects such as adaptive behaviour and changing environments. This comprises qualitative methods ensuring that the system behaves without any flaw, and quantitative analyses that target non-functional properties and evaluate expected performances according to predefined metrics.

Qualitative methods range from reviews and testing to the automated verification of invariants and security properties. Quantitative methods are well-suited for performance analysis and studying the behaviour of a system in different environments and under changing requirements (for a comprehensive collection of papers see [13]). Main techniques are statistical modelchecking (see e.g. [36]), simulation tools (see e.g. [37]) and the analysis of Markov chains using differential equations (see e.g. [66,57]).

Operations Cycle Ops In the operations cycle, the entities of the autonomic system iterate a "runtime feedback control loop" consisting of monitoring, awareness, and self-adaptation activities.

Montoring The task of monitoring is to collect data at runtime for providing information about the environment (e.g. by the collecting sensor data) and the functional and non-functional properties of the system (e.g. by instrumenting the code and collecting runtime data). The monitoring information is passed to the awareness mechanism and may also give feedback to developers about the state of the system and the environment.

Awareness Conventional systems can react directly to the data obtained by the monitor but autonomous systems often need a deeper analysis. The awareness mechanism uses reasoning, planning and learning methods to determine the current situation of the system and to prepare the subsequent system behaviour.

Self-adaptation The adaptation mechanism implements the results of the awareness deliberations. In case of weak adaptation some control parameters of the system are modified or new functions are added or existing functions are modified. Strong adaptation means to modify the architecture of the system.

Deployment and Feedback Data The evolutionary loop connects the development cycle and the operations cycle. During deployment the system is prepared it for its execution. This involves installing, configuring and launching the application. The deployment may also involve executable code generation and compilation/linking.

The feedback data are collected by monitoring and in the awareness process. They may trigger a new Dev cycle and are used to provide information for system redesign, validation, verification, and redeployment.

3 The AIDL Life Cycle for Autonomous Systems

The AIDL life cycle specialises EDLC to techniques for systematically constructing autonomous policies that are based on reinforcement learning and planning techniques. We focus here on the specific issues of learning and planning. For simplicity, we restrict ourselves in this paper to systems with a single agent in an uncertain and possibly changing environment which may be noisy but is fully observable.

In AIDL, the modelling and programming phase of EDLC is extended by activities for constructing a digital twin. The use of a digital twin, i.e. a generative model of the environment dynamics, enables modeling of highly complex transition dynamics that would be unfeasible to capture by closed-form specification. Note that components of the twin (e.g. environment dynamics) can be learned from or adjusted to data collected from the environment.

At development time, the twin is used for training the system using learning algorithms. Validation is leveraging simulation for performing quantitative, sample-based statistical evaluation of a trained system. At runtime, the awareness mechanism is enriched with the twin for online planning.

Figure 2 shows the adjusted life cycle. The three new, small runtime cycles representing the digital twin are very similar to the operations cycle. In each of these cycles, the entities of the system iterate a “runtime feedback control loop” consisting of monitoring, awareness, and self-adaptation activities. The difference is that typically the additional cycles operate on the digital twin and not directly on the system.

The digital twin is not present in the requirements engineering phase. This is a natural choice, since the twin itself is specified here: It should comprise all relevant information about the application domain and cannot possibly inform itself.

Note that leveraging a digital twin could also be possible for further learning/training online at runtime, and simulation-based runtime verification in the monitoring phase. We do not treat these two applications of simulation at runtime in the following, and think that integrating them into AIDLE is an interesting avenue for future research.

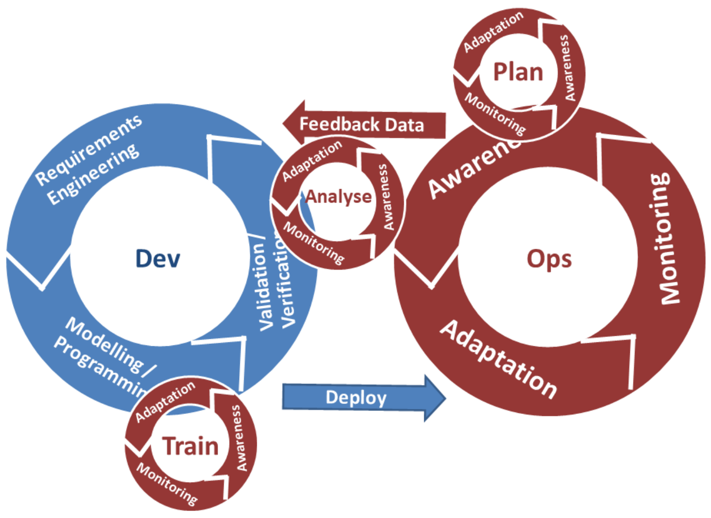

Fig. 2: AIDL life cycle.

3.1 Requirements

For engineering the system requirements we follow the ASCENS ideas and propose a goal-oriented approach where e.g. requirements elicitation can be performed using the ARE ontology. The requirement specification is the end product of the requirements elicitation process; it is defined by descriptions of the environment and of the domain of the envisaged system, adaptation requirements, and by a set of goals.

Domain description For systems which have to take autonomous decisions about the actions of the system, the environment and the required reliability of the system play an important role.

Often there is uncertainty on the behaviour of the environment which may also change dynamically. If the system contains embedded or IoT components, then it may not be fully reliable and certain system actions or components may fail. Such uncertain environments and systems are expressed by probability distributions. Formally, they form a kind of an MDP.

In many applications, in the beginning neither the initial distribution nor the transition distribution are known to the agent but they have to be learned based on the observations of the agent. In the case of a changing environment, or a changing system reliability, or changing goals, these distributions may change as well. Then the adaptation domain consists of a set of goals and MDPs, or of a probability distribution over goals and MDPs.

Goals. A goal represents a desirable system state or property that a software systems should achieve. For an autonomic system this is not always possible but one rather "strives to achieve" such a property, in spite of uncertainties and obstacles.

We distinguish optimisation goals and safety goals. Optimisation goals are soft goals that strive to achieve a property. Typically they are expressed with the help of an objective function and their goal is to find values that maximize or minimize this function.

Safety goals are hard goals which require that certain properties have to be preserved; in KAOS these are called maintain goals (if a property must always to hold) or avoid goals (if a property should never hold). For "classical" software systems such goals can be expressed by temporal logic formulae of the form "$\phi \implies \Box\psi$" (where ϕ and ψ are linear temporal logic formulae). But in presence of uncertainties, these safety properties cannot be universally true. They must consider probabilities for expressing the aleatoric uncertainty. It is also recommendable to estimate the epistemic uncertainty, i.e. the confidence in the validity of the result. Formally, we choose a probabilistic temporal logic such as PCTL [23] and express "soft" safety properties by formulas of the form "$P_{\geq p}(\psi)$ and $\mathbb{C}_{\geq c}$" which state that the goal ψ holds with at least probability p and at least confidence c ($p, c \in (0, 1)$).

3.2 Modelling and Programming

In this phase the system design and the implementation are developed. We focus here on the specific issues of learning and planning, i.e. the choice of the appropriate domain model and of the learning and planning algorithms, their implementation and the training of the agent.

Choice of domain model. The choice of the domain model depends on the kind of uncertainty of the environment and on the translation of goals into rewards and costs.

Aleatoric uncertainty of the environment. In the standard case of an (aleatoric) uncertainty, the environment can be described by a probability distribution, possibly depending on the current state and the action of the agent. Then the system can be modelled by (a variant of) a Markov decision process. For a specification comprising optimising and safety goals constrained Markov decision processes [7] are a good choice whereas for a specification with only one optimising goal a classical MDP is sufficient. If the application has noisy or unreliable sensors and the autonomic entity may not be able to determine the current state with complete reliability one may resort to Partially Observable Markov decision processes (POMDP) [32] as system model.

Changing environment. A more difficult situation arises if the changes of environment are not stationary. Then one can try to model this by a probability distribution over the set of possible environments or - if also the goals and the system may change - one may model the adaptation domain as a probability distribution over MDPs [18].

Specification of rewards and costs. Another issue is the definition of of the immediate rewards and costs. Often their values can be directly derived from the corresponding optimization and safety goals but the correspondence is not always obvious. In this case one can explore different reward and cost functions or try to adjust the rewards and costs in a next round of the development cycle.

Choice of algorithm. There is a wealth of learning and planning algorithms; none of them is known to outperform the others. The choice depends on several factors including the kind of domain model and goals, the size of the state and action space, the real-time performance requirements, and the availability of a digital twin. For a more detailed set of criteria see [39]

Kind of domain model and goals. For MDPs with only one optimising goal, there is wide selection of model-free reinforcement learning algorithms, model-based reinforcement learning algorithms, and online planning algorithms. For applications with safety constraints a scalarisation approach can be followed if during training and learning a policy, actions are not required to be always safe. Otherwise shielding and safe exploration algorithms can be used. Partially Observable Markov Decision Processes (POMDPs) can be solved by online planning algorithms or by combinations of reinforcement learning with planning [40]).

The case of complex adaptation domains with changing environments can be tackled by meta-learning and online planning. Meta-learning algorithms are well-suited for dealing with non-stationary environments that can be described as a probability distribution over a set of environments. Online planning methods are able to react to non-stationary changes of the environment as well as to changes of goals and to noisy actions.

Size of state and action space. For small state and action spaces classical tabular model-free algorithms can be used, e.g. Q-learning and Temporal Difference learning for discrete sets of actions or Gaussian processes in the continuous case. Also planning algorithms use tabular representation methods. When the state space becomes too large, one has to resort to approximate representations of the value function. These function approximations can be linear (such as Fourier transform) or nonlinear (such as deep or forward neural networks). Also combining function approximation and local tabular methods as in [52] is promising.

Because of the computational intractability of belief states, algorithms for POMDPs are mostly only applicable to small and mid size problems. These issues can by partially resolved by factorisation of the state space or by exploiting full observability whenever possible [43].

Performance and availability of a digital twin. By using function approximation, model-free methods scale to complex tasks (such as robotics and motion animation) but they need large amounts of samples and training. Instead, model-based algorithms and pure online planning require less training but need to rely on a faithful model of the environment. Errors in the model undermine the quality of the solutions but recent methods such as uncertainty estimation of the learned models can mitigate the model-bias [18].

The real-time performance of model-based algorithms such as PETS may be another issue, e.g. in case that action selection of the algorithm needs more time than a default time-step of the environment. In [61] T. Wang et al. provide benchmarks for several state-of-the-art reinforcement learning algorithms and show that model-based and model-free algorithms can achieve similar performances. Benchmarks for algorithms with safety constraints are given in [47].

Implementation and training. The task of implementing an autonomous agent by reinforcement learning is twofold: (1) implementing the model, the algorithm and the training pipeline and (2) synthesis of the policy via training (i.e. execution of the training cycle).

Implementation For implementing the model and algorithm one needs to define a software architecture for the appropriate variant of Markov Decision Processes, the learning algorithm, and the application. E.g. this can be an object-oriented architecture, differential equations, or a neural network architecture. For standard applications a reinforcement learning framework can be used such as the Reinforcement Learning Toolbox of MathWorks[4] or Gym of OpenAI[5]. Neural networks for function approximation can be implemented with the help of deep learning frameworks such as PyTorch[6] and TensorFlow[7].

Training The objective of training is to synthesize a policy which achieves the required optimizing goals and safety goals constraints.

Training is executed in a training cycle in which the learning agent interacts with the environment through a repeated trial-and-error process. A certain number of finite episodes are performed and the parameters of the policy are tuned for maximizing the cumulative reward, minimizing the loss, and for ensuring the required probability and confidence of the safety goals. Typically, after each episode the parameters of the policy are updated and - if possible - the environment is reset. Training options comprise the length of

[4] https://de.mathworks.com/products/reinforcement-learning.html
[5] https://gym.openai.com/
[6] https://pytorch.org/
[7] https://www.tensorflow.org/

an episode, the maximum number of episodes, the individual or the average rewards and costs.

Training can be performed in the real environment or by simulations on a digital model. The latter has the advantage that typically many more training cycles are possible and that it is reversible, i.e. that the environment can be reset. For many applications, training consists of two parts, a simulation on a digital twin of the application and training of the autonomous agent in the real environment.

A good training practice is to start with a simplified setting consisting of a simple simulated environment and a simple reinforcement learning algorithm. The algorithm and environment are then refined until the desired setting is achieved. Note that this approach means to iteratively run through deployment, operations cycle, feedback, validation, and re-adjustment of requirements and design until the model is accepted and then can be deployed for operation. For comparing the learning algorithms and choosing the most suitable one, a good practice is to deploy and train different algorithms in parallel.

3.3 Validation and Verification

Because of the large size of and the uncertainties about the environment, the construction of policies for autonomic systems requires extensive validation and verification. This is includes the validation and verification methods of ASCENS as well as all classical verification and testing methods such as unit, integration, system, and user testing, static analysis, and runtime verification.

A key issue is the statistical analysis of the training results such as the analysis of cumulative and average episode return. Statistical model checking [36] and Bayesian model checking [69] are the main tools for verifying safety constraints. Both methods use a runtime cycle for performing simulations and statistical analysis. In statistical model checking, finitely many randomised simulations of the system are executed and statistical methods are used for deciding whether the samples provide a statistical evidence for the satisfaction or violation of the specification. Bayesian model checking is a variant of statistical model checking which - instead of randomised sampling - incorporates prior information about the model being verified. The advantage is that it this often requires a significantly smaller number of sampled episodes.

The operation of autonomous systems has also to be validated in the real environment. This leads to several additional problems such as recognizing an environment which is different from the training environment, guaranteeing safe exploration of the real environment, and avoiding actions which disturb the real environment. For a discussion and possible solutions see [8].

3.4 Deployment and Feedback Data.

Deployment. Deployment is used in three phases of the AIDL life cycle: for connecting the development cycle with the operations cycle, for executing the training cycle, and during validation and verification. During deployment the system is prepared for execution in a simulated ("in vitro") or in a real environment ("in vivo"). This involves the choice of the runtime infrastructure and the choice of the real environment and in case of simulation, the choice of the parameters of the digital twin.

Other tasks are compilation, linking, and generation of executable code. Microcontrollers and GPUs are typical infrastructures for autonomous systems operating in real environments. Simulations are deployed into GPUs, clouds and clusters of cpus, the latter two for executing in parallel to improve training performance.

Feedback. Feedback is based on the data collected by monitoring and in the awareness process. The feedback data are used for validation and verification and for evaluating and improving the design, the implementation, and the requirements.

3.5 The Operations Cycle

The Ops cycle of AIDL is almost the same as the one of EDLC. It can run in the real environment or in a digital model. The only change is that it emphasizes an additional Ops cycle for simulations which are executed in the awareness phase.

Monitoring. As in EDLC, monitoring employs mechanisms such as sensor information and code instrumentation for collecting data about the state of the environment, of some components of the autonomous system or of the whole system. In the context of MDPs, monitoring comprises also runtime validation of design assumptions about MDP by observing statistical properties and simulation results and their relations with other factors. This includes uncertainty quantification and detecting non-functional changes such as drift and anomalies.

An important task is MDP identification, i.e. in case the agent is able to work in several environment the current environment is monitored and if a change of the environment is detected, the adaptation mechanism of the agent is triggered; using feedback, also a new development cycle may be activated.

For systems with several digital twins, monitoring can check the status of these twins and inform the awareness mechanism. Another use of monitoring is to survey the learning results in case the system continues learning during operation.

Awareness. In this phase reasoning and planning is carried out. The monitored data are evaluated and analysed w.r.t. required (functional) properties. Often simulations on the digital twin are performed for predicting the behaviour of system and environment. The results can then be used e.g. for online planning and deciding on the next steps of the agent.

Adaptation. Based on the awareness results, different forms of adaptation are possible. Weak adaptation amounts simply to execute the action selected by the policy or to change some control parameters of the algorithm such as the change of the direct reward. Strong adaptation means e.g. to change the dynamic model (MDP) by updating the direct reward or the transition distribution. Also the system may be reconfigured, e.g. by exchanging system components or sensor functions.

4 Case Studies

In this section we illustrate AIDL by two existing case studies: a simple search-and-rescue case study [10] and a so-called particle dance case study [12]. The search-and-rescue scenario is solved by online planning whereas the particle dance illustrates the systematic development of a reinforcement learning solution. In subsections 4.1 and 4.2 the two case studies are presented along the phases of the AIDL life cycle. Subsection 4.3 gives a short comparison of both approaches.

4.1 Case Study: Engineering Adaptation by Simulation-based Online Planning

The first example [10] is a simple search-and-rescue scenario which is solved by online planning. The experimental results shows that the generated planning policy of the agent is able to act autonomously and is robust w.r.t. unexpected events and changes of system goals at runtime.

Search-and-rescue scenario. A robot is deployed in a damaged area and must rescue victims by bringing them to an ambulance. If the robot encounters a fire, it has first extinguish the fire, and only then it can continue its way.

Requirements. The domain model consists of victims, fires, and ambulances in an environment with an unknown topology which is represented by a finite graph. The rescue robot can move (to a neighbor position), load or drop a victim (at its position), do nothing, and extinguish a fire (at a neighbor position). The robot has two goals. Its achieve goal is to find the victims and bring them to an ambulance. The safety constraint requires the agent to ignite all fires that are adjacent to its current position.

$$\textbf{Goal}\ \text{Achieve}\ \mathit{SaveVictims2Amb} : \Diamond\Big(\bigwedge_{i=1,\dots,n} \mathit{victim}_i\ \text{at ambulance}\Big) \tag{1}$$

$$\textbf{Goal}\ \text{Constraint}\ \mathit{IgniteFire} : \Box(\forall Fire f : \mathit{adjacent}(f) \Rightarrow \mathit{ignite}(f)) \tag{2}$$

There are also several adaptation requirements. First, the environment and the system may change: fires probabilistically ignite and cease; the actions of the robot are not reliable and may fail with a certain probability. The robot may also inadvertently drop the victim it is bringing to the ambulance. Moreover, the goals of the robot may change: the goal of saving victims from fire may change to the SaveVictims2Amb goal of “saving victims and bringing them to an ambulance.”

Modelling and programming. For modelling and implementing the search-and-rescue scenario, an object-oriented domain model of the scenario and a generic framework, called OnPlan, were developed. Then the domain model was plugged into OnPlan. Figure 3 shows the class diagram of the domain model.

OnPlan is a framework for modelling autonomous systems based on online planning [10]. it has a generic object-oriented architecture which realises an arbitrary MDP. It comprises components for states, actions, rewards, and also for the transition probabilities that define the policy (called strategy in [10]) of the robot. In addition, the architecture has a monitoring component for observing the environment and an abstract planning component which has a concrete simulation-based online planning component as realisation.

The latter makes use of a digital twin of the application domain for gathering information about potential system episodes. During the planning steps, future episodes are simulated at runtime. Simulation provides information about probability and value of the different state space regions, thus guiding system behaviour execution. After simulating possible choices of actions and behavioural alternatives, the transition probabilities of the MDP are updated and the agent executes an action (in reality) that performed well in simulation.

The dynamic model of OnPlan realises the behaviour of an Operations cycle (see below) and performs monitoring of the environment and planning and execution of actions iteratively at runtime.

Training is not necessary for OnPlan but the digital twin has to be a true model of the real environment.

OnPlan comes with two instantiations for online planning, Monte Carlo Tree Search [34] for discrete domains and the Gaussian approach of Cross Entropy Open Loop Planning [62] for continuous state and action spaces. For the search-and-rescue scenario, the former was used and plugged into the OnPlan architecture.

Validation. Validation is performed by statistical model checking using the MultiVesta tool [50]. Measurements include the estimation of the mean expected future reward.

A main aspect is the validation of the quality of the autonomous behaviour and of the robustness to changes. Concerning the autonomous behaviour, we test the system in an environment exhibiting aleatoric uncertainty: fires probabilistically ignite and cease and the actions of the robot are not reliable and may fail with a certain probability. Figure 4 shows that the planning component is able to generate a policy for transporting victims to safe positions autonomously.

Figures 5 and 6 address the adaptability and robustness of the system. Figure 5 shows the robot is able to recover from the unexpected events efficiently. The transportation of victims to safety is only marginally impaired by the sudden unexpected changes of the situation. The framework is also able to react adequately to a re-specification of system goals. In Fig. 6 before step 40, the robot was given a reward for keeping the number of fires low resulting in a reduction of the number of burning victims. On-wards from step 40, reward was instead provided for victims that have been transported to safety.

In all three figures 4, 5, and 6 the blue line indicates the percentage of saved victims, the red line the percentage of victims in fire, and the green line the percentage of positions in fire. Dotted lines indicate 0.95 confidence intervals.

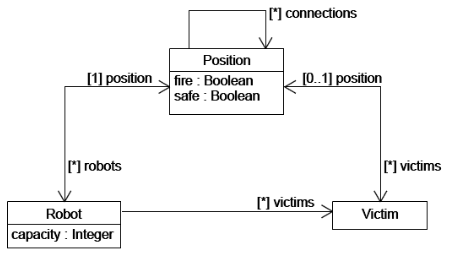

Fig. 3: Class diagram of search-and-rescue domain.

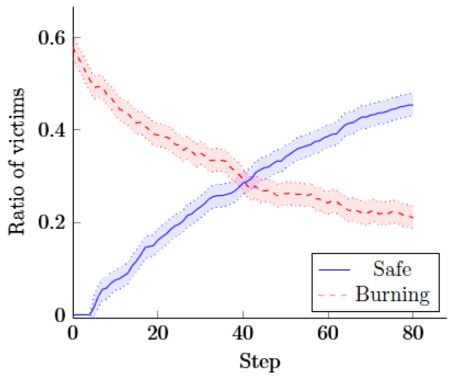

Fig. 4: Autonomous agent performance. Reward is given for victims at safe positions.

Operations cycle. Monitoring of the environment is performed by OnPlan through an operation "observe" which senses the whole graph including the current position of the robot, the victims and

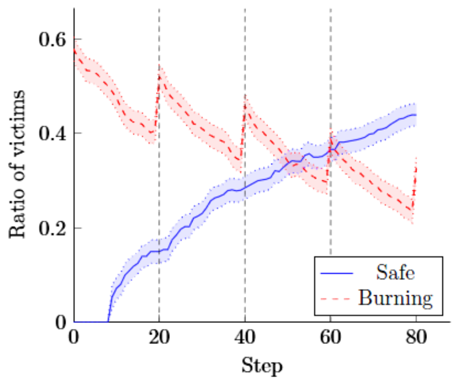

Fig. 5: Autonomous agent performance despite unexpected events at runtime. Every 20th step, all victims carried by the agent fall to the ground, and the number of fires raises to 10.

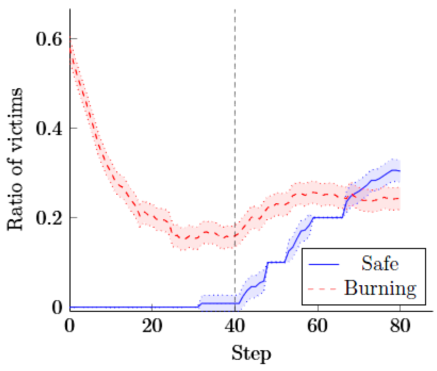

Fig. 6: Autonomous agent performance with a re-specification of system goal at runtime. Before step 40, the agent is given a reward for keeping the number of fires low, resulting in a reduction of the number of burning victims. Onwards from step 40, reward is provided for victims that have been transported to safety.

fires. In addition, it counts the fires and victims, and monitors the success of the rescue actions.

By simulating iteratively future episodes at runtime, the robot becomes aware of the current situation.

Short term weak self-adaptation is achieved by online planning. The policy is iteratively updated according to the results of the Monte Carlo Search and to the observations of the robot action.

Deployment, feedback, and new development cycle Deploying the OnPlan scenario is standard and consists of packaging the software and deploying it on the infrastructure. Feedback comes from the monitoring data which inform the developers about the current status of fires and victims as well the (victim saving and fire fighting) performance of the agent. Strong adaptation arises in case of goal revision. If e.g. the current agent goal was to extinguish fires but new victims are detected, then a new development cycle is initiated for changing the goal and giving rewards for saving the victims.

4.2 Case Study Safe learning: Policy SYnthesis with safety Constraints (PSyCo).

In line with the AIDL life cycle, the PSyCo approach is a systematic method for specifying and implementing agents that shape rewards dynamically over the learning process based on their confidence in requirement satisfaction [12]. It is centered around a safe reinforcement learning algorithm which combines evolutionary learning with Bayesian model checking. The basic idea is to emphasize return optimization when the learner is confident, and to focus on satisfying given constraints otherwise. This enables to explicitly distinguish requirements wrt. aleatoric uncertainty that is inherent to the domain, and epistemic uncertainty arising from an agent's learning process based on limited observations.

Particle Dance Scenario. In the Particle Dance scenario, an agent has to learn to follow a randomly moving particle as closely as possible.

Requirements. The domain is modelled as an MDP with an unknown transition distribution. State and action space are bounded continuous subsets of R. The reward computes the negative distance between particle and agent.

Minimising the distance means maximizing the reward. Thus the optimising goal is to maximise the expectation of the cumulative return $\mathcal{R}$:

$$\textbf{Goal} \text{ Optimize } \textit{Return} : \max \mathbb{E}(\mathcal{R}) \tag{3}$$

The safety constraint requires the agent to keep a minimum distance of the particle except in a fixed small number of cases. We require that the particle satisfies this requirement with a high probability and a high confidence.

$$\textbf{Goal} \text{ Constraint } \textit{BoundedCollisions} : \mathbb{P}_{\geq p_{\text{req}}}(\Box\phi) \text{ and } \mathbb{C}_{\geq c_{\text{req}}} \tag{4}$$

Here the formula ϕ expresses that the distance between particle and agent is greater than the minimum distance. Typically, we set the required probability for satisfying the constraint $p_{\text{req}} = 0.85$ and the required confidence $c_{\text{req}} = 0.98$.

Modelling, programming, and training. For dealing with the safety constraint, the MDP domain model of the Particle Dance is extended to form a Constrained MDP over continuous state and action spaces. The safety requirement $\Box\phi$ is transformed into a notion of cost $\mathcal{C}_\phi$ which (for each episode of Particle Dance) counts the number of violations of the safety property ϕ. The transformed goal is the following constrained optimization problem.

$$\max \mathbb{E}(\mathcal{R}) \text{ s.t. } \mathbb{P}_{\geq p_{\text{req}}}(\mathcal{C}_\phi = 0) \text{ and } \mathbb{C}_{\geq c_{\text{req}}} \tag{5}$$

To solve this goal, we developed the so-called Safe Neural Evolutionary Strategies (SNES) reinforcement learning algorithm. As usual in deep learning, SNES models a policy as neural network. SNES synthesises such policies by combining a safe evolutionary learning algorithm with an algorithm for Bayesian model checking. The basis of the safe learning algorithm is the Lagrangian approach for solving constrained MDPs [7] where the constrained problem

$$\max \mathcal{R} \text{ s.t. } \mathcal{C}_\phi = 0 \tag{6}$$

is transformed to a Lagrange formulation:

$$\max \mathcal{R} - (1-\lambda)\mathcal{C}_\phi \tag{7}$$

where $\lambda \in \mathbb{R}^+$ is a Lagrangian multiplier [9].

The resulting optimisation algorithm adaptively weights return and cost such that the resulting policy is likely to be positively verified. The algorithm does not ensure safety while learning, but only when converging to a solution of the Lagrangian. Bayesian model checking serves for modelling the epistemic uncertainty about the satisfaction probability of the results. We use it in two ways: To guide the learning process towards feasible solutions and to verify synthesized policies. The learning procedure of SNES is based on an evolutionary algorithm, called Evolutionary strategies (ES). This is a gradient free, search-based optimization algorithm that has shown competitive performance in reinforcement learning tasks.

Function approximation for SNES is realised by a feedforward neural network with one hidden layer. Training is performed over 60000 episodes (of length 50). Every 1000 episodes, Bayesian model checking is performed for a maximum of 1000 episodes (outside the learning loop of SNES) to evaluate the policy synthesized by SNES up to that point.

Validation. Validation is performed by Bayesian model checking and an analysis of e.g. the proportion of constraint satisfaction and the confidence in the results. For example, we can observe that the SNES agent learns to follow the particle closely. Figure 7 illustrates this by sample trajectories of the particle and the agent (color gradients denote time).

Figure 8 shows the proportion of episodes that satisfy the given requirement. We can see that the proportion closely reaches the defined bound of $p_{\text{req}} = 0.85$, shown by the dashed vertical line. Note that the satisfying proportion is closely above the required bound.

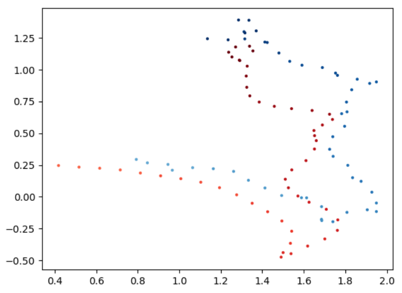
Fig. 7: Sample trajectories of the particle (light to dark blue, color gradient denotes time) and the agent (light to dark red).

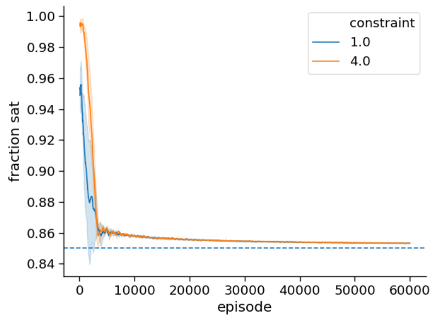

Fig. 8: Proportion of episodes satisfying cost requirement.

Figure 9 shows the confidence of the learning agent in its ability to satisfy the given requirement based on the observations made in the learning process. Note that the confidence is mostly kept above the confidence requirement $c_{\text{req}} = 0.98$ given in the specification. This shows SNES is effectively incorporating observations, probability requirements, and confidence into its learning process.

Operations cycle. Monitoring consists in sensing the distance of the agent to the particle and in recording the number of distance violations and the cumulative reward. Since the policy is synthesised the operations cycle consists of monitoring and executing the policy. Thus there is no explicit adaptation and awareness phase in this loop.

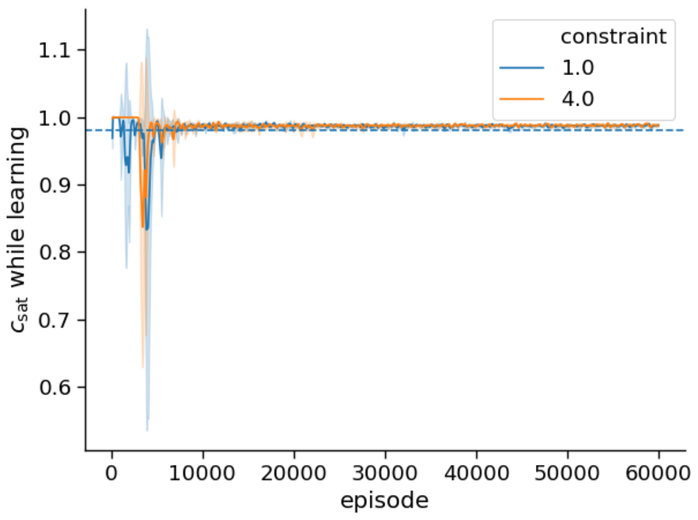

Fig. 9: Confidence c_{sat} in satisfying specification based on observations in the course of learning.

Deployment, feedback and new development cycle. As for OnPlan, deployment is standard and consists of packaging the software and deploying it on the infrastructure.

Feedback is given e.g. in case the monitor detects that the particle behaviour is changing or the agent behaviour is degrading so that the agent is not following closely the particle. Then a new development cycle is initiated for revising the requirements and the algorithm, new training and validation rounds, and finally the deployment of a revised policy.

4.3 Comparison

The above results show that although the requirements are similar the two solutions are complementary in many ways.

Both case studies have goals requiring a high level of confidence. The search-and-rescue scenario is modelled as an MDP whereas as the particle dance is modelled as a CMDP. The sets of states and actions are discrete and finite for the search-and-rescue case but infinite and continuous for the particle dance.

Reinforcement learning is model-free; but it needs many training cycles and thus is “slow” at “programming” time. Online planning is model-based; but it does not need any training and thus is fast at "programming" time. During the Ops cycle, the synthesised reinforcement policy is directly executed and thus is fast, whereas online planning uses runtime simulation which may be too slow for real-time applications.

Adaptation of the synthesised policy requires a new Dev cycle, whereas the online planning policy is able to act autonomously and is robust w.r.t. unexpected events at runtime. For a change of system goals, also online planning requires an adjustment of the rewards and thus a new development cycle.

5 Related Work

Systematic engineering approaches for intelligent agents and multi-agent systems such as Gaia [68], Tropos [14] support a sequential development process starting with the collection of goal-oriented requirements and the model of the environment and then proceeding with architectural design, detailed design, and implementation. Gaia follows an organisational metaphore where agents play roles whereas Tropos is founded on the BDI (*Belief*, *Desire*, and *Intention*) agent architecture [46]. The AgentComponent approach [35] proposes a component-based software development process fully based on UML models. IBM's approach to autonomic systems is based on the MAPE-K architecture [30] built around an "autonomic manager" that iterates a feedback control loop consisting of four activities: *Monitoring*, *Analysing*, *Planning*, and *Executing*. The "generic life cycle for context-aware adaptive systems" based on MAPE-K addresses foreseen and unforeseen evolution of the environment [31].

More recent development approaches for autonomous systems focus on specialised goal-oriented requirements (such as SOTA [5], GEM [29], and ARE [58]) and on feedback control loops (such as [16,60]). The SOTA [5] approach is an extension of existing goal-oriented requirements engineering that integrates elements of dynamic systems modelling. Semantically, SOTA is built on the General Ensemble Model GEM [29]. The Autonomous Requirements Engineering approach ARE [58] focusses on systematically eliciting so-called autonomy requirements.

Similar to AIDL and EDLC, DevOps [48] is an agile software development method which connects software development with runtime management based on a software life cycle. DevOps does not specialize on autonomous systems and its life cycle consists of only one life cycle instead of three. MLOps [3] instantiates DevOps to the development of machine learning applications but different from AIDL, it focusses on big data applications. AIOps [2] aims at automating and enhancing IT operations through analytics and machine learning, but it does not consider software development.

The engineering process [22] for machine learning is closely related to AIDL. It follows a different life cycle and addresses adaptive instead of autonomous systems, but as AIDL it proposes central activities necessary for developing machine learning applications.

The FRAP framework [39] does not aim for a full engineering life cycle; instead it identifies fine grain design decisions for reinforcement learning and planning algorithms. In addition to computational effort and function representation, criteria such as trial selection, return estimation, update procedure, and back-up are considered.

6 Concluding Remarks

In this paper we proposed a systematic development process, called AIDL, for constructing/ synthesizing policies of autonomous systems using planning and reinforcement learning techniques. AIDL can be seen as an instance of the EDLC development process for collective autonomic systems. It emphasizes the particular issues of machine learning techniques such as training, digital environment and agent models, and additional runtime cycles in almost all phases of development. We illustrated AIDL with two existing complementary case studies for reinforcement learning and online planning.

AIDL is not yet complete. Our two case studies address autonomic systems with single agents; a next step will be to refine and extend AIDL to AIDL-E for engineering collective autonomic systems. Also further learning/training online at runtime, simulation-based runtime verification in the monitoring phase, and additional non-functional requirements such as reliability, robustness, and security of policies should be discussed and integrated into our development approach. An ambitious mid term objective is to build an integrated development environment for AIDL.

An interesting methodical research question is how to use abstraction and refinement for stepwise learning of digital twin and how abstraction can help to learn policies. Also the relationship between aleatoric and epistemic uncertainty is not always straightforward and deserves further investigation.

Acknowledgement. We thank the anonymous reviewer for constructive criticisms and helpful suggestions.

A Markov Decision Processes

A Markov Decision Process (MDP) M defines a domain as a set S of states consisting of all states of the environment and the agent, a set of A of agent actions, and a probability distribution $T : p(S|S, A)$ describing the transition probabilities of reaching some successor state when executing an action in a given state. For expressing optimisation goals the labelled transition system is extended by a reward function $R : S \times A \times S \to \mathbb{R}$ which gives the expected immediate reward gained by the agent for taking each action in each state. Moreover, an initial state distribution $\rho : p(S)$ is given.

An episode $\vec{e} \in E$ is a finite or infinite sequence of transitions (s_i, a_i, s_{i+1}, r_i), $s_i, s_{i+1} \in S$, $a_i \in A, r_i = R(s_i, a, s_{i+1})$ in the MDP. For a given discount parameter $\gamma \in [0, 1]$ and any finite or infinite episode $\vec{e}$, the cumulative return $\mathcal{R}$ is the discounted sum of rewards $\mathcal{R} = \sum_{i=1}^{|\vec{e}|} \gamma^i r_i$. Depending on the application, the agent behaves in an environment according to a memoryless stationary policy $\pi : S \to p(A)$ or according to a deterministic memoryless policy $\pi : S \to A$ with the goal to maximise the expectation of the cumulative return $\mathbb{E}(\mathcal{R})$.

A partially observable Markov Decision Process (POMDP) [32] is a Markov decision process together with a set Ω of observations and an observation probability distribution $O : p(\Omega|S, A)$.

A Constrained Markov Decision Process (CMDP) has an additional cost function $C : S \times A \times S \to \mathbb{R}$ which can be used for expressing constraints and safety goals.

References

1. ASCENS: Autonomic Component Ensembles. Integrated Project, 2010-10-01 - 2015-03-31, Grant agreement no: 257414, EU 7th Framework Programme. http://www.ascens-ist.eu/, accessed on April 21, 2020.

2. Gartner, Inc.: Market Guide for AIOps Platforms. November 07, 2019. https://www.bmc.com/forms/tools-and-strategies-for-effective-aiops.html, accessed on October 07, 2020.
3. Google Cloud Solutions: MLOps: Continuous delivery and automation pipelines in machine learning. https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning, accessed on October 07, 2020.
4. OpenAI. Spinning Up in Deep RL! Part 2: Kinds of RL Algorithms, 2018. https://spinningup.openai.com, accessed on July 07, 2020.
5. D. Abeywickrama, N. Bicocchi, M. Mamei, and F. Zambonelli. The SOTA approach to engineering collective adaptive systems. *Int. J. Softw. Tools Technol. Transf.*, 22(4):399–415, 2020.
6. M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe reinforcement learning via shielding. In *AAAI*, pages 2669–2678. AAAI Press, 2018.
7. E. Altman. *Constrained Markov decision processes*, volume 7. CRC Press, 1999.
8. D. Amodei, C. Olah, J. Steinhardt, P. F. Christiano, J. Schulman, and D. Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.
9. B. Beavis and I. Dobbs. *Optimisation and Stability theory for Economic Analysis.* Cambridge University Press, 1990.
10. L. Belzner, R. Hennicker, and M. Wirsing. OnPlan: A framework for simulation-based online planning. In C. Braga and P. C. Ölveczky, editors, *Formal Aspects of Component Software - 12th Internat. Conf., FACS 2015, Revised Selected Papers*, volume 9539 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2015.
11. L. Belzner, M. M. Hölzl, N. Koch, and M. Wirsing. Collective autonomic systems: Towards engineering principles and their foundations. *Trans. Found. Mastering Chang.*, 1:180–200, 2016.
12. L. Belzner and M. Wirsing. Synthesizing safe policies under probabilistic constraints with reinforcement learning and Bayesian model checking. *Science of Computer Programming*, 206:102620, 2021.
13. M. Bernardo, R. De Nicola, and J. Hillston, editors. *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems, SFM 2016*, volume 9700 of *Lecture Notes in Computer Science*. Springer, 2016.
14. P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *JAAMAS*, 8(3):203–236, 2004.
15. C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton. A survey of Monte Carlo Tree Search methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1):1–43, 2012.
16. Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. M. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009.
17. T. Bureš, R. De Nicola, I. Gerostathopoulos, N. Hoch, M. Kit, N. Koch, G. V. Monreale, U. Montanari, R. Pugliese, N. Šerbedžija, M. Wirsing, and F. Zambonelli. A life cycle for the development of autonomic systems: The e-mobility showcase. *SASO Workshops*, 2013:71–76, 2013.
18. I. Clavera, J. Rothfuss, J. Schulman, Y. Fujita, T. Asfour, and P. Abbeel. Model-based reinforcement learning via meta-policy optimization. In *CoRL 2018*, volume 87 of *Proceedings of Machine Learning Research*, pages 617–629. PMLR, 2018.

19. R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *ACM Trans. Auton. Adapt*, 9(2):7:1–7:29, 2014.
20. M. M. Drugan. Reinforcement learning versus evolutionary computation: A survey on hybrid algorithms. *Swarm Evol. Comput.*, 44:228–246, 2019.
21. J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, and J. L. Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Nat. Comput.*, 12(1):43–67, 2013.
22. T. Gabor, A. Sedlmeier, T. Phan, F. Ritz, M. Kiermeier, L. Belzner, B. Kempter, C. Klein, H. Sauer, R. Schmid, J. Wieghardt, M. Zeller, and C. Linnhoff-Popien. The scenario coevolution paradigm: adaptive quality assurance for adaptive systems. *Int J Softw Tools Technol Transfer*, 2020.
23. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.
24. M. Hasanbeig, A. Abate, and D. Kroening. Cautious reinforcement learning with logical constraints. In *AAMAS*, pages 483–491. International Foundation for Autonomous Agents and Multiagent Systems, 2020.
25. N. Hoch, H. Bensler, D. B. Abeywickrama, T. Bures, and U. Montanari. The e-mobility case study. In *[65]*, pages 513–533.
26. P. Horn. *Autonomic computing: IBM perspective on the state of information technology.* IBM T.J.Watson Labs, NY, 2001.
27. M. M. Hölzl, N. Koch, M. Puviani, M. Wirsing, and F. Zambonelli. The ensemble development life cycle and best practices for collective autonomic systems. In *[65]*, pages 325–354, 2015.
28. M. M. Hölzl, A. Rauschmayer, and M. Wirsing. Engineering of software-intensive systems: state of the art and research challenges. In *[64]*, pages 1–44. 2008.
29. M. M. Hölzl and M. Wirsing. Towards a system model for ensembles. In *Formal Modeling: Actors, Open Systems, Biological Systems*, number 7000 in Lecture Notes in Computer Science, pages 241–261. Springer, 2011.
30. IBM. An architectural blueprint for autonomic computing. Technical report, IBM Corporation, 2005.
31. P. Inverardi and M. Mori. Software lifecycle process to support consistent evolutions. In *Software Engineering for Self-Adaptive Systems*, pages 239–264. 2010.
32. L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.
33. S. Kernbach, T. Schmickl, and J. Timmis. Collective adaptive systems: challenges beyond evolvability. *CoRR abs/1108.5643*, 2011.
34. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *EMCL 2006*, volume 4212 of *Lecture Notes in Artificial Intelligence*, pages 282–293. Springer, 2006.
35. R. Krutisch, P. Meier, and M. Wirsing. The agent component approach, combining agents, and components. In *First German Conference on Multiagent System Technologies, MATES 2003*, volume 2831 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003.
36. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV 2010*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.
37. M. Loreti and J. Hillston. Modelling and analysis of collective adaptive systems with CARMA and its tools. In Bernardo et al. [13], pages 83–119.
38. P. Mayer, J. Velasco, A. Klarl, R. Hennicker, M. Puviani, F. Tiezzi, R. Pugliese, J. Keznikl, and T. Bures. The autonomic cloud. In *[65]*, pages 495–512, 2015.

39. T. M. Moerland, J. Broekens, and C. M. Jonker. A framework for reinforcement learning and planning. *CoRR*, abs/2006.15009, 2020.
40. T. M. Moerland, J. Broekens, and C. M. Jonker. Model-based reinforcement learning: A survey. *CoRR*, abs/2006.16712, 2020.
41. T. M. Moerland, A. Deichler, S. Baldi, J. Broekens, and C. M. Jonker. Think too fast nor too slow: The computational trade-off between planning and reinforcement learning. *CoRR*, abs/2005.07404, 2020.
42. A. Nagabandi, I. Clavera, S. Liu, R. S. Fearing, P. Abbeel, S. Levine, and C. Finn. Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. In *ICLR 2019*. OpenReview.net, 2019.
43. S. C. W. Ong, S. W. Png, D. Hsu, and W. S. Lee. Planning under uncertainty for robotic tasks with mixed observability. *Int. J. Robot. Res.*, 29(8):1053–1068, 2010.
44. C. Pinciroli, M. Bonani, F. Mondada, and M. Dorigo. Adaptation and awareness in robot ensembles: Scenarios and algorithms. In *[65]*, pages 471–494.
45. M. Puviani, G. Cabri, and F. Zambonelli. Patterns for self-adaptive systems: agent-based simulations. *EAI Endorsed Trans. Self-Adaptive Systems*, 1(1):e4, 2015.
46. A. S. Rao and M. P. Georgeff. Modeling rational agents within a bdi-architecture. In *Proc. Knowledge Representation and Reasoning*, pages 473–484, 1991.
47. A. Ray, J. Achiam, and D. Amodei. Benchmarking safe exploration in deep reinforcement learning. Technical report, Open AI, 2019.
48. J. Roche. Adopting DevOps practices in quality assurance. *Commun. ACM*, 56(11):38–43, 2013.
49. S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. Online planning algorithms for pomdps. *J. Artif. Intell. Res.*, 32:663–704, 2008.
50. S. Sebastio and A. Vandin. Multivesta: statistical model checking for discrete event simulators. In *ValueTools '13*, pages 310–315. ICST/ACM, 2013.
51. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.
52. D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.
53. R. S. Sutton and A. G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 2^{nd} edition, 2018.
54. C. Szepesvári. *Algorithms for Reinforcement Learning.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.
55. M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *J. Mach. Learn. Res.*, 10:1633–1685, 2009.
56. S. Thrun and L. Y. Pratt. Learning to learn: Introduction and overview. In S. Thrun and L. Y. Pratt, editors, *Learning to Learn*, pages 3–17. Springer, 1998.
57. M. Tschaikowski and M. Tribastone. A unified framework for differential aggregations in Markovian process algebra. *J. Log. Alg. Meth. Prog.*, 84(2):238–258, 2015.
58. E. Vassev and M. Hinchey. Engineering requirements for autonomy features. In *[65]*, pages 379–403. 2015.
59. R. Vilalta, C. G. Giraud-Carrier, P. Brazdil, and C. Soares. Inductive transfer. In C. Sammut and G. I. Webb, editors, *Encyclopedia of Machine Learning and Data Mining*, pages 666–671. Springer, 2017.

60. N. Šerbedžija and S. Fairclough. Biocybernetic loop: From awareness to evolution. In *IEEE Evolutionary Computation 2009*, pages 2063–2069. IEEE, 2009.
61. T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba. Benchmarking model-based reinforcement learning. *CoRR*, abs/1907.02057, 2019.
62. A. Weinstein and M. L. Littman. Open-loop planning in large-scale stochastic domains. In *AAAI 2013*. AAAI Press, 2013.
63. D. Weyns, B. R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer, 2013.
64. M. Wirsing, J.-P. Banâtre, M. M. Hölzl, and A. Rauschmayer, editors. *Software-Intensive Systems and New Computing Paradigms - Challenges and Visions*, volume 5380 of *Lecture Notes in Computer Science*. Springer, 2008.
65. M. Wirsing, M. M. Hölzl, N. Koch, and P. Mayer, editors. *Software Engineering for Collective Autonomic Systems – The ASCENS Approach*, volume 8998 of *Lecture Notes in Computer Science*. Springer, 2015.
66. M. Wirsing, M. M. Hölzl, M. Tribastone, and F. Zambonelli. ASCENS: Engineering autonomic service-component ensembles. In *FMCO 2011*, volume 7542 of *Lecture Notes in Computer Science*, pages 1–24. 2013.
67. M. J. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *Knowledge Eng. Review*, 10(2):115–152, 1995.
68. F. Zambonelli, N. R. Jennings, and M. J. Wooldridge. Developing multiagent systems: The Gaia method. *ACM Trans. Softw. Eng. Meth.*, 12(3):317–370, 2003.
69. P. Zuliani, A. Platzer, and E. M. Clarke. Bayesian statistical model checking with application to Simulink verification. *Formal Meth. Syst. Des.*, 43(2):338–367, 2013.