Combining Static and Runtime Methods to Achieve Safe Standing-Up for Humanoid Robots

Francesco Leofante[1], Simone Vuotto[1,2], Erika Ábrahám[2], Armando Tacchella[1(✉)], and Nils Jansen[3]

[1] University of Genoa, Genoa, Italy
armando.tacchella@unige.it
[2] RWTH Aachen University, Aachen, Germany
[3] University of Texas at Austin, Austin, USA

Abstract. Due to its complexity, the standing-up task for robots is highly challenging, and often implemented by scripting the strategy that the robot should execute per hand. In this paper we aim at improving the approach of a scripted stand-up strategy by making it more stable and safe. To achieve this aim, we apply both static and runtime methods by integrating reinforcement learning, static analysis and runtime monitoring techniques.

1 Introduction

Bipedal locomotion is a challenging task for a humanoid robot. In particular, in the case of a fall, it is essential for the overall robustness to have reliable recovery procedures, *i.e.*, the robot must be able to get back into an upright posture. However it is not trivial to come up with reliable *standing-up routines*. This is because standing-up requires that the robot's center of mass (COM) projection to the ground leaves the convex hull spanned by the feet contact points. Therefore, knees, elbows, hands, and the backside of the robot should be used to provide additional support. As mentioned, *e.g.*, in [1], this results in whole-body motions with sequences of support points. The many degrees of freedom of humanoid robots and the changing contact points make it difficult to apply conventional motion-planning techniques.

Given the intrinsic difficulties of the standing-up task, observation of the human example may well serve as inspiration for the development of adequate motion sequences. However, compared to humans, humanoid robots often lack essential degrees of freedom, *e.g.*, in the trunk. Furthermore, the robot joints are often restricted to a limited range of motion and can only provide limited torques. The authors of [1] developed standing-up routines using a physics-based simulation and implemented such routines on two small humanoid robots. The approach however lacks flexibility as the routines are *scripted*, *i.e.*, predetermined command sequences are fed to the motors in an open-loop fashion. Scripted routines are based on certain assumptions about the robot and its environment. If something

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 496–514, 2016.
DOI: 10.1007/978-3-319-47166-2_34

changes in the environment or in the robot, the assumptions may get violated and the routine may fail. To improve reliability, different solutions have been proposed in which a robot *learns* how to stand up (see, *e.g.*, [2–4]). However, these works consider rather simple scenarios, and none of them considers a full humanoid as a case study.

In this paper we propose an approach to improve scripted strategies by integrating static and runtime techniques.

Contribution 1. One drawback of scripted strategies is that the result of executing a certain robot action is not unique, for several reasons (*e.g.*, due to imprecise sensors and actuators, or uncertainties in the environment). In order to improve the stability of scripted standing-up strategies under consideration of those uncertainties, we apply (model-free) *reinforcement learning* [5], *i.e.*, the robot learns how to get up on its feet by interacting with the environment, observing the effects of its actions, and trying to come up with a strategy that maximises the probability to reach the desired final state. To avoid damage on the robot, we use a *simulator* in this learning process.

Contribution 2. Reinforcement learning gives us a stable strategy, determining actions the robot should execute for standing up. There are possibilities to drive the learning towards avoiding certain *unsafe* states (*e.g.*, with critical joint values or unstable poses). However, using reinforcement learning we cannot assure that the resulting strategy will avoid such unsafe states with a given probability. In [6], we proposed a *greedy model repair* approach, based on static analysis, which can be used to repair probabilistic strategies on Markov models such that the resulting repaired strategy assures certain probabilistic safety properties. Here we apply this approach to repair the strategy computed by reinforcement learning, such that the reachability of certain unsafe states is kept below a required threshold while still achieving the desired task of standing-up.

Contribution 3. As reinforcement learning uses simulation and because our repair is model-based, the repaired strategy assures safety for the model, but this safety property does not necessarily transfer to the real system. To maintain safety during operation, we propose the additional integration of *runtime monitoring* to observe the real-time behaviour. If the difference between the observed real-time behaviour and the model behaviour is too large, we use a feedback loop to the static methods to adapt the model and the strategy.

The rest of the paper is structured as follows: In Sect. 2 we recall some preliminaries. In Sect. 3 we introduce the standing-up task. We explain our approach to solve this task and present experimental results in Sect. 4. We conclude the paper in Sect. 5.

2 Preliminaries

Probabilistic Models. Here we introduce discrete-time Markov models we use as basic modeling formalism for probabilistic systems.

Definition 1 (DTMC). *A* discrete-time Markov chain (DTMC) *is a tuple* $\mathcal{D} = (S, s^{init}, P)$ *of a finite non-empty set* S *of* states*, an* initial state $s^{init} \in S$*, and a* transition probability function $P\colon S \times S \to \mathbb{R}$*, such that* $\sum_{s' \in S} P(s, s') = 1$ *for all* $s \in S$*.*

A *path* of a DTMC $\mathcal{D}$ is a non-empty (finite or infinite) sequence $s_0 s_1 \dots$ of states $s_i \in S$ such that $P(s_i, s_{i+1}) > 0$ for all i. A unique probability measure $Pr^{\mathcal{D}}$ on sets of paths is defined via the usual cylinder set construction, see [7]. Notably, the cylinder set of a finite path $s_0 \dots s_n$ (*i.e.*, the set of all infinite paths with prefix $s_0 \dots s_n$) has the total probability $Pr^{\mathcal{D}}(s_0 \dots s_n) = \Pi_{i=0}^{n-1} P(s_i, s_{i+1})$. We use $Pr_s^{\mathcal{D}}(\Diamond B)$ to denote the total probability of all paths of $\mathcal{D}$ that start in s and visit at least one state from a target set $B \subseteq S$.

Sometimes it is advantageous to use *parametric* DTMC models, where the parameters can represent, *e.g.*, design parameters, whose values should be fixed later. Let $Var = \{x_1, \dots, x_n\}$ be a finite set of real-valued *parameters* x_i with parameter domains $\mathbb{D}_i \subseteq \mathbb{R}$, and let $Val \subseteq \{v : Var \to \cup_{i=1}^{n} \mathbb{D}_i\}$ be the set of all *valuations* v for Var that assign to each $x_i \in Var$ a value from its domain $v(x_i) \in \mathbb{D}_i$. Let furthermore Exp_{Var} be a set of arithmetic expressions over Var, such that each $e \in Exp_{Var}$ can be evaluated to a real value $v(e) \in \mathbb{R}$ in the context of a valuation $v \in Val_{Var}$; in this work we use linear arithmetic expressions, but in general one could also consider non-linear expressions or rational functions [8,9]. For $e \in Exp_{Var}$ we define $Var(e) \subseteq Var$ to be the set of all parameters that occur in e. We write $e \equiv 0$ if $v(e) = 0$ for each valuation $v \in Val$, and $e \not\equiv 0$ otherwise (*e.g.*, $x - x \equiv 0$ but $x - y \not\equiv 0$). Sometimes we skip the index Var if it is clear from the context.

Definition 2 (pDTMC). *A* parametric discrete-time Markov chain (pDTMC) *is a tuple* $\mathcal{P} = (S, s^{init}, Var, P)$ *of a finite non-empty set* S *of* states*, an* initial state $s^{init} \in S$*, a finite set* Var *of* parameters*, and a (parametric)* transition probability function $P\colon S \times S \to Exp_{Var}$*. A valuation* $v \in Val_{Var}$ *is* realisable *for* $\mathcal{P}$ *if* $\sum_{s' \in S} v(P(s, s')) = 1$ *for all* $s \in S$*. A pDTMC is called* realisable *if it has at least one realisable valuation.*

Note that each realisable valuation v of a pDTMC $\mathcal{P} = (S, s^{init}, Var, P)$ *induces* a DTMC $\mathcal{D} = (S, s^{init}, P_v)$ with $P_v(s, s') = v(P(s, s'))$ for all $s, s' \in S$. In the following we consider only realisable pDTMCs, and require that each variable is used to specify successor probabilities for at most one state:

$$\forall s_1, s_2, s_1', s_2' \in S.\ Var(P(s_1, s_2)) \cap Var(P(s_1', s_2')) \neq \emptyset \to s_1 = s_1'\,. \tag{1}$$

Probabilistic systems with non-deterministic behaviour can be modelled by Markov decision processes.

Definition 3 (MDP). *A* Markov decision process (MDP) *is a tuple* $\mathcal{M} = (S, s^{init}, Act, P)$ *of a finite non-empty set* S *of* states*, an* initial state $s^{init} \in S$*, a finite non-empty set* Act *of* actions*, and a* transition probability function $P\colon S \times Act \times S \to [0, 1]$ *such that* $\sum_{s' \in S} P(s, a, s') \in \{0, 1\}$ *for all* $(s, a) \in S \times Act$*, and for all* $s \in S$ *there exists at least one* $a \in Act$ *with* $\sum_{s' \in S} P(s, a, s') = 1$*.*

For a given $s \in S$ let Act_s denote the set $\{a \in Act \mid \sum_{s' \in S} P(s,a,s') = 1\}$ of actions that are *enabled* in s. Semantically, in each state, first an enabled action is determined non-deterministically and then a successor state is selected probabilistically. Thus a *path* of the MDP $\mathcal{M} = (S, s^{init}, Act, P)$ is a non-empty (finite or infinite) sequence $s_0 a_0 s_1 a_1 \ldots$ of states $s_i \in S$ and actions $a_i \in Act_{s_i}$ such that $P(s_i, a_i, s_{i+1}) > 0$ for all i.

We use *memoryless strategies* (also referred to as *schedulers* or *policies*) to resolve the non-determinism by defining for each state $s \in S$ a probability distribution over its enabled actions, *i.e.* a function $\sigma \colon S \times Act \to [0,1]$ such that $\sum_{a \in Act_s} \sigma(s,a) = 1$ and $\sigma(s,a) = 0$ for all $a \in Act \setminus Act_s$. If the strategy assigns in each state probability one to a single action, it is called *deterministic*. Each strategy σ for an MDP $\mathcal{M} = (S, s^{init}, Act, P)$ induces an DTMC $\mathcal{D}_\sigma = (S, s^{init}, P_\sigma)$ with $P_\sigma(s,s') = \sum_{a \in Act} \sigma(s,a) \cdot P(s,a,s')$ for all $s, s' \in S$.

We also need the notion of *rewards*, which can be used to model, *e.g.*, the costs of executing certain actions. A *reward function* is given by $R \colon S \times Act \times S \to \mathbb{R}$. For MDP $\mathcal{M}$, reward function R, and strategy σ, the *expected discounted total reward* in state s is the expected value of $\sum_{i=0}^{\infty} \gamma^i \cdot R(s_i, a_i, s_{i+1})$ along paths $s_0 a_0 s_1 a_1 \ldots$ starting in $s = s_0$ under the strategy σ, where $\gamma \in \mathbb{R}$, $0 < \gamma < 1$ is the *discount factor*. An optimal strategy σ^* maximising the expected discounted total reward can be computed by solving the equation system

$$\forall s \in S.\ \forall a \in Act.\ Q^*_{s,a} \ = \ \sum_{s' \in S} P(s,a,s') \left(R(s,a,s') + \gamma \cdot \max_{a' \in Act_{s'}} Q^*_{s',a'} \right) \quad (2)$$

and choosing in each state $s \in S$ the action $a^*_s = \mathrm{argmax}_{a \in Act_s} Q^*_{s,a}$ with probability $\sigma^*(s, a^*_s) = 1$, and defining $\sigma^*(s,a) = 0$ for all other actions $a \in Act \setminus \{a^*_s\}$. Intuitively, $Q^*_{s,a}$ is the expected discounted total reward of paths starting in s, executing a first, and following an optimal policy afterwards.

Reinforcement Learning. A general class of algorithms from machine learning called *reinforcement learning* [10] lets an agent learn such an optimal strategy σ^* for a non-deterministic probabilistic system. We consider a reinforcement learning algorithm called *Q-learning* [10]. This approach takes as input a set S of states and a set Act_s of actions for each state $s \in S$. Furthermore, it needs to observe (based on a simulator or the execution of a real system) the successor state and the reward achieved when executing an action $a \in Act$ in a state $s \in S$. Based on iterative observations, Q-learning maintains a *quantity-matrix* (*Q-matrix*) of dimension $|S| \times |Act|$, with the goal to approximate the values $Q^*_{s,a}$ by the entries $Q(s,a)$ (see Eq. 2).

Starting with an arbitrarily initialised Q-matrix, each *episode* observes a path of the system: starting from the initial state $s^{init} = s_0$, it selects some enabled action $a_0 \in Act_{s_0}$, and observes the successor state s_1 and the reward $r_0 = R(s_0, a_0, s_1)$ for this execution; we write (s_0, a_0, s_1, r_0) for this observation. This is continued iteratively until some condition becomes true (*e.g.*, some final state is reached); in this case a new episode starts. For each observation (s_i, a_i, s_{i+1}, r_i) in each episode, the Q-matrix-value $Q(s_i, a_i)$ is updated according to

$$Q_{k+1}(s_i, a_i) := (1-\alpha)Q_k(s_i, a_i) + \alpha\left(r_i + \gamma \max_{a \in Act_{s_{i+1}}} Q_k(s_{i+1}, a)\right) \quad (3)$$

where $0 < \alpha < 1$ is the *learning rate* (which might dependent on k, s_i, and a_i).

We say that Q-learning is an *off-policy* algorithm, meaning that it learns an optimal strategy no matter what the agent does, as long as it explores enough. If the algorithm terminates, after a sufficiently large number of episodes, a strategy can be derived from the Q-matrix by specifying either a deterministic strategy taking in each state s the action a_s with the highest expected discounted total reward with probability 1 (*i.e.*, $\sigma(s, a_s) = 1$ for $a_s = argmax_{a \in Act_s} Q(s, a)$ and $\sigma(s, a) = 0$ for all other actions $a \in Act \setminus \{a_s\}$), or a strategy that defines a distribution over a set $Act'_s \subseteq Act_s$ of actions with the highest estimated rewards, *e.g.*, $\sigma(s, a) = e^{Q(s,a)/c^{temp}} / \sum_{a' \in Act'_s} e^{Q(s,a')/c^{temp}}$ for all $s \in S$ and $a \in Act'_s$ and $\sigma(s, a) = 0$ otherwise, where c^{temp} is the *temperature* parameter which moves the selection strategy from purely random (high c^{temp}) to fully greedy (low c^{temp}).

Model Repair. We will also make use of the following approach to adapt schedulers to satisfy certain safety requirements. Assume a parametric DTMC model with fixed variable domains, an initial (realisable) parameter valuation v_0 inducing a DTMC $\mathcal{D}_0$, a set B of unsafe states, and an upper bound $\lambda \in [0, 1]$ on the probability $Pr^{\mathcal{D}_0}_{s^{init}}(\Diamond B)$ to reach unsafe states from the initial state s^{init} in $\mathcal{D}_0$. Assume furthermore that $Pr^{\mathcal{D}_0}_{s^{init}}(\Diamond B) > \lambda$. Our goal is to modify the initial valuation such that the resulting valuation satisfies the probability bound.

Some available approaches for this task are based on non-linear programming [11] or on approximative methods [12]. Statistical model checking combined with reinforcement learning was used in [13] for a related problem on robustness. In this work we use a *greedy model repair* approach [6]. This approach considers the DTMC $\mathcal{D}_0$ induced by the initial parameter valuation v_0 and uses efficient probabilistic model checkers like `PRISM` [9] or `MRMC` [14] to compute for each state s the probability $Pr^{\mathcal{D}_0}_s(\Diamond B)$ of reaching unsafe states from s. If this probability for the initial state is above the allowed bound, we heuristically select states and try to repair their probability distributions by changing the parameter values within their domains. These *local repair steps* successively reduce the probability of reaching an unsafe state from s^{init} until its value becomes lower than the required threshold λ.

The basic idea is illustrated in Fig. 1. Assume a valuation v_k and a DTMC $\mathcal{D}_k$ induced by it. Using model checking we know for each state s the probability $Pr^{\mathcal{D}_k}_s(\Diamond B)$ to reach unsafe states from s in $\mathcal{D}_k$. The higher this probability, the more "dangerous" it is to visit this state.

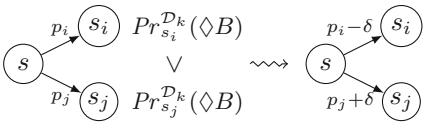

Fig. 1. The idea of greedy model repair

To repair the model, we iteratively consider single probability distributions in isolation, and modify the parameter values such that we decrease the probability to move to more "dangerous" successor states. Our approach is *sound and complete* for the considered model class: each local repair step improves the

reachability probability towards a desired bound for a repairable pDTMC, and under the condition stated in Eq. 1, the repair algorithm always terminates with a satisfying solution (if one exists).

3 The Standing-Up Task

Our study is based on the humanoid robot Bioloid by ROBOTIS [15]. The robot provides all the features needed to make the standing-up task and the associated learning problem non-trivial. In particular, Bioloid has 18 degrees of freedom (DOF) and can stand up without exploiting dynamics from both prone and supine posture due to its powerful Dynamixel AX-12A actuators [16].

In our work we use a simulated model of such a robot [17]. We use the robot simulator `V-REP` [18] (see Fig. 2), which allows the creation of fully customisable simulations which can be controlled through an API. To study real-world physics and object interaction, `V-REP` includes four different physics engines which may give slightly different results and have different performances. Currently, our experiments are carried out using Open Dynamics Engine (http://www.ode.org).

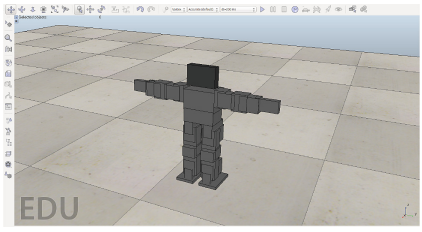

Fig. 2. The simulated Bioloid robot

In Sect. 4.1 below we describe how we model the robot in its environment by a Markov model with rewards, where the transition probability function is not known a priori (but observable via experiments). We refer to the robot's states as *poses*. To solve the *standing-up task*, we seek for an optimal strategy maximising the expected discounted total reward in the Markov model. Due to our definition of the reward function, the solution is a strategy which leads to the stand-up pose (optimally with probability 1) and minimises the expected number of falls, self-collisions and the expected number of actions required to achieve the stand-up pose. We also define a variant, the *safe standing-up task*, wherein the probability of falls and self-collisions (in the model) should be *provably* below some upper bound. It should be noted that solving the standing-up task does not necessarily require explicit knowledge about the transition probability function, whereas solving the safe standing-up task requires it to be made explicit.

4 A Novel Approach for Solving the Standing-Up Task

First we discuss the global structure of our approach, illustrated in Fig. 3, to solve the standing-up task in a stable and safe way. We assume that the robot initially lies on the ground in pose s^{init}, and it should be brought to the stand-up pose s^{goal}. We assume furthermore that the robot can observe its state, and that it has a safe restart strategy available that brings it to its initial pose in a safe way. Finally, as input we assume a scripted action sequence $A = (a_0, \ldots, a_k)$

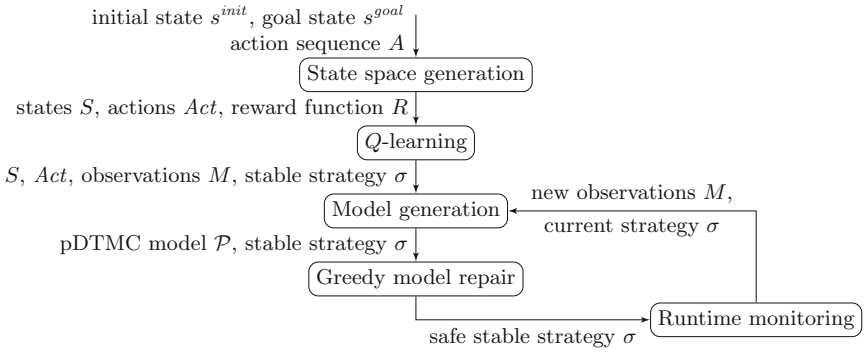

Fig. 3. The framework of our approach to solve the standing-up task

that is expected to bring the robot from its initial state s^{init} to the stand-up pose s^{goal} via the execution of the actions $a_0, \ldots, a_k$ (see, *e.g.*, [1] on how to generate such a path).

Based on the scripted input path, the first module *"State space generation"* in our approach determines a portion of the state space and a finite subset of all possible actions; the following modules will restrict their search to these state and action sets. Additionally, this module encodes the goal of standing-up by a reward function.

These specifications serve as input for the second component "$Q-learning$", which applies reinforcement learning to compute a *stable* strategy σ for standing-up: the strategy σ leads to the standing-up pose not only via the scripted path A, but it offers further alternatives such that the robot will be able to stand up even if due to some changes in the robot or in the environment A does not lead to the goal any more.

Based on observations M made about the robot's behaviour during $Q-learning$, the module *"Model generation"* builds a formal pDTMC model $\mathcal{P}$ of the robot's behaviour.

Reinforcement learning aimed at standing-up via a minimal (expected) number of actions and a minimal (expected) number of falls and self-collisions. However, as falling and self-colliding can break the robot, we want the probability of falling or self-colliding to be below a certain pre-defined threshold. The *"Greedy model repair"* component adapts the previously learnt strategy to be provably *safe*, regarding the above safety requirement (on the model).

The resulting safe stable strategy can now be applied on the real robot. To account for behavioural differences between the real robot and its model, the last component, *"Runtime monitoring"*, observes the robot's behaviour during execution, and adds a feedback loop to adapt the model and the strategy if remarkable differences between the model and the real robot are observed.

Our approach combines the strengths of static approximative methods, static formal verification and runtime monitoring to achieve a stable and safe solution for the standing-up task: (1) Reinforcement learning proved its value for the

model-free computation of stable strategies, however, it cannot give any formal guarantees. (2) Greedy model repair, based on probabilistic model checking, adapts the output of reinforcement learning if needed to satisfy certain probabilistic safety requirements. (3) Finally, runtime monitoring provides feedback to bridge differences between the model and reality. In the following we describe each of the components in more detail.

4.1 Component 1: State Space Generation

Our modelling is based on the following assumptions:

Discrete Time. Starting from a known initial state, the interaction between the robot and the environment is represented as a discrete sequence of alternating (*i*) choice of an action and its execution, and (*ii*) observation of the next state and other feedback signals from the environment. It is assumed that the actions are fully accomplished before observing the next state, *i.e.*, the observation occurs once transient dynamics are over.

Time-Invariance. The properties of the environment and of the robot do not vary over time. Variations intervening at a later time can be accommodated by the feedback loop in our method (see Sect. 4.5).

Probabilistic effects. Actions have probabilistic effects, *i.e.*, given a state s and an action a, a probability distribution $P(s, a, \cdot)$ governs the set of potential next states. We assume that $P(s, a, \cdot)$ is not known, *i.e.*, no model describing the effects of actions is given.

Markov Property. The effect of an action depends only on the action and the state in which it is executed, but not on past executions.

Next we describe how we compute the action space, the state space and the reward function, which will be used by the other modules.

Action Space. An action of the robot can change all of its 18 joint angles within their physical limits. To achieve a manageable, but for our purposes still sufficient action space, we apply the following reductions. Firstly, some joints are assumed to be inhibited, *i.e.*, they cannot be used to stand up. This is the case of, *e.g.*, the ankle joints that make feet roll. Secondly, joints are always operated symmetrically, *i.e.*, if the left hip joint is moved so is the right one. This gives us a six-dimensional action space, three dimensions corresponding to pairs of joints in the arms, and three dimensions corresponding to pairs of joints in the legs. Thirdly, we make the action space discrete by allowing actions to either increment (1) or decrement (-1) each of the joint angles by a fixed amount Δ^{act}, or leave them unchanged (0), resulting in the action space $Act = \{-1, 0, 1\}^6 \cup \{a^{restart}\}$, where $a^{restart}$ indicates safe restart, leading back to the initial state. We will use the notation a^{skip} for the "do nothing" action $(0, 0, 0, 0, 0, 0)$, causing no state change. We assume that every action is enabled in each state (if the execution of an action is physically not possible, it will be reflected in the successor states), and that Δ^{act} is chosen such that each action in A can be realised by a sequence

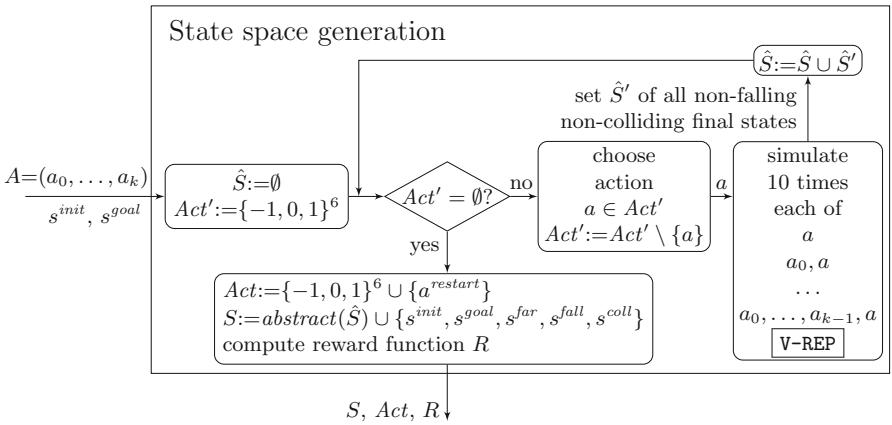

Fig. 4. The state space generation framework

of actions from Act. For simplicity, in the following we consider A to be refined accordingly, *i.e.*, we assume that the actions in A are in Act.

State Space. The state of the Bioloid, *i.e.*, its pose in the three-dimensional space, can be specified by a vector

$$\mathbf{s} = (x, y, z, q_0, q_1, q_2, q_3, \rho_1, \ldots, \rho_{18}) \in \mathbb{R}^{25} \ ,$$

where (x, y, z) are the COM coordinates and (q_0, q_1, q_2, q_3) are the quaternions defining the orientation of the torso according to the absolute coordinate system of the simulator; $(\rho_1, \ldots, \rho_{18})$ are the 18 joint angles corresponding to the 18 DOFs of the robot.

Due to physical constraints, the state space is bounded but infinite. To enable learning and the application of formal methods, we need to restrict ourselves to a finite subset S of the states. Standard grid-based discretisation is not applicable in our case as it would result in a finite but extremely large state set: if we would consider just two discrete points in each dimension, we would end up with 2^{25} states. Instead, we base our discretisation on the input action sequence $A = (a_0, \ldots, a_k)$ that leads the robot from its initial state s^{init} to its *stand-up pose* s^{goal} via a sequence of statically stable poses. Our aim is to keep the state space at a manageable size, but to cover not only the states reachable from s^{init} along A but include also a "tube" around those paths to be able to represent also further standing-up paths that have similarities with A but which are not identical to it.

The state space generation is illustrated in Fig. 4. Using `V-REP` and s^{init} as initial pose, for each action $a \in Act \setminus \{a^{restart}\}$ we simulate 10 times each action sequence of the form $a_0, \ldots, a_i, a$, where $a_0, \ldots, a_i$ is a (possibly empty) prefix of A, and collect in the set $\hat{S}$ the non-falling and non-colliding final states of all simulations. Next we build an abstraction of the state set $\hat{S}$ by determining the smallest box containing all states in $\hat{S}$, putting a grid on it, and picking the mid-

points of all grid elements that contain at least one point from $\hat{S}$. We extend the resulting abstracted state set S with the initial state s^{init}, the goal state s^{goal}, and the special state s^{far} representing the not included grid elements, *i.e.*, poses that are "too far" away from the poses of interest. We also add two auxiliary states s^{fall} and s^{coll} to represent falling or self-collision of the robot. In order to keep the notion of distance between poses, states are stored in k-d trees [19].

Reward Function. To locally assess the quality of an action in terms of safety and effectiveness, we quantify the immediate reward (or penalty) associated to performing a given action in a given state by a reward function $R\colon S \times A \times S \to [-c, c]$ for some $c \in \mathbb{R}$ as follows, based on some fixed $c^{goal}, c^{fall}, c^{coll}, c^{far}, c^{exec} \in \mathbb{R}^+$:

1. if a collision is detected, either with the floor (fall) or with the robot itself (self-collision), then the robot is penalised by $R(s, a, s^{fall}) = -c^{fall}$ respectively $R(s, a, s^{coll}) = -c^{coll}$;
2. the robot is rewarded by $R(s, a, s^{goal}) = c^{goal}$ when an action leads (close) to the goal state;
3. when the robot ends up in the special state s^{far} we give it a large negative reward $R(s, a, s^{far}) = -c^{far}$;
4. in all other cases, a small negative reward $R(s, a, s') = -c^{exec}$ accounts for energy consumption to perform the action.

Above we assumed that collision detection is somehow made possible in the (simulated) robot, *e.g.*, by detecting abnormal accelerations of the COM in case of a fall, or by sensing abnormal forces or torques at the limbs or the joints upon the execution of an action which cannot be fully accomplished because of a self-collision. Note furthermore that, since the reward values are bounded, the expected discounted total reward is always finite.

Experimental Results. We used an input trace A of 13 actions, where each action changes the joint angles by -30, 0 or $+30$ degrees (*i.e.*, $\Delta^{act} = 30$). The action set $Act = \{-1, 0, 1\}^6 \cup \{a^{restart}\}$ contains $729 + 1$ actions.

For the state space generation we executed $13 \cdot 729 \cdot 10 = 94770$ simulations. From these simulations, $|\hat{S}| = 60272$ led to non-falling non-colliding final states.

For the state space abstraction we divided the box spawn by those final states into grids, where the grid structure was based on $n_1 = 50$ equidistant points in each COM dimension, $n_2 = 20$ points for the torso angles, and $n_3 = 12$ for the joint angles (for the joint dimensions this corresponds to a grid distance of $\Delta^{act} = 30$ degrees). Selecting those grid elements that contained at least one of the 60272 final states and adding the special states for falling etc. gave us a state set with $|S| = 17614$ states.

To improve precision, in the future we plan to experiment with larger state spaces by simulating each path several times (instead of once), and considering extensions of prefixes of the scripted path by two actions (instead of one).

For the reward function, we use the constants $c^{goal} = 1000$, $c^{fall} = 100$, $c^{coll} = 100$, $c^{far} = 100$ and $c^{exec} = 1$.

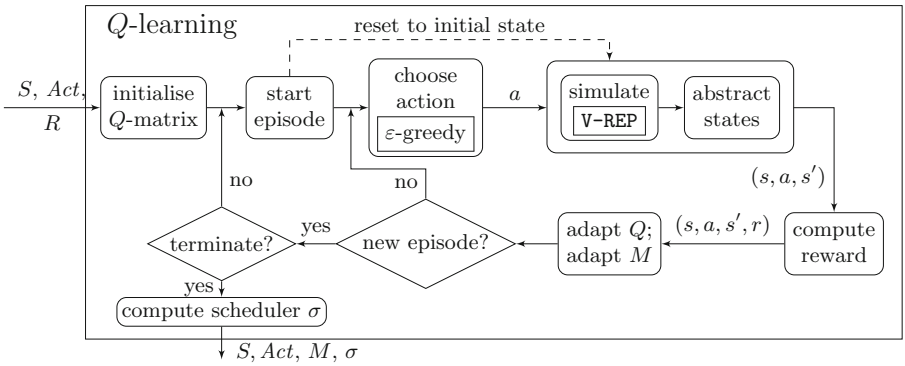

Fig. 5. The Q-learning framework

4.2 Component 2: Reinforcement Learning

In order to perform learning, we use the Q-learning [10] algorithm implemented in the `Pybrain` library [20].

The framework of our Q-learning application is illustrated in Fig. 5. We select the actions to be performed during the episodes with the $\varepsilon - greedy$ exploration strategy: based on some predefined value $\varepsilon \in \mathbb{R}^+$, the learner takes its current best action $a = \text{argmax}_{a \in Act \setminus \{a^{restart}, a^{skip}\}} Q(s, a)$ with probability $(1 - \varepsilon)$ and a randomly selected different action (according to a homogeneous distribution) from $Act \setminus \{a, a^{restart}, a^{skip}\}$ with probability ε. The value of ε decreases smoothly with each episode, in order to put stronger weight on exploration at the beginning of learning, and shift it on exploitation later [21].

We use the `V-REP` simulator for making observations $(\hat{s}, a, \hat{s}')$ about the execution of action a in state $\hat{s} \in \mathbb{R}^{25}$. The observed successor state $\hat{s}' \in \mathbb{R}^{25}$ is first abstracted to a state in $abs(\hat{s}') = s' \in S$ as follows. If we observe fall or self-collision, we map $\hat{s}'$ to s^{fall} respectively s^{coll}; otherwise, if $\hat{s}'$ lies "sufficiently near" to one of the considered states, we represent it by the "closest" state; otherwise, if it is "too far" from the defined poses of interest, we represent it by the special state s^{far}. Formally, we define a mapping $abs : \mathbb{R}^{25} \to S$ as

1. $abs(\hat{s}') = s^{fall}$ in case of falling, else
2. $abs(\hat{s}') = s^{coll}$ in case of self-collision, else
3. $abs(\hat{s}') = s^{goal}$ if $\parallel \hat{s}' - s^{goal} \parallel < d^{goal}$, else
4. $abs(\hat{s}') = s^*$ if $d^* \leq d^{far}$, and
5. $abs(\hat{s}') = s^{far}$ otherwise.

where $\parallel \cdot \parallel$ is the standard Euclidean norm, $d^{far} = \frac{1}{2} \min_{s_1, s_2 \in S} \parallel s_1 - s_2 \parallel$, $d^* = \min_{s \in S} \parallel \hat{s}' - s \parallel$, and $s^* = \text{argmin}_{s \in S} \parallel \hat{s}' - s \parallel$. Note that this mapping is efficient thanks to the k-d tree data structure we use.

Next the reward $r = R(s, a, s')$ is computed and the extended observation (s, a, s', r) is used to update the Q-matrix according to Eq. 3. Additionally, we remember that we observed (s, a, s') in a function M, which will be needed for an approximation of the probabilistic transition function (see Sect. 4.4).

More precisely, we use a function $M : S \times Act \times S \to \mathbb{N}$ with initial values $M(s, a, s') = 0$ for all $s, s' \in S$ and $a \in Act$ to represent this information; in the implementation, only entries with positive function values are stored. Each time s' was observed as the successor state of s when executing action a, we increase the value of $M(s, a, s')$ by 1.

An episode ends when the abstraction s' of the observed successor state $\hat{s}'$ is one of s^{goal}, s^{far}, s^{fall} or s^{coll}; in these cases, if a predefined termination condition of the learning algorithm is not yet satisfied, a new episode starts in the initial state. Otherwise, a next iteration starts from the successor state $\hat{s}'$.

In our implementation, the learning algorithm terminates after a fixed number of episodes; this number is experimentally determined to be sufficient for near-optimal strategies. Note that each episode terminates with probability 1.

After termination, for each state $s \in S \setminus \{s^{goal}, s^{far}, s^{fall}, s^{coll}\}$ we select a set $Act'_s \subseteq Act_s$ of actions with the highest $Q(s, \cdot)$-values. Q-learning outputs the strategy $\sigma : S \times Act \to [0, 1]$ defined by $\sigma(s^{goal}, a^{skip}) = \sigma(s^{far}, a^{restart}) = \sigma(s^{fall}, a^{restart}) = \sigma(s^{coll}, a^{restart}) = 1$,

$$\sigma(s, a) = \exp(Q(s, a)/c^{temp}) / \sum_{a' \in Act'_s} \exp(Q(s, a')/c^{temp}) \tag{4}$$

for each $s \in S \setminus \{s^{goal}, s^{far}, s^{fall}, s^{coll}\}$ and $a \in Act'_s$, and $\sigma(s, a) = 0$ else.

Experimental Results. In our experiments we used for the ε-greedy strategy the value $\varepsilon = 0.15$, and for the generation of the scheduler the temperature value $c^{temp} = 10$. We initialised all entries in the Q-matrix by the value 10. To speed up learning, we use the scripted action sequence A in the first 50 episodes. The fact that 26 different states were visited during these first 50 episodes, all simulating the same sequence of 13 actions, shows that the robot exhibits random behaviour. From the 51st episode on, we used the ε-greedy strategy.

Though we defined a relatively small state set, after some first experiments we recognised that the learning converges very slowly, the main bottleneck being the simulation (taking about half a second per action). To speed up learning, we first implemented a batch approach [22], where several episode simulations are run in parallel on multiple cores and the Q-matrix is updated only when all simulations in the batch (400 in our case) terminate. In this setting the ε value for the ε-greedy strategy was initialized to 0.15 and decremented at the end of each batch with a decay factor 0.999 until it reached 0.1 and then left fixed to this value. Also α was initialized to 0.5 and decremented with the same decay factor. γ was instead left unchanged to the initial value of 0.9.

This improvement speeded up the learning, however we observed that even when executing over 100000 episodes, most of the exploration was still done close to the initial state and relatively few exploration happened close to the goal state. To give an intuition, in 105753 episodes, the following table lists for $i = 0, \ldots, 12$ the total number of simulation steps executed in states reachable via a prefix $(a_0, \ldots, a_{i-1})$ of length i of the input trace $A = (a_0, \ldots, a_{12})$ (first row, $i = 0$ stays for the empty prefix, *i.e.*, executions in s^{init}), and splits this

number into simulation steps that used the successor action a_i in A (second row) and simulation steps that used a different action (third row):

i	0	1	2	3	4	5	6	7	8	9	10	11	12
Total	105753	69210	55543	43399	36191	31181	24635	21383	17875	15285	13174	11765	9318
a_i	52697	44481	34556	31754	29195	24397	20974	17843	15250	13036	11381	9332	8213
not a_i	53056	24729	20987	11645	6996	6784	3661	3540	2625	2249	1793	2433	1105

Therefore, we introduced a new approach that starts episodes not only in the initial state, but randomly in any state reachable by executing a (possibly empty) prefix of our input trace A. The starting state is determined by a probability distribution choosing the prefix of length i with probability $p_i = \frac{2(|A|-i)}{|A|(|A|+1)}$ for $i = 0, \ldots, k$ (where $|A| = k + 1$). Thus the probability decreases with the prefix length as follows (numbers rounded, $|A| = 13$):

p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}
0.14	0.13	0.12	0.10	0.09	0.08	0.07	0.06	0.05	0.04	0.03	0.02	0.01

This approach speeded up learning remarkably by distributing the exploration. The following table is analogous to the previous one but is based on the new approach and 166147 episodes:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
Total	23697	42862	59236	74388	81859	83676	84651	86884	83910	80263	85465	68568	61246
a_i	20731	37359	49761	60250	67399	71125	73265	73346	72485	69983	64605	59344	53778
not a_i	2966	5503	9475	14138	14460	12551	11386	13538	11425	10280	20860	9224	7468

4.3 Component 3: Model Generation

Q-learning does not rely on a concrete model but on an estimation obtained implicitly during the learning process, whereas verification requires an explicit model. In order to enable formal methods, we define a *parametric* DTMC model of the robot in its environment, where the parameters represent the scheduler choices, as follows.

We use parameters $p_{s,a}$ to represent the probabilities with which the strategy chooses action a in state s. When we instantiate the parameters to $p_{s,a} = \sigma(s, a)$, we get a DTMC model of the robot that chooses actions according to the strategy σ received from Q-learning. Exceptions are the unsafe states s^{fall}, s^{coll} and s^{far}, for which we choose the restart action $a^{restart}$ with probability 1 to bring the robot back to its initial state. Furthermore, we make the goal state s^{goal}

absorbing by taking a self-transitions (using the action a^{skip}) back to it with probability 1. Formally, we introduce a set

$$Var = \{p_{s,a} \mid s \in S \setminus \{s^{far}, s^{fall}, s^{coll}, s^{goal}\} \wedge a \in Act_s \wedge \sigma(s,a) > 0\}$$

of parameters and define an initial valuation $v_0 : Var \to (0,1]$, $v_0(p_{s,a}) = \sigma(s,a)$ according to the strategy σ determined by Q-learning. As we want to keep the graph structure of the DTMC, we define the domains of the parameters as $(0,1]$; choosing smaller domains could be used to put stronger restrictions on how far the modification may change the original scheduler.

We use the observations M to define for each state s and action a the probability distribution $\mu_{s,a} : S \to [0,1]$ which characterises the successor states when executing a in s. We define $\mu_{s,a}(s') = M(s,a,s')/M(s,a)$ with $M(s,a) = \sum_{s'' \in S} M(s,a,s'')$ if $M(s,a) > 0$. Otherwise, if $M(s,a) = 0$ then no observations were made for action a in state s, therefore we cannot predict the successors; in this case we define the far state s^{far} to be the successor with probability 1 by setting $\mu_{s,a}(s^{far}) = 1$ and $\mu_{s,a}(s') = 0$ for $s' \in S \setminus \{s^{far}\}$.

Now we can formalise the parametric DTMC model $\mathcal{P} = (S, s^{init}, Var, P)$ of the robot, where for $(s,s') \in S \times S$ we set $P(s,s')$ to

- 1 for $(s,s') \in \{(s^{goal}, s^{goal}), (s^{far}, s^{init}), (s^{coll}, s^{init}), (s^{fall}, s^{init})\}$;
- 0 for $(s = s^{goal} \wedge s' \neq s^{goal}) \vee (s \in \{s^{far}, s^{coll}, s^{fall}\} \wedge s' \neq s^{init})$;
- $\sum_{a \in Act_s, \sigma(s,a) > 0} p_{s,a} \cdot \mu_{s,a}(s')$ for $s \notin \{s^{far}, s^{fall}, s^{coll}, s^{goal}\}$.

Experimental Results. We generated the parametric DTMC as described above and instantiated it with the scheduler σ from Q-learning. To test validity, we applied model checking to compute the probabilities of reaching the unsafe states s^{fall}, s^{coll} or s^{far} from the initial state in the model, and compared these probabilities to statistical observations gained by simulation (using the same scheduler σ for 300 simulations). It is worth mentioning that the probability to reach the goal state in the model is 1, *i.e.*, the goal state is the only bottom strongly connected component in the model's graph. Especially it means also that the probability to reach the goal state without visiting unsafe states is 1 minus the probability to reach an unsafe state. The results in the following table show that the behaviour of our parametric model is close to reality under σ:

	s^{fall}	s^{coll}	s^{far}
Probability to reach in model	0.001	0.005	0.048
Probability to reach in simulation	0	0.003	0.046

For model checking we employed the `StoRM` tool[1]. Is is of course also possible to use other probabilistic model checkers like `PRISM` [9], however, the flexible API met our needs best in terms of incrementality.

[1] Yet unpublished, developed by Christian Dehnert, RWTH Aachen University, Germany.

Using graph analysis methods, the generated model also allowed us to gain some information about the action sequences that the underlying scheduler might choose to bring the robot in the stand-up pose. We observed that the model contained at least 15 such traces (we did not perform a complete search) of length at most $|A| = 13$, and even some *shorter* traces of length 12.

4.4 Component 4: Greedy Model Repair

Reinforcement learning gives us a scheduler to achieve the standing-up task, however, it does not assure any upper bounds on the probabilities of reaching unsafe states. To achieve not only stability but also safety, we instantiate the parametric DTMC with the initial valuation v_0 (corresponding to the scheduler σ received from Q-learning) and check the resulting DTMC model for safety. If yes, the induced strategy can be applied. Otherwise, we will *automatically repair* the strategy to become safe by modifying the parameter values using the greedy model repair approach [6], introduced in Sect. 2. For the kind of models we have in our application the model repair is complete, *i.e.*, it will always yield a repaired model that satisfies the safety constraints, if there exists any (Fig. 6).

More formally, given $\lambda \in (0, 1) \subseteq \mathbb{Q}$, our aim is to modify the initial parameter valuation v_0 to a valuation v such that the probability to reach unsafe states s^{fall}, s^{coll} or s^{far} from s^{init} in the DTMC $\mathcal{D}(\mathcal{P}, v)$ induced by v is at most λ. For each state s and action a, the repair potentially changes the scheduler probabilities $\sigma(s, a)$, while the support $\{a \in Act_s \mid \sigma(s, a) > 0\}$ remains unchanged, *i.e.*, it neither introduces new actions nor assigns possible actions probability. Furthermore, to keep the strategy near-optimal, we are interested in a solution "close" to the initial scheduler, *i.e.*, we want to change the distributions smoothly.

Different heuristics can be used to select the state whose distribution should be repaired or affected by the repair and to decide how strong the modifications might be. We use the following ones: Assume a current parameter valuation v.

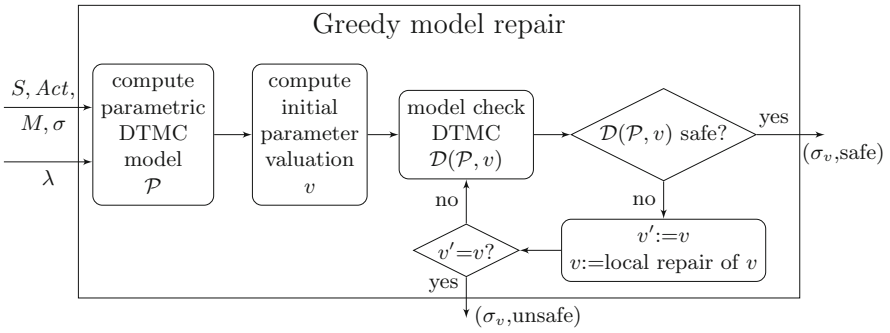

Fig. 6. The greedy model repair framework

From the model checking result we know for each state $s \in S$ the probability p_s^{unsafe} to reach one of the unsafe states from s in $\mathcal{D}(\mathcal{P}, v)$. Remember that for each state $s \in S \setminus \{s^{far}, s^{fall}, s^{coll}, s^{goal}\}$, the probability to reach a successor state s' is $P(s, s') = \sum_{a \in Act_s, \sigma(s,a)>0} v(p_{s,a}) \cdot \mu_{s,a}(s')$. For each such state s, we first determine the "safest" action $a_s^{safe} \in Act_s$, $\sigma(s, a_s^{safe}) > 0$ that minimises the probability $\sum_{s' \in S} \mu_{s,a}(s') \cdot p_{s'}^{unsafe}$ to reach unsafe states, and for each action $a \in Act_s \setminus \{a_s^{safe}\}$ with $v(p_{s,a}) > \delta$ we increase $v(p_{s,a_s^{safe}})$ by δ and decrease $v(p_{s,a})$ by δ. After we have repaired each repairable distribution, we iterate model checking and repair until the probability to reach an unsafe state from the initial state s^{init} in $\mathcal{D}(\mathcal{P}, v)$ is below $\lambda = 0.001$ (*i.e.*, below 0.1 %). To speed up the repairing process, we first select a relatively large δ and decrement it when no further repair is possible (and the safety threshold is not yet reached). In our experiments we use $\delta \in \{0.2, 0.1, 0.05, 0.01\}$ in a decreasing order.

The algorithm returns a scheduler σ_v computed from the final valuation v by setting $\sigma(s, a) = v(p_{s,a})$.

Experimental Results. We performed model repair on the model generated in the previous section, following the heuristics described above. The model repair was fast and performed well without further adaptations. After the termination of the repair process, we used 300 simulations to validate the repaired model. The results are summarised in the following table:

	s^{fall}	s^{coll}	s^{far}
Probability to reach in *repaired* model	0.0001	0.0002	0.0012
Probability to reach in simulation	0.063	0.113	0.747

The above results demonstrate that the repaired model is not valid any more. After a thorough analysis we have found out that the model was invalidated by the repair because the repair increased the probabilities of actions for which relatively few observations were available, and therefore come with high uncertainties. To solve this problem, we modified the definition of the parametric DTMC by defining higher probabilities to get to unsafe successor states if fewer observations per successor state were available. More precisely, for a state s and an action a let $\nu_{s,a} = \exp(-\frac{M(s,a)}{50 \cdot |\{s' \in S \mid M(s,a,s')>0\}|}) \cdot 100$. For each state-action pair (s, a) with $M(s, a) > 0$, instead of specifying $\mu_{s,a}(s') = M(s, a, s')/M(s, a)$, we set $\mu_{s,a}(s') = M(s, a, s')/(M(s, a) + \nu_{s,a})$ for each $s' \neq s^{far}$ and define $\mu_{s,a}(s^{far}) = (M(s, a, s^{far}) + \nu_{s,a})/(M(s, a) + \nu_{s,a})$.

This adaptation gave us the following results, which suggest that, under the above consideration of uncertainties in the model generation, the repair leads to a valid model (we used 500 simulations):

	s^{fall}	s^{coll}	s^{far}
Probability to reach in *repaired* model	0.0003	$6.8 \cdot 10^{-6}$	0.02
Probability to reach in simulation	0	0	0

4.5 Component 5: Runtime Monitoring

Once a strategy is learnt and repaired, it can be deployed on the robot. If the model is valid and neither the robot nor the environment undergo changes, the robot is safe. However, if one of these conditions is violated, the previously safe and stable strategy might fail. To account for those cases, we integrate a feedback loop via runtime monitoring to the model generation and repair stages. During deployment, we collect observations like previously done in the M matrix, and from time to time we re-compute the model according to the new observations, and repair the current scheduler (if necessary) based on the adapted model. This process helps to improve the validity of the model in case there are no changes in the robot nor its environment.

However, if either the robot or its environment changes, due to large M-matrix entries it might take very long till observations of a modified setting will have a visible effect on the computed probability distributions in the model. To account also for such cases, after each model computation, we may scale down entries of M in such a way that new observations will lead to faster adaptations.

Experimental Results. To check the adaptiveness of our approach, after completed scheduler generation, we modified the simulations such that one of the actions in the input trace A leads to self-collision. In this way we modeled that a part of the robot is broken, such that the input trace did not lead from the initial to the goal state any more. We simulated 300 episodes; only 2 of them reached the goal, 297 collided, none has fallen, and 1 episode ended in a far state.

We used the observations from these 300 episodes, adapted the model and repaired the scheduler. The repair changed the scheduler towards alternative traces to the stand-up pose: from further 300 episodes (using the repaired scheduler), 197 reached the goal (without visiting unsafe states), 83 collided, 1 has fallen and 19 ended in a far state. These results hint to a potential improvement brought by monitoring, but further experiments with more episodes will be needed for a more precise evaluation of the monitoring feedback loop.

5 Lessons Learnt

We proposed an integrated approach, combining reinforcement learning, static analysis and runtime monitoring techniques to develop a stable and safe strategy for a robot to stand-up. First experimental results gave interesting insights into both the challenges as well as the potentials of such an application. In general, we can conclude with the following observations:

- Even for highly complex systems it is possible to find a manageable subset of the state space still representing sufficient information to solve relevant problems.
- In our application, formal methods were employable without any major obstacle. The main bottleneck was the time-consuming simulation.
- The combination of reinforcement learning and static methods allowed to derive new and even shorter paths, different from the original scripted one.
- Runtime monitoring can be successfully combined with formal methods to adapt systems to changing (internal or environmental) conditions.

To witness applicability, our next steps will focus on more detailed models and learning phases. Especially, we will consider (1) the generation of larger state spaces, allowing more alternatives for the standing-up procedure, (2) further adaptations of reinforcement learning to speed up its convergence, (3) improve the repair heuristics, and (4) further increase the adaptation speed to changing conditions by improving the feedback loop via runtime monitoring.

References

1. Stückler, J., Schwenk, J., Behnke, S.: Getting back on two feet: reliable standing-up routines for a humanoid robot. In: Proceedings of the IAS-9, pp. 676–685. IOS Press (2006)
2. Morimoto, J., Doya, K.: Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. Robot. Auton. Syst. **36**(1), 37–51 (2001)
3. Morimoto, J., Doya, K.: Reinforcement learning of dynamic motor sequence: learning to stand up. In: Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 3, pp. 1721–1726 (1998)
4. Schuitema, E., Wisse, M., Ramakers, T., Jonker, P.: The design of LEO: a 2D bipedal walking robot for online autonomous reinforcement learning. In: Proceedings of the IROS 2010, pp. 3238–3243 (2010)
5. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning. MIT Press, Cambridge (1998)
6. Pathak, S., Ábrahám, E., Jansen, N., Tacchella, A., Katoen, J.-P.: A greedy approach for the efficient repair of stochastic models. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 295–309. Springer, Heidelberg (2015). doi:10.1007/978-3-319-17524-9_21
7. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
8. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. Softw. Tools Technol. Transf. **13**(1), 3–19 (2010)
9. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_47
10. van Otterlo, M., Wiering, M.: Reinforcement learning and Markov decision processes. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning, vol. 12, pp. 3–42. Springer, Heidelberg (2012)
11. Bartocci, E., Grosu, R., Katsaros, P., Ramakrishnan, C.R., Smolka, S.A.: Model repair for probabilistic systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 326–340. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19835-9_30

12. Chen, T., Hahn, E.M., Han, T., Kwiatkowska, M., Qu, H., Zhang, L.: Model repair for Markov decision processes. In: Proceedings of the TASE 2013, pp. 85–92. IEEE (2013)
13. Bartocci, E., Bortolussi, L., Nenzi, L., Sanguinetti, G.: On the robustness of temporal properties for stochastic models. In: Proceedings of the HSB 2013. EPTCS, vol. 125, pp. 3–19 (2013)
14. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. **68**(2), 90–104 (2011)
15. Bioloid premium kit. http://en.robotis.com/index/product.php?cate_code=121010. Accessed 3 July 2016
16. Dynamixel actuators. http://en.robotis.com/index/product.php?cate_code=101010. Accessed 3 July 2016
17. Bioloid URDF model. https://github.com/dxydas/ros-bioloid. Accessed 3 July 2016
18. Rohmer, E., Singh, S.P.N., Freese, M.: V-REP: a versatile and scalable robot simulation framework. In: Proceedings of the IROS 2013, pp. 1321–1326 (2013)
19. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
20. Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T., Schmidhuber, J.: PyBrain. J. Mach. Learn. Res. **11**, 743–746 (2010)
21. Defazio, A., Graepel, T.: A comparison of learning algorithms on the arcade learning environment. arXiv preprint arXiv:1410.8620 (2014)
22. Lange, S., Gabel, T., Riedmiller, M.: Batch reinforcement learning. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning, pp. 45–73. Springer, Heidelberg (2012)