OnPlan: A Framework for Simulation-Based Online Planning

Lenz Belzner(✉), Rolf Hennicker, and Martin Wirsing

Institut für Informatik, Ludwig-Maximilians-Universität München, Munich, Germany
belzner@ifi.lmu.de

Abstract. This paper proposes the OnPlan framework for modeling autonomous systems operating in domains with large probabilistic state spaces and high branching factors. The framework defines components for acting and deliberation, and specifies their interactions. It comprises a mathematical specification of requirements for autonomous systems. We discuss the role of such a specification in the context of simulation-based online planning. We also consider two instantiations of the framework: Monte Carlo Tree Search for discrete domains, and Cross Entropy Open Loop Planning for continuous state and action spaces. The framework's ability to provide system autonomy is illustrated empirically on a robotic rescue example.

1 Introduction

Modern application domains such as machine-aided robotic rescue operations require software systems to cope with uncertainty and rapid and continuous change at runtime. The complexity of application domains renders it impossible to deterministically and completely specify the knowledge about domain dynamics at design time. Instead, high-level descriptions such as probabilistic predictive models are provided to the system that give an approximate definition of chances and risks inherent to the domain that are relevant for the task at hand.

Also, in contrast to classical systems, in many cases there are numerous different ways for a system to achieve its task. Additionally, the environment may rapidly change at runtime, so that completely deterministic behavioral specifications are likely to fail. Thus, providing a system with the ability to compile a sensible course of actions at runtime from a high-level description of its interaction capabilities is a necessary requirement to cope with uncertainty and change.

One approach to deal with this kind of uncertain and changing environments is *online planning*. It enables system autonomy in large (or even infinite) state spaces with high branching factors by interleaving planning and system action execution (see e.g. [1–3]). In many domains, action and reaction are required very often, if not permanently. Resources such as planning time and computational power are often limited. In such domains, online planning replaces the requirement of absolute optimality of actions with the idea that in many situations it

C. Braga and P.C. Ölveczky (Eds.): FACS 2015, LNCS 9539, pp. 1–30, 2016.
DOI: 10.1007/978-3-319-28934-2_1

is sufficient and more sensible to conclude as much as possible from currently available information within the given restricted resources. One particular way to perform this form of rapid deliberation is based on simulation: The system is provided with a generative model of its environment. This enables it to evaluate potential consequences of its actions by generating execution traces from the generative model. The key idea to scale this approach is to use information from past simulations to guide the future ones to directions of the search space that seem both likely to happen and valuable to reach.

In this paper we propose the OnPlan framework for modeling autonomous systems operating in domains with large or infinite probabilistic state spaces and high branching factors. The remainder of the paper is outlined as follows. In Sect. 2 we introduce the OnPlan framework for online planning, define components for acting and deliberation, and specify their interactions. We then extend this framework to simulation-based online planning. In Sects. 3 and 4 we discuss two instantiations of the framework: Monte Carlo Tree Search for discrete domains (Sect. 3), and Cross Entropy Open Loop Planning for continuous state and action spaces (Sect. 4). We illustrate each with empirical evaluations on a robotic rescue example. Section 5 concludes the paper and outlines potential lines of further research in the field.

2 A Framework for Simulation-Based Online Planning

In this Section we propose the OnPlan framework for modeling autonomous systems based on online planning. We introduce the basic concept in Sect. 2.1. In Sect. 2.2, we will refine the basic framework to systems that achieve autonomy performing rapidly repeated simulations to decide on their course of action.

2.1 Online Planning

Planning is typically formulated as a search task, where search is performed on sequences of actions. The continuously growing scale of application domains both in terms of state and action spaces requires techniques that are able to (a) reduce the search space effectively and (b) compile as much useful information as possible from the search given constrained resources. Classical techniques for planning have been exhaustively searching the search space. In modern application scenarios, the number of possible execution traces is too large (potentially even infinite) to get exhaustively searched within a reasonable amount of time or computational resources.

The key idea of online planning is to perform planning and execution of an action iteratively at runtime. This effectively reduces the search space: A transition that has been executed in reality does not have to be searched or evaluated by the planner any more. Online planning aims at effectively gathering information about the *next* action that the system should execute, exploiting the available resources such as deliberation time and capabilities as much as possible. Algorithm 1 captures this idea informally. In the following, we will introduce the OnPlan framework that formalizes the idea of online planning.

Algorithm 1. Online Planning (Informally)

```
1: while true do
2:     observe state
3:     plan action
4:     execute action
5: end while
```

Framework Specification. The OnPlan framework is based on the following requirements specification.

1. A set $\mathcal{S}_{\text{real}}$ which represents states of real environments. While this is a part of the mathematical formulation of the problem domain, it is not represented by a software artifact in the framework.
2. A set *Agent* that represents deliberating and acting entities.
3. Representations of the agent's observable state space $\mathcal{S}$ and the agent's action space $\mathcal{A}$. The observable state space $\mathcal{S}$ represents information about the environment $\mathcal{S}_{\text{real}}$ that is relevant for an agent and its planning process. It is in fact an abstraction of the environment.
4. A function $observe : Agent \times \mathcal{S}_{\text{real}} \to \mathcal{S}$ that specifies how an agent perceives the current state of its environment. This function defines the abstraction and aggregation of information available to an agent in its real environment to an abstract representation of currently relevant information. In some sense, the function *observe* comprises the monitor and analyze phases of the MAPE-K framework for autonomous computing [4].
5. A function $actionRequired : Agent \times \mathcal{S} \to Bool$ that is used to define triggering of action execution by an agent. A typical example is to require execution of an action after a certain amount of time has passed since the last executed action.
6. For each action in $\mathcal{A}$, we require a specification of how to execute it in the real domain. To this end, the framework specification comprises a function $execute : \mathcal{A} \times \mathcal{S}_{\text{real}} \to \mathcal{S}_{\text{real}}$. This function defines the real (e.g. physical) execution of an agent's action.
7. We define a set *RewardFunction* of reward functions of the form $R : \mathcal{S} \to \mathbb{R}$. A reward function is an encoding of the system goals. States that are valuable should be mapped to high values by this function. States that should be avoided or even are hazardous should provide low values.
8. We define a set *Strategy* of strategies. Each strategy is a probability distribution $P_{\text{act}}(\mathcal{A}|\mathcal{S})$ of actions over states. In the following, we will often omit the signature and simply write P_{act} for $P_{\text{act}}(\mathcal{A}|\mathcal{S})$. It defines the probability that an agent executes a particular action in a given state. If an agent $a \in Agent$ in state $s_{\text{current}} \in \mathcal{S}$ is required to act (i.e. when $actionRequired(a, s_{\text{current}})$ returns true), then the action that is executed is sampled from the distribution: $a \sim P_{\text{act}}(\cdot|s_{\text{current}})$, where $P_{\text{act}}(\cdot|s_{\text{current}})$ denotes the probability distribution of actions in state s_{current} and $\sim$ denotes sampling from this distribution. Sampling can be seen as non-deterministic choice proportional to a distribution.

9. A set *Planner* of planning entities. Planning is defined by a function $plan : Planner \times \mathcal{S} \times RewardFunction \times Strategy \rightarrow Strategy$. A planning entity refines its strategy P_{act} w.r.t. its currently observed abstract state and a reward function to maximize the expected cumulative future reward. It is usually defined as the sum of rewards gathered when following a strategy.

Framework Model. Figure 1 shows a class diagram for the OnPlan framework derived from the mathematical specification. It comprises classes for the main components *Agent* and *Planner*. States and actions are also represented by a class each: states $s \in \mathcal{S}$ are represented by objects of class *State*, actions $a \in \mathcal{A}$ by objects of class *Action*. Probability distributions of actions over states (defining potential agent strategies) are modeled by the class *Strategy*. Reward functions are represented by object of class *RewardFunction*. All classes are abstract and must be implemented in a concrete online planning system.

Note that OnPlan supports multiple instances of agents to operate in the same domain. While inter-agent communication is not explicitly expressed in the framework, coordination of agents can be realized by emergent system behavior: As agents interact with the environment, the corresponding changes will be observed by other agents and incorporated into their planning processes due to the online planning approach.

Component Behavior. Given the specification and the component model, we are able to define two main behavioral algorithms for *Agent* and *Planner* that are executed in parallel: *Agent* $\|$ *Planner*. I.e., this design decouples information aggregation and execution (performed by the agent) from the deliberation process (performed by the planner).

Algorithms 2 and 3 show the behavior of the respective components. We assume that all references shown in the class diagram have been initialized. Both behaviors are infinitely looping. An agent observes the (real) environment, encodes the observation to its (abstract) state and passes the state to its corresponding planning component, as long as no action is required (Algorithm 2, lines 2–5). When an action is required – e.g. due to passing of a certain time frame or occurrence of a particular situation/event – the agent queries the planner's current strategy for an action to execute (line 6). Finally, the action proposed by the strategy is executed (line 7) and the loop repeats.

The behavior of the planning component (Algorithm 3) repeatedly calls a particular planning algorithm that refines the strategy w.r.t. current state and specified reward function. We will define a particular class of planning algorithms in more detail in Sect. 2.2.

Framework Plug Points. The OnPlan framework provides the following plug points derived from the mathematical specification. They are represented by abstract operations such that domain specific details have to be implemented by any instantiation.

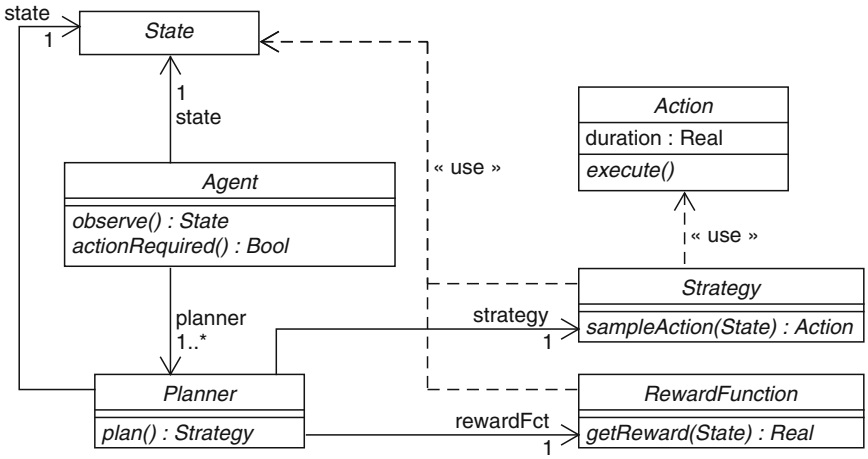


Fig. 1. Basic components of OnPlan

Algorithm 2. Agent Component Behavior

Require: Local variable action : *Action*

```
1: while true do
2:     while !actionRequired() do
3:         state ← observe()                          ▷ observe environment
4:         planner.state ← state                      ▷ inform planner
5:     end while
6:     action ← planner.strategy.sampleAction(state)  ▷ sample from strategy
7:     action.execute()                               ▷ execute sampled action
8: end while
```

Algorithm 3. Planner Component Behavior

```
1: while true do
2:     strategy ← plan()
3: end while
```

1. The operation *Agent::observe() : State*. This operation is highly dependent on the sensory information available and is therefore implemented in a framework instantiation.
2. The operation *Agent::actionRequired() : Bool*. The events and conditions that require an agent to act are highly depending on the application domain. The timing of action execution may even be an optimization problem for itself. The state parameter of the mathematical definition is implicitly given by the reference of an agent to its state.
3. The operation *Action::execute()*. Action execution is also highly dependent on technical infrastructure and physical capabilities of an agent.
4. The operation *RewardFunction::getReward(State) : Real*. Any concrete implementation of this operation models a particular reward function.

5. The operation *Strategy::sampleAction(State)* : *Action* should realize sampling of actions from the strategy w.r.t. to a given state. It depends on the used kind of strategy, which may be discrete or continuous, unconditional or conditional, and may even be a complex combination of many independent distributions.
6. Any implementation of the operation *Planner::plan()* should realize a concrete algorithm used for planning. Note that the arguments of the function *plan* from the mathematical specification are modeled as references from the *Planner* class to the classes *State*, *RewardFunction* and *Strategy*. We will discuss a particular class of simulation-based online planners in the following Sect. 2.2.

2.2 Simulation-Based Online Planning

We now turn our focus on a specific way to perform online planning: *simulation based online planning*, which makes use of a simulation of the domain. It is used by the planner to gather information about potential system episodes (i.e. execution traces). Simulation provides information about probability and value of the different state space regions, thus guiding system behavior execution. After simulating its possible choices and behavioral alternatives, the agent executes an action (in reality) that performed well in simulation. The process of planning using information from the simulation and action execution is iteratively repeated at runtime, thus realizing online planning.

A simple simulation based online planner would generate a number of randomly chosen episodes and average the information about the obtained reward. However, as it is valuable to generate as much information as possible with given resources, it is a good idea to guide the simulation process to high value regions of the search space. Using variance reduction techniques such as importance sampling, this guidance can be realized using information from previously generated episodes [5–7].

Framework Specification. In addition to the specification from Sect. 2.1, we extend the OnPlan framework requirements to support simulation-based online planning.

1. For simulation based planning, actions $a \in \mathcal{A}$ require a duration parameter. If no such parameter is specified explicitly, the framework assumes a duration of one for the action. We define a function $d : \mathcal{A} \to \mathbb{R}$ that returns the duration of an action.
2. OnPlan requires a set *Simulation* of simulations of the environment. Each simulation is a probability distribution of the form $P_{\text{sim}}(\mathcal{S}|\mathcal{S} \times \mathcal{A})$. It takes the current state and the action to be executed as input, and returns a potential successor state according to the transition probability. Simulating the execution of an action $a \in \mathcal{A}$ in a state $s \in \mathcal{S}$ yields a successor state $s' \in \mathcal{S}$. Simulation is performed by sampling from the distribution P_{sim}: $s' \sim P_{\text{sim}}(\cdot|(s, a))$, where $P_{\text{sim}}(\cdot|(s, a))$ denotes the probability distribution of successor states

when executing action a in state s and $\sim$ denotes sampling from this distribution. Note that the instantiations of the framework we discuss in Sects. 3 and 4 work with a fixed simulation of the environment. It does not change in the course of system execution, in contrast to the strategy.

3. We require a set $\mathit{SimPlanner} \subseteq \mathit{Planner}$ of simulation based planners.
4. Any simulation based planner defines a number $e_{\max} \in \mathbb{N}^+$ of episodes generated for each refinement step of its strategy.
5. Any simulation based planner defines a maximum planning horizon $h_{\max} \in \mathbb{N}^+$ that provides an upper bound to its simulation depth. A low planning horizon results in fast but shallow planning – long term effects of actions are not taken into account when making a decision. The planning horizon lends itself to be dynamically adapted, providing flexibility by allowing to choose between fast and shallow or more time consuming, but deep planning taking into account long term consequences of actions.
6. Any simulation based planner defines a discount factor $\gamma \in [0; 1]$. This factor defines how much a planner prefers immediate rewards over long term ones when refining a strategy. The lower the discount factor, the more likely the planner will built a strategy that obtains reward as fast as possible, even if this means an overall degradation of payoff in the long run. See Algorithm 5 for details on discounting.
7. We define a set $\mathcal{E} \subseteq (\mathcal{S} \times \mathcal{A})^*$ of episodes to capture simulated system execution traces. We also define a set $\mathcal{E}_w \subseteq \mathcal{E} \times \mathbb{R}$ of episodes weighted by the discounted sum of rewards gathered in an execution trace. The weight of an episode is defined as its cumulative discounted reward, which is given by the recursive function $R_{\mathcal{E}} : \mathcal{E} \to \mathbb{R}$ as shown in Eq. 1. Let $s \in \mathcal{S}$, $a \in \mathcal{A}$, $e, e' \in \mathcal{E}$ where $e = (s, a) :: e'$, and let $R : \mathcal{S} \to \mathbb{R}$ be a reward function.

$$\begin{aligned} R_{\mathcal{E}}(\mathrm{nil}) &= 0 \\ R_{\mathcal{E}}(e) &= R(s) + \gamma^{d(a)} R_{\mathcal{E}}(e') \end{aligned} \tag{1}$$

   An element of $\mathcal{E}_w$ is then uniquely defined by $(e, R_{\mathcal{E}}(e))$.
8. In the OnPlan framework, the simulation-based planner uses the simulation P_{sim} to generate a number of episodes. The resulting episodes are weighted according to rewards gathered in each episode, w.r.t. the given reward function of the planner. Simulation is driven by the current strategy P_{act}. This process is reflected by following function.

$$\begin{aligned} &\mathit{generateEpisode} : \\ &\quad \mathit{SimPlanner} \times \mathit{Simulation} \times \mathit{Strategy} \times \mathit{RewardFunction} \to \mathcal{E}_w \end{aligned}$$

9. Importance sampling in high value regions of the search space is realized by using the resulting weighted episodes to refine the strategy such that its expected return (see Sect. 2.1) is maximized. The goal is to incrementally increase the expected reward when acting according to the strategy by gathering information from simulation episodes in an efficient way. This updating of the strategy is modeled by the following function.

$$\mathit{updateStrategy} : \mathit{SimPlanner} \times 2^{\mathcal{E}_w} \times \mathit{Strategy} \to \mathit{Strategy}$$

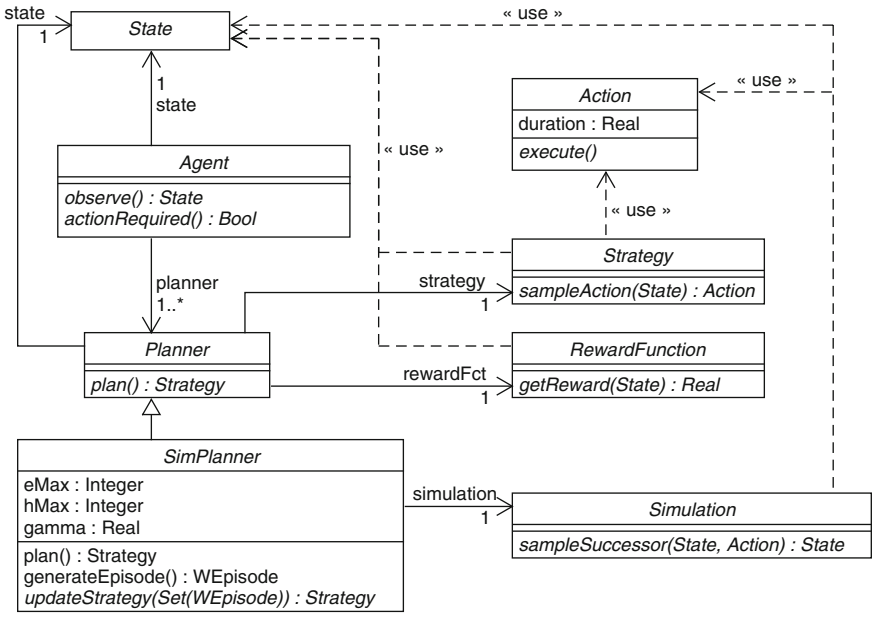


Fig. 2. Components of the OnPlan framework

Algorithm 4. Simulation-based planning

Require: Local variable E_w : *Set(WEpisode)*
1: **procedure** PLAN
2: $E_w \leftarrow \emptyset$
3: **for** 0 ... eMax **do**
4: $E_w \leftarrow E_w \cup$ generateEpisode()
5: **end for**
6: **return** updateStrategy(E_w)
7: **end procedure**

Framework Model. Using mathematically justified approaches for strategy refinement provides a solution to the notorious exploration-exploitation tradeoff (see e.g. [8]): While learning (or planning), an agent has to decide whether it should exploit knowledge about high-value regions of the state space, or whether it should use its resources to explore previously unknown regions to potentially discover even better options. We will discuss two instances of OnPlan that provide principled and mathematically founded methods that deal with the question where to put simulation effort in Sects. 3 and 4.

Figure 2 shows the components of the OnPlan framework for simulation-based online planning. It comprises the components of the basic OnPlan framework (Sect. 2.1), and additionally defines a specialization *SimPlanner* of the *Planner* class, and a class *Simulation* that models simulations of the form P_{sim}. The parameters $e_{\max}, h_{\max}$ and γ are modeled as attributes of the *SimPlanner* class.

Algorithm 5. Generating weighted episodes

```
Require: Local variables s : State, r, t : Real, e : Episode, a : Action
 1: procedure GENERATEEPISODE
 2:     s ← state
 3:     r ← rewardFct.getReward(s)
 4:     t ← 0
 5:     e ← nil
 6:     for 0 ... hMax do
 7:         a ← strategy.sampleAction(s)
 8:         e ← e::(s, a)
 9:         s ← simulation.sampleSuccessor(s, a)
10:         t ← t + a.duration
11:         r ← r + gamma^t · rewardFct.getReward(s)
12:     end for
13:     return (e, r)
14: end procedure
```

We further assume a class *WEpisode* that models weighted episodes. As it is a pure data container, it is omitted in the class diagram shown in Fig. 2.

The *SimPlanner* class also provides two concrete operations. The operation *SimPlanner::plan() : Strategy* realizes the corresponding abstract operation of the *Planner* class and is a template method for simulation based planning (see Algorithm 4). Episodes are modeled by a type *Episode*, weighted episodes by a type *WEpisode* respectively. The function *generateEpisode* is realized by the concrete operation *generateEpisode() : WEpisode* of the *SimPlanner* class and used by the *plan* operation. The function *updateStrategy* from the mathematical specification is realized as abstract operation *updateStrategy(Set(WEpisode))* in the class *SimPlanner*.

Simulation-Based Planning. *SimPlanner* realizes the *plan* operation by using a simulation to refine its associated strategy. We formalize the algorithm of the *plan* operation in the following. Algorithm 4 shows the simulation-based planning procedure. The algorithm generates a set of episodes weighted by rewards (lines 2–5). This set is the used to refine the strategy (line 6). The concrete method to update the strategy remains unspecified by OnPlan.

Algorithm 5 shows the generation of a weighted episode. After initialization (lines 2–5), an episode is built by repeating the following steps for $h_{\max}$ times.

1. Sample an action $a \in \mathcal{A}$ from the current strategy w.r.t. the current simulation state $s \in \mathcal{S}$, i.e. $a \sim P_{\text{act}}(s)$ (line 7).
2. Store the current simulation state and selected action in the episode (line 8).
3. Simulate the execution of a. That is, use the action a sampled from the strategy in the previous step to progress the current simulation state s, i.e. $s \sim P_{\text{sim}}(s, a)$ (line 9).
4. Add the duration of a to the current episode time $t \in \mathbb{R}$. This is used for time-based discounting of rewards gathered in an episode (line 10).

5. Compute the reward of the resulting successor state discounted w.r.t. current episode time t and the specified discount factor γ, and add it to the reward aggregation (line 11).

After simulation of $h_{\max}$ steps, the episode is returned weighted by the aggregated reward (line 13).

Framework Plug Points. In addition to the plug points given by the basic framework (see Sect. 2.1), the framework extension for simulation-based online planning provides the following plug points.

1. The operation *sampleSuccessor(State, Action): State* of class *Simulation*. This operation is the interface for any implementation of a simulation P_{sim}. The concrete design of this implementation is left to the designer of an instance of the framework. Both simulations for discrete and continuous state and action spaces can instantiate OnPlan. Note that, as P_{sim} may be learned from runtime observations of domain dynamics, this operation may be intentionally underspecified even by an instantiated system. Also note that the implementation of this operation does not necessarily have to implement the real domain dynamics. As simulation based planning typically relies on statistical estimation, any delta of simulation and reality just decreases estimation quality. While this also usually decreases planning effectiveness, it does not necessarily break planning completely. Thus, our framework provides a robust mechanism to deal with potentially imprecise or even erroneous specifications of P_{sim}.
2. The operation *updateStrategy(Set(WEpisode)): Strategy* of class *SimPlanner*. In principle, any kind of stochastic optimization technique can be used here. Examples include Monte Carlo estimation (see e.g. [6]) or genetic algorithms. We will discuss two effective instances of this operation in the following: Monte Carlo Tree Search for discrete domains in Sect. 3, and Cross Entropy Open Loop Planning for domains with continuous state-action spaces in Sect. 4.

Figure 3 shows an informal, high-level summary of OnPlan concepts and their mutual influence. Observations result in the starting state of the simulations.

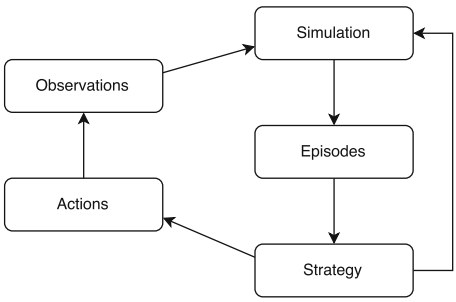


Fig. 3. Mutual influence of OnPlan concepts

Simulations are driven by the current strategy and yield episodes. The (weighted) episodes are used to update the strategy. The strategy yields actions to be executed. Executed actions influence observations made by an agent.

In the following Sections, we will discuss two state-of-the-art instances of the OnPlan framework for simulation-based online planning introduced in Sect. 2. In Sect. 3, we will illustrate Monte Carlo Tree Search (MCTS) [9] and its variant UCT [10] as an instantiation of OnPlan in discrete domains. in Sect. 4, we will discuss Cross Entropy Open Loop Planning (CEOLP) [3,11] as an instance of OnPlan for simulation based online planning in continuous domains with infinite state-actions spaces and branching factors.

3 Framework Instantiation in Discrete Domains

In this Section we discuss Monte Carlo Tree Search (MCTS) as an instantiation of the OnPlan framework in discrete domains.

3.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) provided a framework for the first discrete planning approaches to achieve human master-level performance in playing the game Go autonomously [12]. MCTS algorithms are applied to a vast field of application domains, including state-of-the-art reinforcement learning and planning approaches in discrete domains [2,9,13].

MCTS builds a search tree incrementally. Nodes in the tree represent states and action choices, and in each node information about the number of episodes an its expected payoff is stored. MCTS iteratively chooses a path from the root to leaf according to these statistics. When reaching a leaf, it simulates a potential episode until search depth is reached. A new node is added to the tree as a child of the leaf, and the statistics of all nodes that were traversed in this episode are updated according to the simulation result.

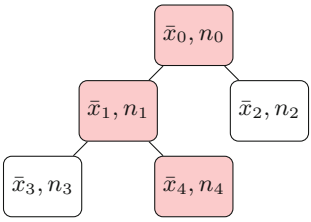
Figure 4 illustrates an iteration of MCTS. Each iteration consists of the following steps.

1. Nodes are selected w.r.t. node statistics until a leaf is reached (Fig. 4a).
2. When a leaf is reached, simulation is performed and the aggregated reward is observed (Fig. 4b).
3. A new node is added per simulation, and node statistics of the path selected in step (a) are updated according to simulation result (Fig. 4c).

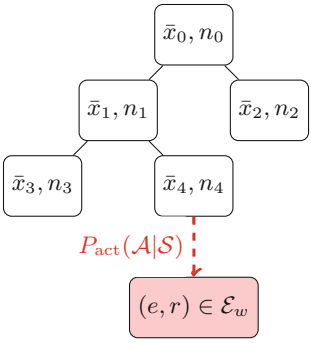
Steps (1) to (3) are repeated iteratively, yielding a tree that is skewed towards high value regions of the state space. This guides simulation effort towards currently promising search areas.

3.2 UCT

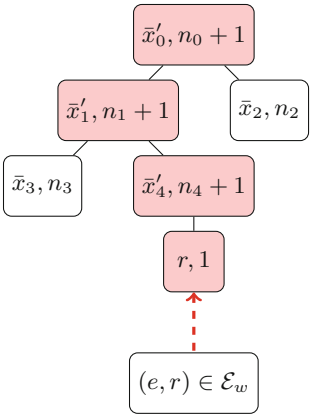
UCT (upper confidence bounds applied to trees) is an instantiation of MCTS that uses a particular mechanism for action selection in tree nodes based on



(a) Selection



(b) Simulation



(c) Update

Fig. 4. Illustration of Monte Carlo Tree Search. $(e, r) \in \mathcal{E}_w$ is a weighted episode as generated by Algorithm 5. Nodes' mean values can be updated incrementally (see e.g. [14]): $\bar{x}'_i = \bar{x}_i + \frac{r - \bar{x}_i}{n_i + 1}$.

regret minimization [10]. UCT treats action choices in states as multi-armed bandit problems. Simulation effort is distributed according to the principle of *optimism in the face of uncertainty* [15]: Areas of the search space that have shown promising value in past iterations are more likely to be explored in future

ones. UCT uses the mathematically motivated upper confidence bound for regret minimization UCB1 [16] to formalize this intuition. The algorithm stores the following statistics in each node.

1. $\bar{x}_a$ is the average accumulated reward in past episodes that contained the tuple (s, a), where s is the state represented by the current node.
2. n_s is the number of episodes that passed the current state $s \in \mathcal{S}$.
3. n_a is the corresponding statistic for each action a that can be executed in s.

Equation 2 shows the selection rule for actions in UCT based on node statistics. Here, $c \in \mathbb{R}$ is a constant argument that defines the weight of exploration (second term) against exploitation (first term). The equation provides a formalization of the exploration-exploitation tradeoff – the higher the previously observed reward of a child node, the higher the corresponding UCT score. However, the more often a particular child node is chosen by the search algorithm, the smaller the second term becomes. At the same time, the second term increases for all other child nodes. Thus, child nodes that have not been visited for some time become more and more likely to be included into future search episodes.

$$UCT(s, a) = \bar{x}_a + 2c\sqrt{\frac{2 \ln n_s}{n_a}} \tag{2}$$

3.3 Framework Instantiation

Monte Carlo Tree Search instantiates the OnPlan framework for simulation-based online planning based on the following considerations.

1. *Strategy::sampleAction(State): Action* is instantiated by the action selection mechanism used in MCTS. As MCTS is a framework itself, the particular choice is left underspecified. Examples of action selection mechanisms include uniform selection (all actions are chosen equally often), ϵ-greedy selection (the action with best average payoff is selected, with an ϵ probability to chose a random action) or selection according to UCT (see Eq. 2). Note that also probabilistic action selection strategies can be used, providing support for mixed strategies in a game-theoretic sense. Simulation outside the tree is performed according to an initial strategy. Typically, this is a uniformly random action selection. However, given expert knowledge can also be integrated here to yield potentially more valuable episodes with a higher probability.
2. *SimPlanner::updateStrategy(Set(WEpisode)): Strategy* adds the new node to the tree and updates all node statistics w.r.t. the simulated episode weighted by accumulated reward. Note that a single episode suffices to perform an update. Different mechanisms for updating can be used. One example is averaging rewards as described above. Another option is to set nodes' values to the maximum values of their child nodes, yielding a Monte Carlo Bellman update of the partial state value function induced by the search tree [2].

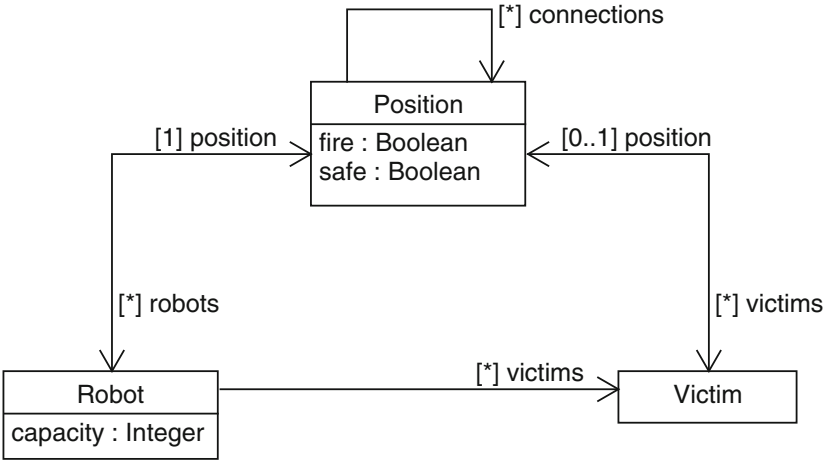


Fig. 5. Class diagram of the example domain.

3. While multiple simulations may be performed from a node when leaving the tree, typically the update (adding a node and updating all traversed nodes' statistics) is performed after each iteration. Thus, when using mcts for simulation-based planning, the number of episodes per strategy update $e_{\max}$ is usually set to 1.
4. The remaining plug-points – *execute* of class *Action*, *getReward* of class *RewardFunction* and *sampleSuccessor* of class *Simulation* – have to be instantiated individually for each domain and/or system use case.

3.4 Empirical Results

We implemented an instantiation of OnPlan with uct in an example search-and-rescue scenario to show its ability to generate autonomous goal-driven behavior and its robustness w.r.t. unexpected events and changes of system goals at runtime.

Example Domain. Figure 5 shows a class diagram of the scenario. A number of arbitrarily connected positions defines the domains topology. At some positions there is an ambulance (pos.safe = true). Positions may be on fire, except those that host an ambulance, i.e. class *Position* has the following invariant: pos.safe implies not(pos.fire) for all pos ∈ *Position*. Fires ignite or cease probabilistically depending on the number of fires at connected neighbor positions. A position may host any number of robots and victims. A robot can carry a number of victims that is bounded by its capacity. A carried victim does not have a position. A robot has five types of actions available.

1. Do nothing.
2. Move to a neighbor position that is not on fire.

3. Extinguish a fire at a neighbor position.
4. Pick up a victim at the robot's position if capacity is left.
5. Drop a carried victim at the robot's position.

All actions have unit duration. Each action may fail with a certain probability, resulting in no effect. Note that the number of actions available to a robot in a particular situation may vary due to various possible instantiations of action parameters (such as the particular victim that is picked up or the concrete target position of movement).

Experimental Setup. In all experiments, we generated randomly connected topologies with 20 positions and a connectivity of 30 %, resulting in 6 to 7 connections per position on average. We randomly chose 3 safe positions, and 10 that were initially on fire. 10 victims were randomly distributed on the non-safe positions. We placed a single robot agent at a random starting position. All positions were reachable from the start. Robot capacity was set to 2. The robot's actions could fail with a probability of up to 5 %, chosen uniformly distributed for each run. One run consisted of 80 actions executed by the agent. Results for all experiments have been measured with the statistical model checker MultiVeStA [17]. In all experiments, we set the maximum planning depth $h_{\max} = 20$. The discount factor was set to $\gamma = 0.9$. As MCTS was used for planning, we set $e_{\max} = 1$: The tree representing $P_{\text{act}}(\mathcal{A}|\mathcal{S})$ is updated after every episode. UCT's exploratory constant was set to $c = 20$ in all experiments.

In the following experiments, we let the agent deliberate for 0.2 s. That is, *actionRequired()* returned true once every 0.2 s; i.e. each action was planned for 0.2 s, incorporating information from past planning steps.

As long as not stated otherwise, we provided a reward of 100 to the planning agent for each victim that was located at a safe position. Let $I : \mathit{Bool} \to \{0, 1\}$ be an indicator function that yields 1 if the argument is defined and true and 0, otherwise. Let $\mathit{victims} : \mathcal{S} \to 2^{\mathit{Victim}}$ be the set of all victims present in a given state. Then, for any state $s \in \mathcal{S}$ the reward function was defined as follows.

$$R(s) = 100 \cdot \sum_{v \in \mathit{victims}(s)} I(v.\mathit{position.safe}) \tag{3}$$

The reward function instantiates the *getReward* operation of class *RewardFunction* in the OnPlan framework. Action implementations instantiate the *execute* operations of the corresponding subclasses of the *Action* class (e.g. move, pick up victim, etc.). A simulation about domain dynamics is provided to the simulation-based planner. It instantiates the *sampleSuccessor* operation of the *Simulation* class.

Estimation of Expected Future Reward. In a preliminary experiment, we observed the estimation of mean expected future reward. The MCTS planner increases the expected future reward up to step 60. Onwards from step 60 it decreases as the agent was informed about the end of the experiment after 80 steps.

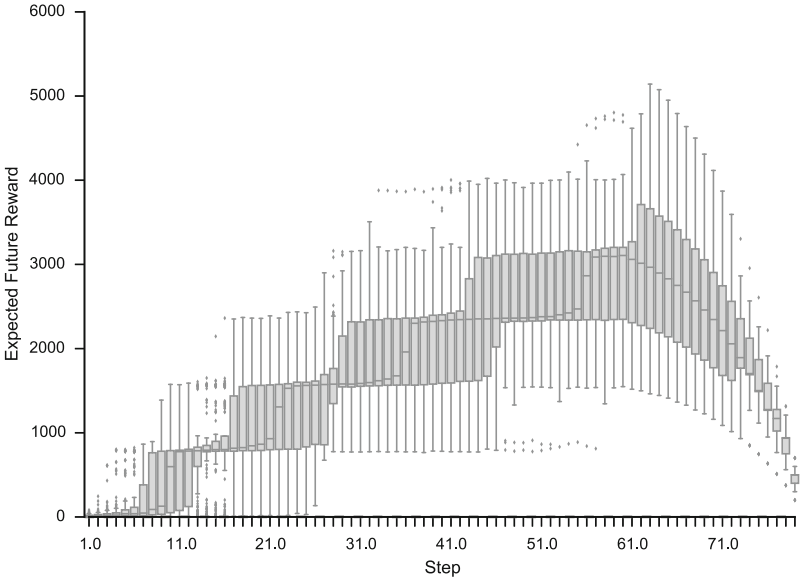


Fig. 6. Expected accumulated future reward at each step by the MCTS planner.

The planning depth $h_{\max} = 20$ thus detects the end of an experiment at step 60. The mean expected reward for executed actions is shown in Fig. 6.

We also measured the increase in accuracy of the estimation of expected reward by MCTS. We measured the normalized coefficient of variation (CV) to investigate estimation accuracy, as the mean of expected future reward is highly fluctuating in the course of planning. The CV is a standardized measure of dispersion of data from a given distribution and independent from the scale of the mean, in contrast to standard deviation. Normalization of the CV renders the measurement robust to the number of samples. The normalized CV of a sample set is defined as quotient of the samples' standard deviation s and their mean $\bar{x}$, divided by the square root of available samples n. Note that the CV decreases as n increases, reflecting the increased accuracy of estimation as more samples become available.

$$\frac{s/\bar{x}}{\sqrt{n}} \tag{4}$$

We recorded mean $\bar{r}$ and standard deviation s_a of the expected reward gathered from simulation episodes for each potential action a, along with the number of episodes where a was executed at the particular step n_a. The normalized CV of an action then computes as follows.

$$\frac{s_a/\bar{r}}{\sqrt{n_a}} \tag{5}$$

Figure 7 shows the normalized CV w.r.t. expected reward of the actions executed by the agent at a given step in the experiment. We observed that MCTS

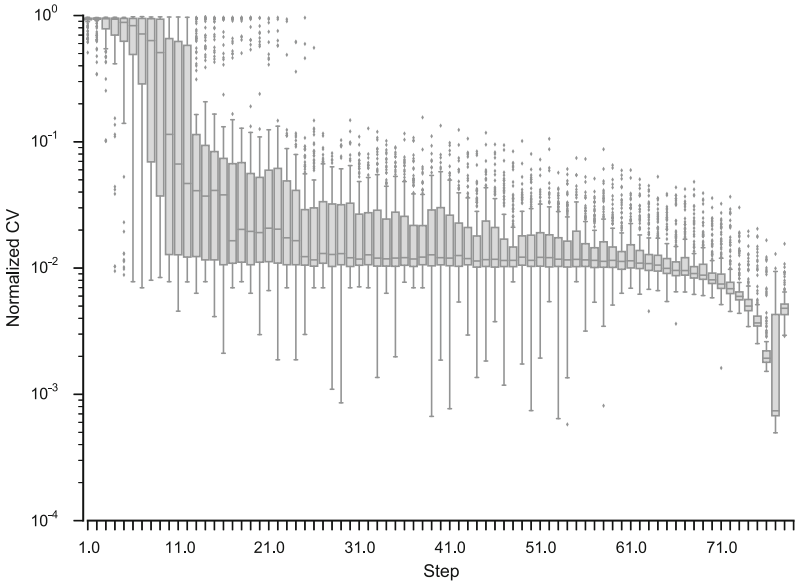


Fig. 7. Normalized coefficient of variation of expected reward estimation for actions executed by the MCTS planner. Note the logarithmic scale on the y-axis. After about 20 steps, estimation noise resembles the noise level inherent to the domain (up to 5 % action failures and average spread of fires).

steadily improves its estimation accuracy of expected reward. After about 20 steps, estimation noise resembles the noise level inherent to the domain (up to 5 % action failures and average spread of fires).

Autonomous System Behavior. In a baseline experiment, we evaluated OnPlan's ability to synthesize autonomous behavior according to the given reward function. Figure 8 shows the average ratio of victims that was at a safe position w.r.t. to the number of actions performed by the agent, within a 95 % confidence interval. Also, the ratio of victims that are located at a burning position is displayed. No behavioral specification besides the instantiation of our planning framework has been provided to the agent. It can be seen that the planning component is able to generate a strategy that yields sensitive behavior: The robot transports victims to safe positions autonomously.

Robustness to Unexpected Events. In a second experiment we exposed the planning agent to unexpected events. This experiment is designed to illustrate robustness of the OnPlan framework to events that are not reflected by the simulation P_{sim} provided to the planning component. In this experiment, all victims currently carried by the robot fall to the robot's current position every 20 steps. Also, a number of fires ignite such that the total number of fires accumulates to 10. Note that these events are not simulated by the agent while

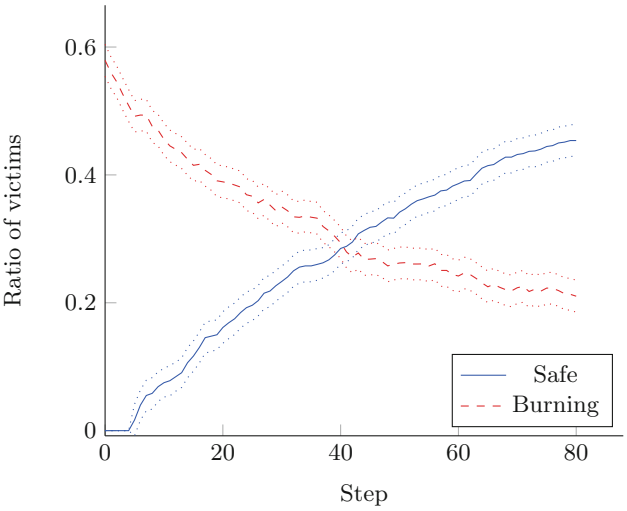


Fig. 8. Autonomous agent performance based on an instantiation of the OnPlan framework with a MCTS planner. Reward is given for victims at safe positions. Dotted lines indicate 0.95 confidence intervals.

planning. Figure 9 shows the agent's performance in the presence of unexpected events with their 95 % confidence intervals. It can be seen that transportation of victims to safety is only marginally impaired by the sudden unexpected changes of the situation. As MCTS is used in an online manner that is based on replanning at each step, the planning framework is able to recover from the unexpected events efficiently.

System Goal Respecification. A third experiment highlights the framework's ability to adapt behavior synthesis to a system goal that changes at runtime. Before step 40, the agent was 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 was instead provided for victims that have been transported to safety. Besides respecification of the reward function to reflect the change of system goal no additional changes have been made to the running system. I.e., only the *rewardFct* reference of the planner was changed. This change impacts the weighting of episodes (see Algorithm 5, lines 3 and 11). The different weighting in turn impacts the updating of the planner's current strategy.

Figure 10 shows system performance in this experiment, together with 95 % confidence intervals. The results indicate that the framework indeed is able to react adequately to the respecification of system goals. As system capabilities and domain dynamics remain the same throughout the experimental runtime, all high-level specifications such as action capabilities (i.e. the action space $\mathcal{A}$) and knowledge about domain dynamics (i.e. the generative model P_{sim}) are sensibly employed to derive valuable courses of actions, regardless of the current system goal. OnPlan thus provides a robust system adaptation mechanism for runtime goal respecifications.

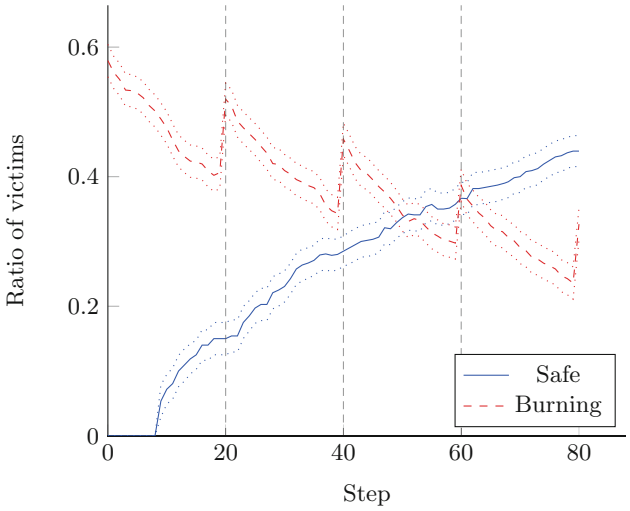


Fig. 9. Autonomous agent performance despite unexpected events at runtime. Every 20th step, all victims carried by the agent fall to the ground, and the number of fires raises to 10. Dotted lines indicate 0.95 confidence intervals.

4 Framework Instantiation in Continuous Domains

We now focus on an instantiation of the OnPlan framework that works in continuous space and action domains. I.e. states and actions are represented as vectors of real numbers $\mathbb{R}^n$, for some $n \in \mathbb{N}$. This means that state and action spaces are of infinite size. In this section we show how Cross Entropy Open Loop Planning (CEOLP) [3,11] instantiates our planning framework, and illustrate how information obtained from simulations in the planning process can be used to identify promising ares of the search space in continuous domains. CEOLP works by optimizing action parameters w.r.t. expected payoff by application of the cross entropy method.

4.1 Cross Entropy Optimization

The cross entropy method for optimization [7,18] allows to efficiently estimate extrema of an unknown function $f : X \to Y$ via importance sampling. To do so, an initial probability distribution (that we call sampling distribution) $P_{\text{sample}}(X)$ is defined in a way that covers a large region of the function's domain. For estimating extrema of f, a set of samples $x \in X$ is generated w.r.t. the sampling distribution (i.e. $x \sim P_{\text{sample}}(X)$). The size of the sample set is a parameter of the cross entropy method. For all x in the set, the corresponding $y = f(x) \in Y$ is computed. Then samples are weighted w.r.t. their relevance for finding the function extrema. For example, when trying to find maxima of f, samples x are weighted according to $y = f(x)$. Typically this involves normalization to keep sample weights in the $[0; 1]$ interval. We denote the weight of a sample x_i by w_i.

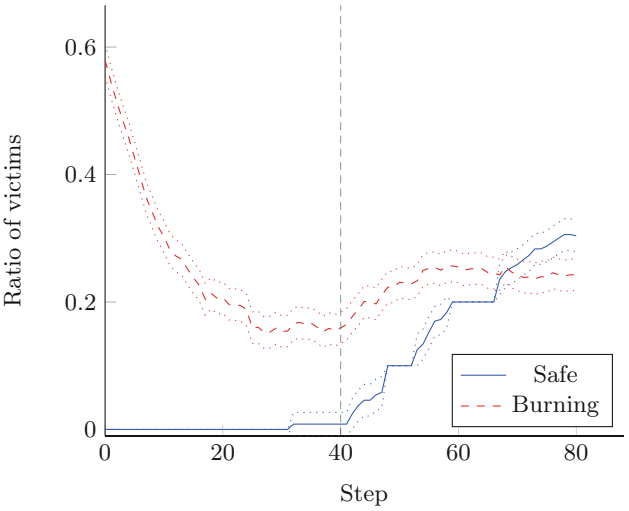


Fig. 10. Autonomous agent performance with a respecification of system goal at runtime. Before step 40, the agent is given a reward for keeping the number of fires low, resulting in a reduction of the number of burning victims. Onwards from step 40, reward is provided for victims that have been transported to safety. Dotted lines indicate 0.95 confidence intervals.

The weighted sample set is used to update the sampling distribution $P_{\text{sample}}(X)$ by minimizing the distributions' cross entropy. Minimizing cross entropy yields a distribution that is more likely to generate samples in X that are located close to the maxima of f. Minimizing of cross entropy has been shown to be equivalent to maximizing the likelihood of the samples x weighted by $f(x)$ [18]. Sampling, weighting and building new sampling distributions by maximum likelihood estimation are repeated iteratively. This yields an iterative refinement of the sampling distribution which increases the probability to sample in the region of the maxima of f, thus providing a potentially better estimate thereof. While convergence of the CE method has been proven for certain conditions, it is not easy to establish these conditions in the most practically relevant settings [19]. However, empirical results indicate that the CE method provides a robust optimization algorithm which has been applied successfully in a variety of domains (see e.g. [18,20,21])

Figure 11 illustrates the idea of iterative refinement of the sampling distribution to increase the probability to generate samples in the region of the maxima of the unknown function f. In this example, a Gaußian sampling distribution was chosen. The sampling distribution is shown as solid line, while the unknown target function is shown as dashed line. While in this Figure the target function has a Gaußian form as well, this is not required for the cross entropy method to work. Initially, the sampling distribution has a large variance, providing a well spread set of samples in the initial generation. Then the samples are weighted w.r.t. their value $f(x)$ and a maximum likelihood estimate is built from the

weighted samples. This yields a Gaußian sampling distribution that exposes less variance than the initial one. Repeating this process finally yields a distribution that is very likely to produce samples that are close to the maximum of the unknown target function.

Sampling from a Gaußian distribution can for example be done via the Box-Muller method [22]. Equation 6 shows a maximum likelihood estimator for a Gaußian distribution (μ, σ^2), given a set I of n samples $\boldsymbol{a}_i \in \mathcal{A}, i \in \{0, ..., n\}$, each weighted by $w_i \in \mathbb{R}$. This yields a new Gaußian distribution that concentrates its probability mass in the region of samples with high weights. Samples with low weights are less influential on the probability mass of the new distribution.

$$\begin{aligned} \mu &= \frac{\sum_{(\boldsymbol{a}_i, w_i) \in I} w_i \boldsymbol{a}_i}{\sum_{(\boldsymbol{a}_j, w_j) \in I} w_j} \\ \sigma^2 &= \frac{\sum_{(\boldsymbol{a}_i, w_i) \in I} w_i (\boldsymbol{a}_i - \mu)^T (\boldsymbol{a}_i - \mu)}{\sum_{(\boldsymbol{a}_j, w_j) \in I} w_j} \end{aligned} \tag{6}$$

Summarizing, the requirements for the cross entropy method are as follows.

1. A way to weight the samples, i.e. a way to compute $f(x)$ for any given $x \in X$.
2. An update mechanism for the distribution based on the weighted samples has to be provided. Typically, this is a maximum likelihood estimator for the sampling distribution.

Note that the cross entropy method is not restricted to a particular form of probability distribution. Also discrete distributions or other continuous ones than a Gaußian can be used to model the sampling distribution [18].

4.2 Online Planning with Cross Entropy Optimization

The key idea of CEOLP is to use cross entropy optimization on a *sequence* of actions. The agent's strategy $P_{\text{act}}(\mathcal{A}|\mathcal{S})$ is thus represented by a *vector* of multivariate Gaußian distributions over the parameter space of the actions $\mathcal{A} \subseteq \mathbb{R}^{\mathbb{N}}$.

In the context of our framework for simulation-based planning, we want to find the maxima of a function that maps sequences of actions to expected rewards, that is $f : \mathcal{A}^* \to \mathbb{R}$. The simulation $P_{\text{sim}}(\mathcal{S}|\mathcal{S} \times \mathcal{A})$ and the reward function $R : \mathcal{S} \to \mathbb{R}$ allow us to estimate $f(\boldsymbol{a})$ for any given $\boldsymbol{a} \in \mathcal{A}^*$: We can generate a sequence of states $\boldsymbol{s} \in \mathcal{S}^*$ by sampling from the simulation and build an episode $e \in \mathcal{E}$ from $\boldsymbol{a}$ and $\boldsymbol{s}$. We can then evaluate the accumulated reward of this episode by computing the discounted sum of gathered rewards $R_{\mathcal{E}}(e)$ (see Eq. 1).

In OnPlan, we generate $e_{\max}$ episodes and weight them by accumulated reward as shown in Algorithm 5. The sampling of actions from the strategy (Algorithm 5, line 9) is done by generating a sample from the Gaußian distribution over action parameters at the position of the strategy vector that matches the current planning depth (i.e. the number of iteration of

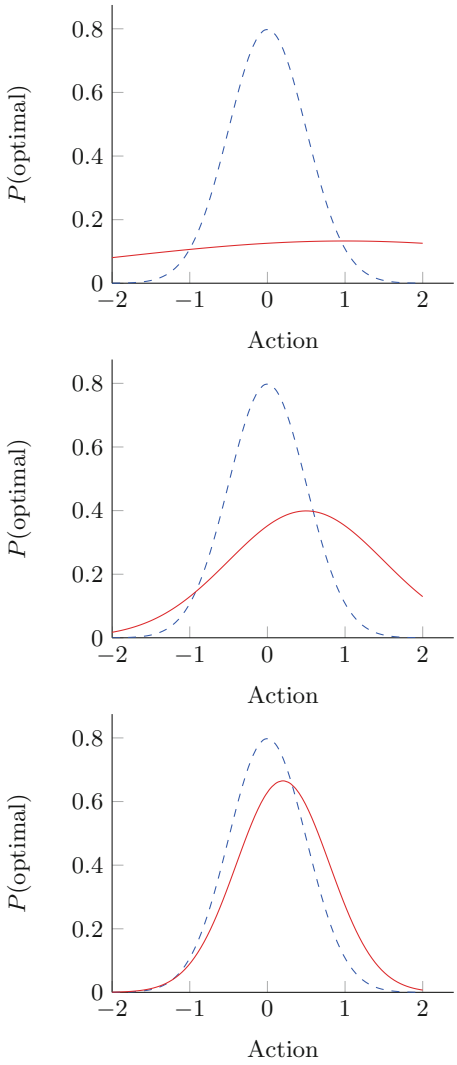


Fig. 11. Illustration of the cross entropy method with a Gaußian sampling distribution. The dashed line represents the unknown target function. The solid line represents the Gaußian sampling distribution that is iteratively refined by maximum likelihood estimation based on samples from the previous iteration, weighted by their target function values.

the for-loop in Algorithm 5, line 6). The Gaußians that form the strategy $P_{\text{act}}(\mathcal{A}|\mathcal{S})$ are updated after generating and weighting $e_{\max}$ episodes, as stated in Algorithm 4. The update is performed via maximum likelihood estimation for each Gaußian in the strategy vector as defined in Eq. 6.

4.3 Framework Instantiation

Cross Entropy Open Loop Planning instantiates the OnPlan framework based on the following considerations.

1. *Strategy::sampleAction(State): Action* generates samples from the current vector of Gaußians that represents P_{act}. As CEOLP is state agnostic and only accumulates action parameters w.r.t. planning depth, this depth is the only information that is used for conditioning the distribution: I.e. when sampling at depth d, the d-th component of the plan distribution is used to generate a value for the action.
2. *SimPlanner::updateStrategy(Set(WEpisode)): Strategy* refines the Gaußians in the strategy by maximum likelihood estimation w.r.t. the samples from the previous generation, weighted by the accumulated reward (see Eq. 6). This yields a strategy that is likely to produce high-reward episodes.
3. The remaining plug-points – *execute* of class *Action*, *getReward* of class *RewardFunction* and *sampleSuccessor* of class *Simulation* – have to be instantiated individually for each domain and/or system use case.

4.4 Empirical Results

We compared an instantiation of our framework with CEOLP with a vanilla Monte Carlo planner that does not perform iterative update of its strategy. The latter proposes actions from a strategy distribution that is the best average w.r.t. weighted simulation episodes. However, in contrast to the CEOLP planner, it does not refine the strategy iteratively while simulating to concentrate its effort on promising parts of the search space.

Experimental Setup. We provided the same number of simulations per action to each planner. The one that instantiates OnPlan updates the strategy distribution every 30 simulations (i.e. $e_{\max} = 30$) and does this 10 times before executing an action. Planning depth was set to $h_{\max} = 50$. The vanilla Monte Carlo planner uses the aggregated result of 300 episodes generated w.r.t. the initial strategy to decide on an action, without updating the strategy within the planning process. It only builds a distribution once after all samples have been generated and evaluated to decide on an action. Action duration was fixed at one second. The discount factor was set to $\gamma = 0.95$ in all experiments.

Example Domain. Figure 12 depicts our evaluation scenario. The circle bottom left represents our agent. Dark rectangular areas are static obstacles, and small boxes are victims to be collected by the planning agent. The agent is provided with unit reward on collecting a victim. Victims move with Gaußian random motion (i.e. their velocity and rotation are randomly changed based on a normal distribution). Note that this yields a highly fluctuating value function of the state space – a plan that was good a second ago could be a bad idea to

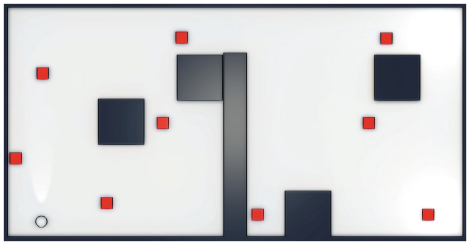


Fig. 12. The continuous sample domain.

realize a second later. This means that information aggregation from simulations should be as efficient as possible to be able to react to these changes in real time.

An agent can perform an action by first rotating for a second and then moving forward for the same amount of time. Rotation rate and movement speed are action parameters to be optimized by the planner in order to collect the victims as fast as possible. The agent is provided with a simulation of the environment as described above. Note that this simulation is an abstraction of the real environment. This means that reality and simulation may differ in their dynamics, even if performing the exactly same set of actions. Also, the simulation is not informed about the movement model of the victims.

The reward function providing unit reward to a planner on collecting a victim instantiates the *getReward* operation of class *RewardFunction* in the OnPlan framework. Action implementations instantiate the *execute* operations of the corresponding subclasses of class *Action*. The simulation provided to the simulation-based planner instantiates the *sampleSuccessor* operation of the *Simulation* class.

Iterative Parameter Variance Reduction. Figure 13 shows an exemplary set of actions sampled from P_{act} for the first action to be executed. Here, the effect of updating the sampling strategy can be seen for the two-dimensional Gaußian distribution over the action parameters speed (x axis) and rotation rate (y axis). While the distribution is spread widely in the initial set of samples, updating the strategies according to the samples' weights yields distributions that increasingly concentrate around regions of the sample space that yield higher expected reward. The figures also show Spearman's rank correlation coefficient of the sampled action parameters to measure the dependency between the two action variables (speed and rotation rate). It can be seen that the degree of correlation increases with iterations. Also, the probability that there is no statistically significant correlation of the parameters decreases: From 0.94 in the initial set of samples to 0.089 in the tenth set.

Estimation of Expected Reward. Figures 14 and 15 show the effect of iteratively updating the strategy on simulation episode quality. We evaluated the

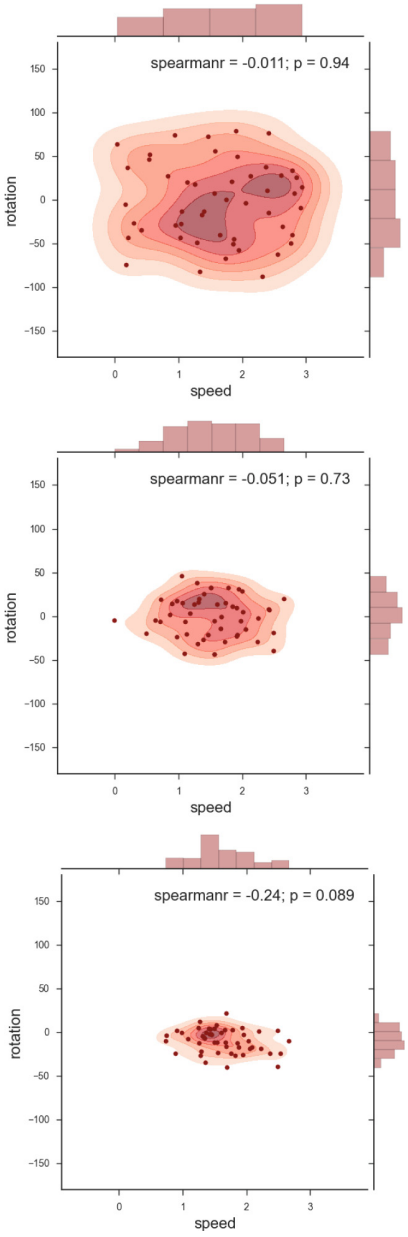


Fig. 13. Actions sampled from P_{act} for the first action to execute at iterations one, five and ten.

magnitude of effect depending on the degree of domain noise. Domain noise is given by movement speed of victims in our example. We compared victim speed of 0.1 and 1.0 (m/s). Figure 14 shows the average accumulated reward of

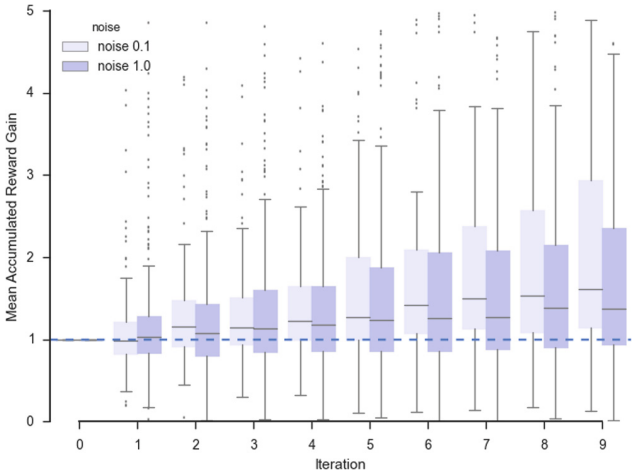


Fig. 14. Mean accumulated reward of sampled episodes per iteration. Results are shown as factor (i.e. gain) of mean accumulated reward in the initial iteration. The data shows a tendency to increase episode quality with iterative updating of the sampling strategy. The magnitude of the effect depends on domain noise. Boxes contain 50 % of measured data, whiskers 99.3 %.

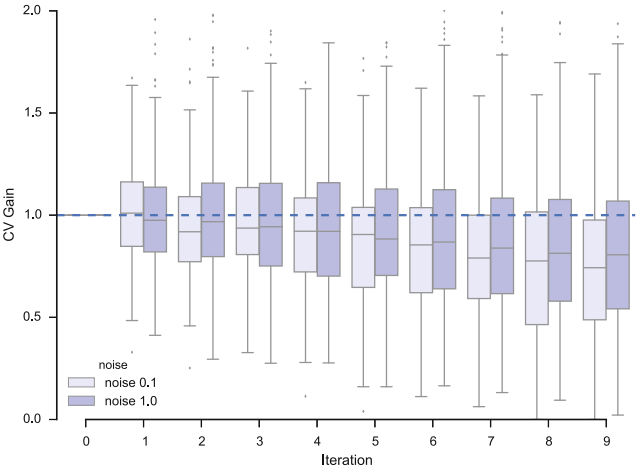


Fig. 15. Coefficient of variation (CV) of mean accumulated reward from the sampled episodes per iteration. Results are shown as factor (i.e. gain) of CV of mean accumulated reward in the initial iteration. The data shows a tendency to increase estimation accuracy with iterative updating of the sampling strategy (i.e. decreasing CV). The magnitude of the effect depends on domain noise. Boxes contain 50 % of measured data, whiskers 99.3 %.

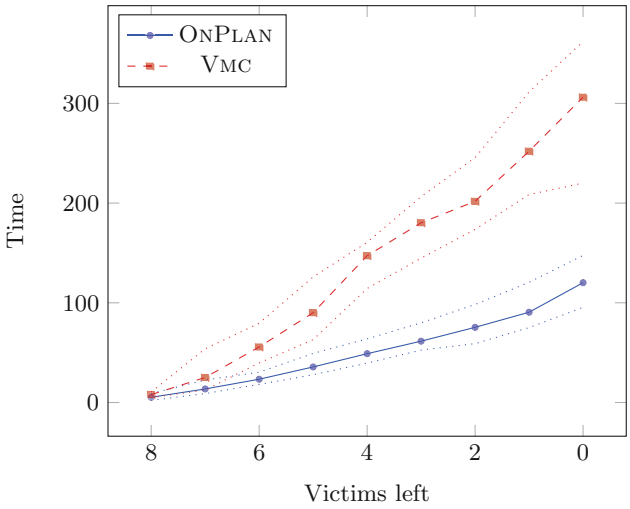


Fig. 16. Comparison of an instance of OnPlan using Ceolp with a vanilla Monte Carlo planner (Vmc). Lines show the median, dotted lines indicate interquartile range (comprising 50 % of measured data).

the episodes generated in a particular iteration, grouped by domain noise. The result is shown as a factor of the value in the initial iteration. The data shows that the episodes' average accumulated reward increases with iterations of strategy updates. The magnitude of the effect depends on domain noise. Figure 15 shows the corresponding coefficient of variation (CV), the quotient of standard deviation and mean of a sample set. This data is also grouped by domain noise. The CV of accumulated reward per episode shows a tendency to be reduced with iterations. This means that the estimation of the target value (accumulated reward per episode) is likely to increase its accuracy due to iterative strategy refinement. Again, the magnitude of the effect depends on domain noise.

Comparison with Vanilla Monte Carlo. Figure 16 shows the time needed to collect the victims by the OnPlan and vanilla Monte Carlo (Vmc) planners. Both are able to autonomously synthesize behavior that leads to successful completion of their task. System autonomy is achieved in a highly dynamic continuous state-action space with infinite branching factor and despite the noisy simulation. However, the planner using our framework is collecting victims more effectively. The benefit of making efficient use of simulation data by cross entropy optimization to drive decisions about actions becomes particularly clear when only a few victims are left. In these situations, only a few combinations of actions yield goal-oriented behavior. Therefore it is valuable to identify uninformative regions of the sampling space fast in order to distribute simulations more likely towards informative and valuable regions.

5 Conclusion and Further Work

Modern application domains such as cyber-physical systems are characterized by their immense complexity and high degrees of uncertainty. This renders unfeasible classical approaches to system autonomy which compile a single solution from available information at design-time. Instead, the idea is to provide a system with a high-level representation of its capabilities and the dynamics of its environment. The system then is equipped with mechanisms that allow to compile sensible behavior according to this high-level model and information that is currently available at runtime. I.e., instead of providing a system with a single predefined behavioral routine it is given a *space* of solutions and a way to evaluate individual choices in this space. This enables systems to autonomously cope with complexity and change.

In this paper we proposed the OnPlan framework for realizing this approach. It provides simulation-based system autonomy employing online planning and importance sampling. We defined the core components for the framework and illustrated its behavioral skeleton. We showed two concrete instantiations of our framework: Monte Carlo Tree Search for domains with discrete state-action spaces, and Cross Entropy Open Loop Planning for continuous domains. We discussed how each instantiates the plug points of OnPlan. We showed the ability of our framework to enable system autonomy empirically in a search and rescue domain example.

An important direction of future work is to extend OnPlan to support learning of simulations from observations at runtime. Machine learning techniques such as probabilistic classification or regression provide potential tools to accomplish this task (see e.g. [23]). Also, other potential instantiations of the framework should be explored, such as the GOURMAND planner based on labeled real-time dynamic programming [1,24], sequential halving applied to trees (SHOT) [25,26], hierarchical optimistic optimization applied to trees (HOOT) [27] or hierarchical open-loop optimistic planning (HOLOP) [3,28]. It would also be interesting to investigate possibilities to extend specification logics such as LTL or CTL [29] with abilities for reasoning about uncertainty and solution quality. Model checking of systems acting autonomously in environments with complexity and runtime dynamics such as the domains considered in this paper provides potential for further research. Another direction of potential further research is simulation-based planning in collectives of autonomous entities that are able to form or dissolve collaborations at runtime, so-called ensembles [30,31]. Here, the importance sampling approach may provide even more effectiveness as in a single-agent context, as the search space typically grows exponentially in the number of agents involved. Mathematically identifying information that is relevant in a particular ensemble could provide a principled way to counter this combinatorial explosion of the search space in multi-agent settings.

Acknowledgements. The authors thank Andrea Vandin for his help with the MultiVeStA statistical model checker [17].

References

1. Kolobov, A., Dai, P., Mausam, M., Weld, D.S.: Reverse iterative deepening for finite-horizon MDPS with large branching factors. In: Proceedings of the 22nd International Conference on Automated Planning and Scheduling, ICAPS (2012)
2. Keller, T., Helmert, M.: Trial-based Heuristic Tree Search for Finite Horizon MDPs. In: Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013), pp. 135–143. AAAI Press, June 2013
3. Weinstein, A.: Local Planning for Continuous Markov Decision Processes. Ph.D. thesis, Rutgers, The State University of New Jersey (2014)
4. Kephart, J.: An architectural blueprint for autonomic computing. IBM (2003)
5. Hastings, W.K.: Monte Carlo sampling methods using Markov chains and their applications. Biometrika **57**(1), 97–109 (1970)
6. Rubinstein, R.Y., Kroese, D.P.: Simulation and the Monte Carlo Method, vol. 707. Wiley, New York (2011)
7. Rubinstein, R.Y., Kroese, D.P.: The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning. Springer Science & Business Media, New York (2013)
8. Audibert, J.Y., Munos, R., Szepesvári, C.: Exploration-exploitation tradeoff using variance estimates in multi-armed bandits. Theor. Comput. Sci. **410**(19), 1876–1902 (2009)
9. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Trans. Comput. Intell. AI Game **4**(1), 1–43 (2012)
10. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
11. Weinstein, A., Littman, M.L.: Open-loop planning in large-scale stochastic domains. In: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (2013)
12. Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., Teytaud, O.: The grand challenge of computer go: Monte carlo tree search and extensions. Commun. ACM **55**(3), 106–113 (2012)
13. Silver, D., Sutton, R.S., Müller, M.: Temporal-difference search in computer go. In: Borrajo, D., Kambhampati, S., Oddi, A., Fratini, S. (eds.) Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10–14, 2013. AAAI (2013)
14. Gelly, S., Silver, D.: Monte-carlo tree search and rapid action value estimation in computer go. Artif. Intell. **175**(11), 1856–1875 (2011)
15. Bubeck, S., Cesa-Bianchi, N.: Regret analysis of stochastic and nonstochastic multi-armed bandit problems. Found. Trends Mach. Learn. **5**(1), 1–122 (2012)
16. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Mach. Learn. **47**(2–3), 235–256 (2002)
17. Sebastio, S., Vandin, A.: Multivesta: Statistical model checking for discrete event simulators. In: Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 310–315 (2013)
18. de Boer, P., Kroese, D.P., Mannor, S., Rubinstein, R.Y.: A tutorial on the cross-entropy method. Annals OR **134**(1), 19–67 (2005)

19. Margolin, L.: On the convergence of the cross-entropy method. Ann. Oper. Res. **134**(1), 201–214 (2005)
20. Kobilarov, M.: Cross-entropy motion planning. I. J. Robotic Res. **31**(7), 855–871 (2012)
21. Livingston, S.C., Wolff, E.M., Murray, R.M.: Cross-entropy temporal logic motion planning. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC 2015, pp. 269–278 (2015)
22. Box, G.E., Muller, M.E.: A note on the generation of random normal deviates. Ann. Math. Stat. **29**, 610–611 (1958)
23. Hester, T., Stone, P.: Texplore: real-time sample-efficient reinforcement learning for robots. Mach. Learn. **90**(3), 385–429 (2013)
24. Bonet, B., Geffner, H.: Labeled RTDP: Improving the convergence of real-time dynamic programming. In: ICAPS, vol. 3, pp. 12–21 (2003)
25. Karnin, Z., Koren, T., Somekh, O.: Almost optimal exploration in multi-armed bandits. In: Proceedings of the 30th International Conference on Machine Learning (ICML-13), pp. 1238–1246 (2013)
26. Cazenave, T., Pepels, T., Winands, M.H.M., Lanctot, M.: Minimizing simple and cumulative regret in monte-carlo tree search. In: Cazenave, T., Winands, M.H.M., Björnsson, Y. (eds.) CGW 2014. CCIS, vol. 504, pp. 1–15. Springer, Heidelberg (2014)
27. Mansley, C.R., Weinstein, A., Littman, M.L.: Sample-based planning for continuous action markov decision processes. In: Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS (2011)
28. Weinstein, A., Littman, M.L.: Bandit-based planning and learning in continuous-action markov decision processes. In: Proceedings of the 22nd International Conference on Automated Planning and Scheduling, ICAPS (2012)
29. Baier, C., Katoen, J.P., et al.: Principles of Model Checking, vol. 26202649. MIT Press, Cambridge (2008)
30. Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.): Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project. LNCS, vol. 8998. Springer, Heidelberg (2015)
31. Hölzl, M.M., Gabor, T.: Continuous collaboration: A case study on the development of an adaptive cyber-physical system. In: 1st IEEE/ACM International Workshop on Software Engineering for Smart Cyber-Physical Systems, SEsCPS 2015, pp. 19–25 (2015)