# A Novel Robot-Task-Description for a Variety of Dynamic Behaviors

Akihiko Yamaguchi and Tsukasa Ogasawara

***Abstract*— This paper aims to unify the "robot language" approach and the reinforcement learning (RL) framework in order to design behaviors of robots with a simple description. We develop a kind of robot language where we describe a robot task, then the robot employs an RL method to acquire the corresponding behavior. The remarkable feature of this approach is that we do not have to specify the procedure of the behavior, and the models of the environment and the robot. To accomplish this approach, we employ the C++ RL library SkyAI as the base system, then we extend the SkyAI's script interface so that we can describe tasks simply. In this mechanism, a task is described with several event-driven functions where the reward and the end-of-episode condition are defined. As the demonstration, we design six kinds of behaviors for a humanoid robot; a crawling, a handstanding, a jumping, a forward rolling, a backward rolling, and a turning task.**

## I. Introduction

In the early stage of the robotics research, "robot languages" are developed in order to flexibly describe tasks (e.g. [1]). In such a robot language, the models of the robot and the environment, and the procedure of a task should be described, which is difficult for non-expert users. In addition, it is difficult to use such a language for describing dynamic behaviors, such as crawling and jumping.

On the other hand, reinforcement learning (RL) is a promising tool to design the behaviors of robots, including dynamic behaviors [2], [3], [4], [5]. Using RL methods, we can design a behavior with a reward function that encodes the task objective, which may be easier and more intuitive than describing the task procedure. In addition, we need not to model an environment and the robot explicitly. The most successful way to design the task is based on imitation or teaching by demonstration (e.g. [6]). However, such an approach is applicable only for the tasks that a human can demonstrate.

This paper aims to unify the both approach; we develop a kind of robot language where we describe a task, but we need not to specify the models and the procedure. We employ an RL method to enable the robot to acquire behaviors without the models and the procedure (i.e. the policy). In this paper, we focus on developing a task description mechanism in order to tell the purpose of task to a robot.

Specifically, we employ the C++ RL library SkyAI [7] as the base system, and extend its script interface so that we can describe the task objective simply. In this mechanism, a task can be described as the reward description and the end-of-episode condition. Each of them is evaluated in several *event-driven functions* such as each time-step and each end-of-action. According to a given description of the task, the robot evaluates its behavior and optimizes it by using an RL algorithm. Thus, using this mechanism, the user can design a behavior only by describing its task setup.

A. Yamaguchi and T. Ogasawara are with Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5, Takayama, Ikoma, Nara 630-0192, JAPAN {akihiko-y, ogasawar}@is.naist.jp

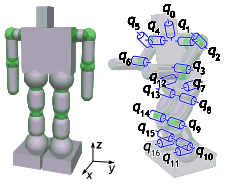



Fig. 1. Simulation model of a humanoid robot.

In order to demonstrate that the proposed description mechanism can design a variety of dynamic behaviors, we design six kinds of behaviors for a humanoid robot; a crawling, a handstanding, a jumping, a forward rolling, a backward rolling, and a turning task. The experimental results show that the robot acquires behaviors as we have expected. We also compare the local maxima of behaviors.

This paper is organized as follows: Section II proposes the task description mechanism. Section III describes how to realize the proposed task description with SkyAI. Section IV demonstrates the experimental results. Section V discusses the task description and the acquired behaviors. Section VI concludes this paper.

## II. Task Description Mechanism

Using the RL framework, we can design a task by a reward function and an end-of-episode condition, rather than specifying the policy. In this section, we discuss how to describe such task information. First, let us consider two examples: a crawling task and a jumping task. Then, we make clear how to describe tasks in a general manner. At the end of this section, we discuss a way to realize a robot system that accomplishes the task descriptions.

### A. Task Examples

We employ a humanoid robot since the robot can perform a variety of motions. Fig. 1 shows the simulation model used in the following experiments.

The robot has 18 links and 17 joints. The sensor input of the robot consists of the global position $(c_{0x}, c_{0y}, c_{0z})$ (the center-of-mass of the body link), the global orientation in quaternion $(q_w, q_x, q_y, q_z)$, their velocities $(\dot{c}_{0x}, \dot{c}_{0y}, \dot{c}_{0z}, \omega_x, \omega_y, \omega_z)$, the joint angles $(q_0, \ldots, q_{16})$, the joint angular velocities $(\dot{q}_0, \ldots, \dot{q}_{16})$, and the contact-with-ground flags (0 or 1) of links $(c_{\mathrm{g}0}, \ldots, c_{\mathrm{g}17})$. The corresponding control

command input $\tilde{u}_{\rm w}$ is the target joint angles, which is denoted as $(q_0^{\rm trg}, \ldots, q_{16}^{\rm trg})$. Note that we can also directly control the joint torques.

*a) Crawling Task:* The objective of the crawling task is to move forward as fast as possible. According to the objective, the reward is designed as follows:

$$r(t) = r_{\rm mv}(t) - r_{\rm rt}(t) - r_{\rm sc}(t) - r_{\rm fd}(t) \quad (1)$$

$$r_{\rm mv}(t) = 0.25(\dot{c}_{0x}(t)e_{\rm z1}(t) + \dot{c}_{0y}(t)e_{\rm z2}(t)) \quad (2)$$

$$r_{\rm rt}(t) = 0.025|\omega_z(t)| \quad (3)$$

$$r_{\rm sc}(t) = 4 \times 10^{-9}\|\tilde{u}(t)\| \quad (4)$$

where $r_{\rm mv}(t)$ is the reward for forward movement, $(e_{\rm z1}, e_{\rm z2}, e_{\rm z3})^\top$ is the $z$-component of the rotation matrix of the body link, $r_{\rm rt}$ is the penalty for rotation, $r_{\rm sc}(t)$ is the step cost, $r_{\rm fd}(t)$ is the penalty for falling down. $r_{\rm fd}(t)$ takes 4 if the body or the head link touches the ground, otherwise it takes 0. $r_{\rm fd}(t)$ has a non-zero value once in each action. Each episode begins with the initial state where the robot is standing up and stationary, and ends if $t > 20$[s] or the sum of reward is less than $-40$.

*b) Jumping Task:* The objective of the jumping task is to jump as high as possible and to keep jumping as long as possible. According to the objective, the reward is designed as follows:

$$r(t) = r_{\rm jp}(t) - r_{\rm sc}(t) - r_{\rm fd}(t) \quad (5)$$

$$r_{\rm jp}(t) = 0.25c_{0z}Air \quad (6)$$

$$r_{\rm sc}(t) = 4 \times 10^{-9}\|\tilde{u}(t)\| \quad (7)$$

where $Air$ takes 1 if the robot floats and takes 0 otherwise, $r_{\rm jp}(t)$ is the reward for jumping, $r_{\rm sc}(t)$ is the step cost for the torque usage, $r_{\rm fd}(t)$ is the penalty for falling down. $r_{\rm fd}(t)$ takes 2 if the body or the head link touches the ground, takes 1 if the other links other than feat touch the ground; otherwise it takes 0. $r_{\rm fd}(t)$ has a non-zero value once in each action. Each episode begins with the initial state where the robot is standing up and stationary, and ends if $t > 5$[s] or $r_{\rm fd}(t)$ has a non-zero value.

### B. General Task Description

A general way to describe such rewards and end-of-episode conditions is to define several event-driven functions. Here, the event means an occurrence on the time axis; for example, each action start, each action end, each time-step start, each time-step end, each episode start, and so on. Each event-driven function is executed when the corresponding event happens, and computes a part of the reward and the end-of-episode condition. We refer to an event-driven function as an *event callback*.

For instance, the crawling task can be specified by the following three event callbacks:

**start-of-episode** :

$SumR = 0$ /* sum of reward */

**start-of-action** :

$flag =$ `false` /* no falling down */

**end-of-time-step** :

$Reward = 0.25(\dot{c}_{0x}(t)e_{\rm z1}(t) + \dot{c}_{0y}(t)e_{\rm z2}(t))$
$Reward = Reward - 0.025|\omega_z(t)| - 4 \times 10^{-9}\|\tilde{u}(t)\|$
**if** $\neg flag \wedge (c_{\rm g0} \vee c_{\rm g1})$ **then**
  $flag =$ `true` /* fall down */
  $Reward = Reward - 4$ /* falling-down penalty */
$SumR = SumR + Reward$ /* update sum-of-reward */
**if** $(SumR < -40) \vee (time > 20)$ **then**
  $EndOfEps =$ `true` /* end of episode */

where $c_{\rm g0}$ and $c_{\rm g1}$ denote the contact-with-ground flags of the body and the head link, $\wedge$ denotes a logical conjunction ("and"), and $\vee$ denotes a logical disjunction ("or"). In each event callback, the reward variable $Reward$ and the end-of-episode flag $EndOfEps$ may be computed using some sensor variables; $Reward$ and $EndOfEps$ are initialized to zero and `false` at the beginning of each callback. The assigned $Reward$ is summed during each action, and the amount is used as the reward signal for an RL agent. Once $EndOfEps$ is `true`, the current episode is terminated. The flag $flag$ is introduced so that the falling-down penalty is given less than once in an action.

Similarly, the jumping task can be specified as follows:

**start-of-action** :

$flag_1 =$ `false` /* no falling down */
$flag_2 =$ `false` /* no falling down */

**end-of-time-step** :

$Reward = -4 \times 10^{-9}\|\tilde{u}(t)\|$
**if** $\neg(\vee c_{\rm g0:17})$ **then**
  $Reward = Reward + 0.25c_{0z}$
**if** $\neg flag_1 \wedge (\vee c_{\rm g0:1})$ **then**
  $flag_1 =$ `true` /* fall down */
  $Reward = Reward - 2$ /* falling-down penalty */
  $EndOfEps =$ `true` /* end of episode */
**if** $\neg flag_2 \wedge (\vee c_{\rm g2:7,8:11,13:16})$ **then**
  $flag_2 =$ `true` /* fall down */
  $Reward = Reward - 1$ /* falling-down penalty */
  $EndOfEps =$ `true` /* end of episode */
**if** $time > 5$ **then**
  $EndOfEps =$ `true` /* end of episode */

where $c_{{\rm g}h:i,j:k}$ denotes a vector $(c_{{\rm g}h}, \ldots, c_{{\rm g}i}, c_{{\rm g}j}, \ldots, c_{{\rm g}k})$, and $\vee c_{{\rm g}i:j}$ denotes the logical disjunction $c_{{\rm g}i} \vee \cdots \vee c_{{\rm g}j}$. Thus, $\vee c_{\rm g0:17}$ is true if at least one link contacts with ground.

### C. Realization

In order to realize a robot system that accomplishes the task descriptions, we need an RL implementation for robots with a script interface. For this purpose, the SkyAI library [7] is the best solution since it has been already applied to robot-learning domains including tasks of actual robots, and it has a script interface. Thus, we employ SkyAI as the base system.

There may have been the other approaches; for example, it is possible to use PyBrain [8] which is an RL library written in Python or RL-Glue [9] which is a language-independent software package for RL experiments. The reason why we choose SkyAI rather than these alternatives is that SkyAI

is implemented in C++ as a modular architecture which is suitable for robot-learning domains, and it already has a script interface which enables us to implement the task description in a little extension.

## III. Task Description using SkyAI

In this section, we make clear how to realize the task description based on the SkyAI library. Though SkyAI has a script interface, it is difficult to describe the above event-callbacks with this script. Thus, we need to extend the script interface. In the following, first, we describe the overview of the SkyAI library [7]; then, we discuss the difficulty of the old script interface in terms of the task description; finally, we describe the extension.

### A. Overview of SkyAI

The most important feature of SkyAI is *modularization* of the RL or the other machine-learning algorithms. The modular architecture enables the high reusability and the high extensibility.

SkyAI is designed for robot-learning domains; real robot systems require a high-speed execution. To achieve this, SkyAI is implemented in *C++*. Each module is implemented as a class of C++. The classes communicate with each other through member functions. We basically use *call-by-reference* in these functions for the high-speed communication.

In order to achieve the high flexibility, the modular structure should be changed during execution after compiling the source code. SkyAI wraps the C++ class system, since the member functions for the inter-class communication are needed to be connected and disconnected. Thus, each member function is encapsulated as a *port* class. Each module can have any number of ports. Ports can be connected and disconnected at any time in execution, which enables to reconfigure the modular structure.

A script language is defined to provide an interface of modular manipulations during execution. Specifically, instantiating modules, connecting ports, and setting parameters of the modules (e.g. a learning rate) can be specified in the script language. We refer to the script of SkyAI as an *agent script*.

Fig. 2 illustrates an example modular structure around an RL module. In an on-line learning system, there are several kinds of cycles, such as episode, action, and time step of low-level controller. The SkyAI modular architecture can handle any kinds of cycles as shown in Fig. 2.

The SkyAI architecture enables to separate domain-specific modules and generic modules. The domain-specific modules are a low-level robot controller and a task module. On the other hand, RL algorithms can be implemented as generic modules.

### B. Problem of SkyAI's Task Description

In the original SkyAI architecture, there have been two ways to define a task:

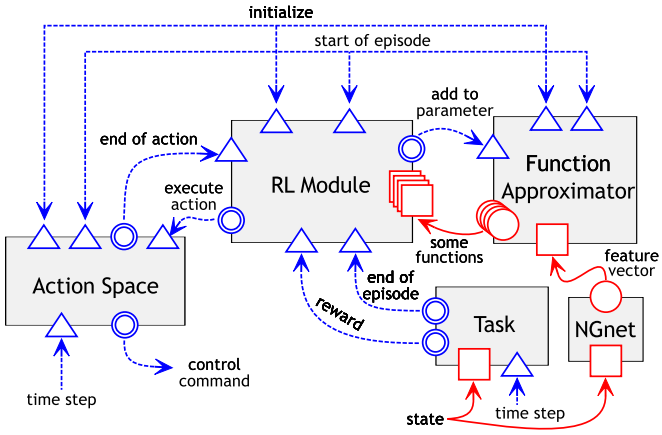



Fig. 2. Example modular structure around an RL module.

Implementing a task module in C++: A task module is defined as a module of SkyAI that includes signal ports emitting reward and end-of-episode flag (see Fig. 2).

Using basic calculation modules: SkyAI provides a number of basic calculation modules, such as addition, multiplication, square, vector operations, logical operations, and so on. Combining these modules, we can describe a reward and a end-of-episode condition.

In both ways, we can represent the same information as described in Section II-B. However, implementing a task module in C++ depresses the flexibility; the user needs to recompile the system in order to make the task module available. On the other hand, the user can define a task without recompiling by using basic calculation modules. However, many basic modules are required to describe a task, which is very complicated for the user.

### C. Extension for Task Description

A better idea to define a task is writing the task definition directly in the SkyAI's script, as we described in Section II-B. Such a description mechanism should be compatible with the modular architecture of SkyAI. Thus, we implement a *universal task module* that is a core mechanism to define a task. In addition, though the script interface originally has had a mechanism for defining a function, the mechanism is not enough to define the callbacks as we discussed in Section II-B. Thus, we also extend the script interface of SkyAI.

*1) Universal Task Module:* This module has several *slot ports* that are called by event signals, such as each episode start, each action start, and so on. In the slot port of each event, the corresponding callback function defined in an agent script is executed. In the agent script, we can define any number of instances of the universal task module, and each of the instances can have individual callbacks. Thus, by introducing the universal task module, we can define any kinds of tasks in the same agent script; such definition can be done dynamically during execution.

Fig. 3 shows the universal task module defined for the simulated humanoid robot. In order to make the sensor

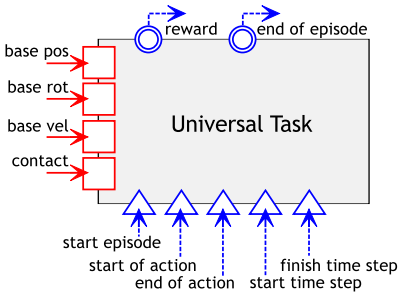



Fig. 3. Universal task module of SkyAI.

variables available in the callbacks, some sensor input ports are defined.

*2) Extension of Script Interface:* In order to define the reward functions and the end-of-episode conditions, at least, the following two extensions are needed:

Equation parser: For writing the equations of reward functions and end-of-episode conditions, we implement a equation parser for the script interface.

Control syntax: In reward and end-of-episode definitions, control flow statements are useful; at least, "if/else" statements are required. Thus, we implement a control syntax of the script interface.

Fig. 4 shows an example of callback functions written in an agent script, which describes the crawling task. The reason why there are many explicit casts is that the mechanism to manipulate the C++ variables from the script is designed to enhance the computation speed. The user can register such functions with an instance of the universal task module.

### D. Applying to the Other Robot Systems

The other robot systems have different sensor input that will be used to define a reward and a end-of-episode condition. Thus, another universal task module should be defined for the other robot system. Such a new universal task module is considered to have many similar mechanism with the universal task module for our humanoid robot in Fig. 3. Thus, SkyAI provides a base class written in C++, and an individual universal task module for each robot system is inherited from the base class. An inherited universal task module is available by adding some sensor input ports.

## IV. Demonstrations

Using the implemented task description mechanism in SkyAI, we design six motion-learning tasks of the humanoid robot introduced in the previous section. The tasks are a crawling, a handstanding, a jumping, a forward rolling, a backward rolling, and a turning task. The whole source codes are available on the SkyAI's website: `http://skyai.org`. A video of the experimental results is available on the author's website[1].

We use a simulated humanoid robot shown in Fig. 1. Its height is 0.328m. It weights 1.20kg. Each joint torque is limited to 1.03Nm, and a PD-controller is embedded on it. The dynamics simulation is calculated with a time step $\delta t = 0.2$[ms]. Experiments are performed in simulation using a dynamics simulator ODE (Open Dynamics Engine; `http://www.ode.org`).

[1] `http://robotics.naist.jp/~akihiko-y/movs/SkyAI-6tasks.mp4`

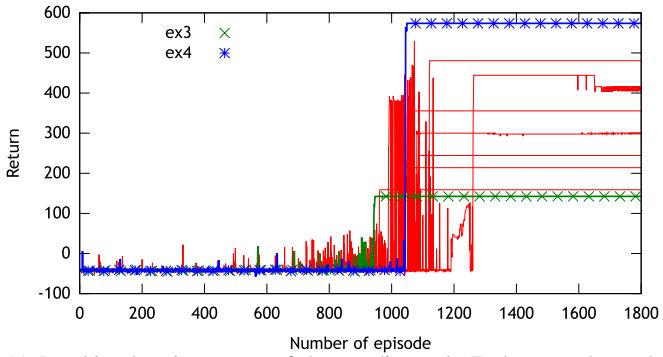
```
def episode_start(task_id)
{
  task_id.memory ={TmpR1= 0.0;  TmpR2= 0.0;}
    // sum-of-reward, total-time
}
def action_start(task_id)
{
  task_id.memory ={TmpB1= false;} // no falling down
}
def timestep_end(task_id)
{
  task_id.memory ={
      Reward= 0.25*(
        cast<real>(BaseVel[0])*cast<real>(BaseRot[(0,2)]) +
        cast<real>(BaseVel[1])*cast<real>(BaseRot[(1,2)]))
      Reward= cast<real>(Reward) -
        0.025*fabs(cast<real>(BaseVel[5]))
    }
  if(!cast<bool>(task_id.memory.TmpB1) &&
      (cast<bool>(task_id.memory.ContactWithGround[0]) ||
        cast<bool>(task_id.memory.ContactWithGround[1])))
  {
    task_id.memory ={
        TmpB1= true  // fall down
        Reward= cast<real>(Reward) - 4.0
          // falling-down penalty
      }
  }
  task_id.memory ={
      TmpR1= cast<real>(TmpR1) + cast<real>(Reward)
        // update sum-of-reward
      TmpR2= cast<real>(TmpR2) + cast<real>(TimeStep)
        // update total-time
    }
  if(cast<real>(task_id.memory.TmpR1) < -40.0 ||
      cast<real>(task_id.memory.TmpR2) > 20.0)
  {
    task_id.memory ={EndOfEps= true;}  // end of episode
  }
}
```

Fig. 4. Callback functions that describe the crawling task. The keyword `def` denotes to define a function.

For the RL method, we employ the Peng's Q($\lambda$)-learning algorithm [10], which is an on-line RL method, i.e. the update procedure is applied after each action. As the action space, we use DCOB [11] which is a discrete action set efficient in learning from scratch. As the basis functions, we use Normalized Gaussian Network [12] which is a popular function approximator in RL applications. As the DoF (degree of freedom) configurations, we use a 4-DoF configuration (one DoF for each leg) in learning the turning task, and a 5-DoF configuration (some joints are coupled to be bilaterally symmetric) otherwise. The detailed description of the DoF configurations is written in [13].

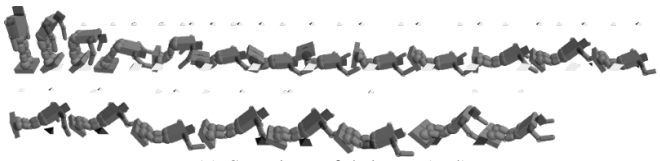
### A. Crawling Task

The crawling task is the same as defined in Section II-A. Fig. 5(a) shows the resulting learning curves, where two curves are highlighted whose snapshots are shown in Fig. 5(b) and 5(c). As shown in Fig. 5(a), there are some kinds of local maxima. In each run, the robot moves forward by crawling. The difference is the forward speed of crawling. The speed of the 4-th episode (Fig. 5(c)) is much faster than that of the 3-th episode (Fig. 5(b)). In the faster behavior, the robot seems to be moving its body more dynamic than in the slower behavior.



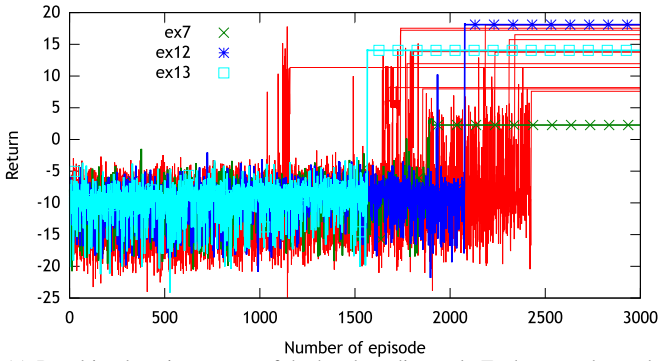

(a) Resulting learning curves of the crawling task. Each curve shows the return per episode in a run. Two curves are highlighted whose snapshots are shown below.

(b) Snapshots of 3-th run (ex3).



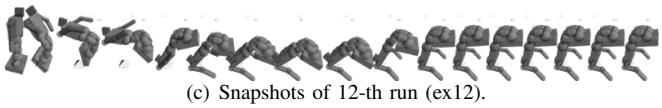
(c) Snapshots of 4-th run (ex4).

Fig. 5. Results of learning the crawling task.

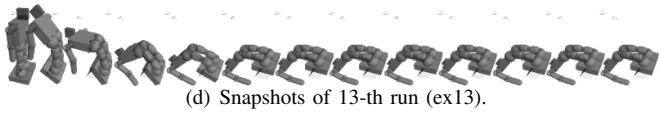



(a) Resulting learning curves of the handstanding task. Each curve shows the return per episode in a run. Three curves are highlighted whose snapshots are shown below.

(b) Snapshots of 7-th run (ex7).



(c) Snapshots of 12-th run (ex12).



(d) Snapshots of 13-th run (ex13).

Fig. 6. Results of learning the handstanding task.

### B. Handstanding Task

The objective of the handstanding task is to support the body on the hands and the head with keeping the torso and the legs vertically in the air. According to the objective, the reward is designed as follows:

$$r(t) = r_{\rm hs}(t) - r_{\rm mv}(t) - r_{\rm sc}(t) - r_{\rm fd}(t) \tag{8}$$

$$r_{\rm hs}(t) = 0.25 c_{0z}(t) Air \tag{9}$$

$$r_{\rm mv}(t) = 0.0025 \|(\dot{c}_{0x}(t), \dot{c}_{0y}(t))\| \tag{10}$$

$$r_{\rm sc}(t) = 4 \times 10^{-9} \|\tilde{u}(t)\| \tag{11}$$

where $Air$ takes 1 if the hands and the head touch the ground and the torso and the leg links are in the air, and takes 0 otherwise. $r_{\rm hs}(t)$ is the reward for handstanding, $r_{\rm mv}$ is the penalty for $xy$-movement, $r_{\rm sc}(t)$ is the step cost. Each episode begins with the initial state where the robot is standing up and stationary, and ends if $t > 5$[s].

Fig. 6(a) shows the resulting learning curves, where three curves are highlighted whose snapshots are shown in Fig. 6(b), 6(c), and 6(d). As shown in Fig. 6(a), there are some kinds of local maxima. The acquired behavior of the 12-th episode (Fig. 6(c)) is the handstanding that we expected. In the 13-th episode (Fig. 6(d)), the robot failed to pull up the torso. In the 7-th episode (Fig. 6(b)), the robot swings the body.

### C. Jumping Task

The jumping task is the same as defined in Section II-A. Fig. 7(a) shows the resulting learning curves, where three curves are highlighted whose snapshots are shown in Fig. 7(b), 7(c), and 7(d). As shown in Fig. 7(a), there are some kinds of local maxima. In each run, the robot acquired a jumping behavior. However, in the 14-th episode (Fig. 7(d)), the robot jumps once in the episode, while in the 4-th and 13-th episodes (Fig. 7(b), 7(c)), the robot jumps multiple times.

### D. Forward Rolling Task

The objective of the forward rolling task is to roll the body forward as far as possible. According to the objective, the reward is designed as follows:

$$r(t) = r_{\rm rlf}(t) r_{\rm sc}(t) \tag{12}$$

$$r_{\rm rlf}(t) = 0.0125 \omega_y \tag{13}$$

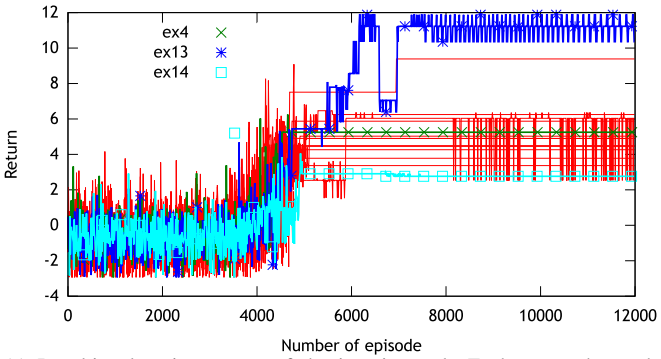
$$r_{\rm sc}(t) = 4 \times 10^{-9} \|\tilde{u}(t)\| \tag{14}$$

where $r_{\rm rlf}(t)$ is the reward for forward rolling, and $r_{\rm sc}(t)$ is the step cost. Each episode begins with the initial state where the robot is standing up and stationary, and ends if $t > 10$[s].

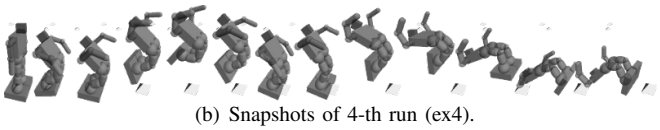
Fig. 8(a) shows the resulting learning curves, where three curves are highlighted whose snapshots are shown in Fig. 8(b), 8(c), and 8(d). As shown in Fig. 8(a), there are some kinds of local maxima. In each run, the robot rolls its body forward. The difference is the amount of roll. The robot rolls once in the 4-th episode (Fig. 8(b)), twice in the 8-th episode (Fig. 8(c)), and five times in the 10-th episode (Fig. 8(d)).

### E. Backward Rolling Task

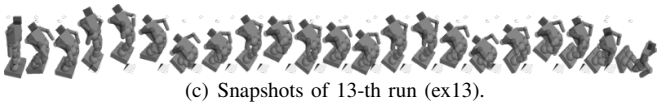
The objective of the backward rolling task is to roll the body backward as far as possible. Thus, this task is almost




(a) Resulting learning curves of the jumping task. Each curve shows the return per episode in a run. Three curves are highlighted whose snapshots are shown below.



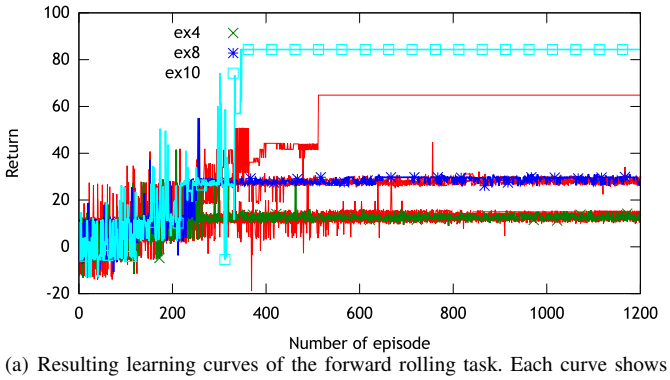
(b) Snapshots of 4-th run (ex4).



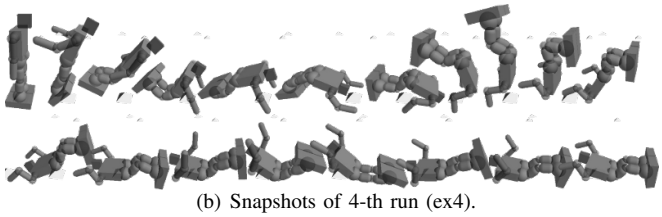
(c) Snapshots of 13-th run (ex13).

(d) Snapshots of 14-th run (ex14).

Fig. 7. Results of learning the jumping task.

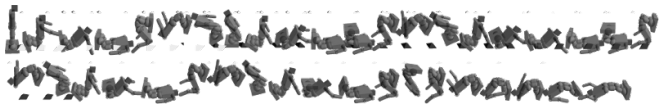



(a) Resulting learning curves of the forward rolling task. Each curve shows the return per episode in a run. Three curves are highlighted whose snapshots are shown below.



(b) Snapshots of 4-th run (ex4).

(c) Snapshots of 8-th run (ex8).



(d) Snapshots of 10-th run (ex10).

Fig. 8. Results of learning the forward rolling task.

the same as the forward rolling task; the difference is the robot's hardware configuration. For example, the knees of the robot are constrained similar to those of humans. The task design is almost the same as that of the forward rolling task; only the difference is the sign of $r_{\mathrm{rlf}}(t)$ is negative in order to evaluate the backward roll.

Fig. 9(a) shows the resulting learning curves, where two curves are highlighted whose snapshots are shown in Fig. 9(b) and 9(c). As shown in Fig. 9(a), there are some kinds of local maxima. In each run, the robot rolls its body backward. The difference is the amount of roll. The robot rolls once in the 4-th episode (Fig. 9(b)), but more than three times in the 12-th episode (Fig. 9(c)).

### *F. Turning Task*

The objective of the turning task is to turn around the $z$-axis as fast as possible. According to the objective, the reward is designed as follows:

$$r(t) = r_{\mathrm{tn}}(t) - r_{\mathrm{sc}}(t) - r_{\mathrm{fd}}(t) \tag{15}$$

$$r_{\mathrm{tn}}(t) = 0.0125\omega_z(t) \tag{16}$$

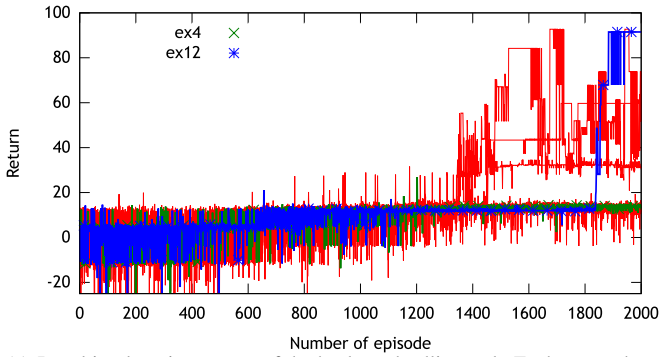
$$r_{\mathrm{sc}}(t) = 0.0025\|(\dot{c}_{0x}, \dot{c}_{0y})\| + 4 \times 10^{-9}\|\tilde{u}(t)\| \tag{17}$$

where $r_{\mathrm{tn}}(t)$ is the reward for turning, $r_{\mathrm{sc}}(t)$ is the step cost for the $x-y$ global movement and the torque usage, $r_{\mathrm{fd}}(t)$ is the penalty for falling down. $r_{\mathrm{fd}}(t)$ takes $4$ if the body link touches the ground, takes $0.1$ if the head link touches the ground; otherwise it takes $0$. $r_{\mathrm{fd}}(t)$ has a non-zero value once in each action. Each episode begins with the initial state where the robot lies down and stationary, and ends if $t > 20$[s] or the sum of reward is less than $-40$.

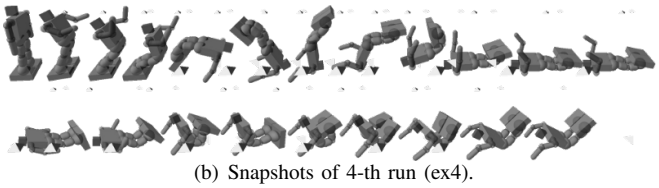
Fig. 10(a) shows the resulting learning curves, where two curves are highlighted whose snapshots are shown in Fig. 10(b) and 10(c). As shown in Fig. 10(a), there are some kinds of local maxima. In each run, the robot turns. However, in the 4-th episode (Fig. 10(b)), the robot rotates around the left front of the waist link; while in the 9-th episode (Fig. 10(c)), the robot rotates around the right shoulder. The latter one is inefficient. This difference causes the variation of the turning speed. The turning speed of the 4-th episode (Fig. 10(b)) is faster than that of the 9-th episode (Fig. 10(c)).

## V. Discussion

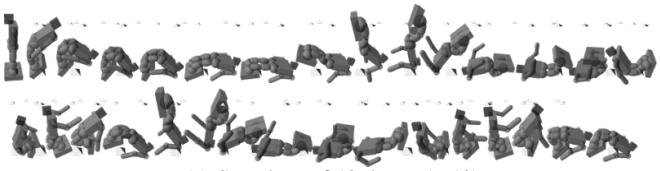
Through the experiments, we found that there are some local maxima in each task. Such local maxima are considered to be obtained not only in our case, but also in the other cases. A local maximum may be a behavior that its designer expected, while the other maxima may not. This problem is considered to be caused by the difficulties of both designing a reward function and applying an RL method for a large DoF domain. At least, in order to overcome the former difficulty, we need to make an interactive learning system into which the user can feed back the difference from the expectation;




(a) Resulting learning curves of the backward rolling task. Each curve shows the return per episode in a run. Two curves are highlighted whose snapshots are shown below.

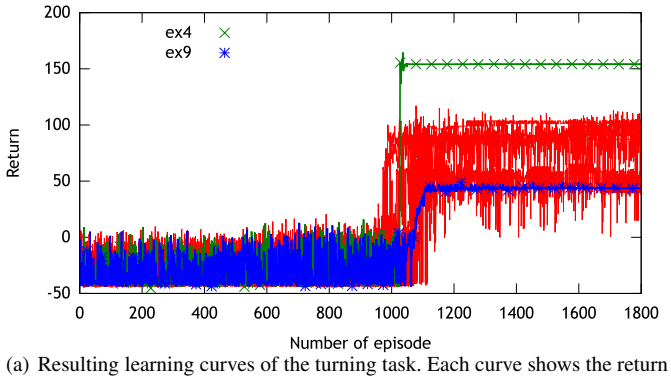


(b) Snapshots of 4-th run (ex4).



(c) Snapshots of 12-th run (ex12).

Fig. 9. Results of learning the backward rolling task.




(a) Resulting learning curves of the turning task. Each curve shows the return per episode in a run. Two curves are highlighted whose snapshots are shown below.

(b) Snapshots of 4-th run (ex4).

(c) Snapshots of 9-th run (ex9).

Fig. 10. Results of learning the turning task.

such an approach is found in [14].

Another problem is that in the experiments of this paper, we specified the state and the action space of the RL agent for each task, which may be difficult for non-expert users. However, using the learning strategy fusion method [13], the state and the action space suitable to the task are automatically chosen, which may make the behavior design easier.

## VI. Conclusion

In this paper, in order to unify the "robot language" approach and the reinforcement learning (RL) framework, we proposed a robot task description mechanism; a task is described with several event-driven functions where the reward and the end-of-episode condition are defined. Using this task description, we can design tasks much more simply than the traditional robot languages, since RL methods release us from specifying the robot/environment models and the task procedure. This task description mechanism was implemented by extending the script interface of the C++ RL library SkyAI. Using the task description mechanism, we designed six kinds of dynamic behaviors for a humanoid robot; a crawling, a handstanding, a jumping, a forward rolling, a backward rolling, and a turning task. By an RL method, the behaviors of these tasks were obtained from their task descriptions. We also discussed the current problems of our approach, which are planned to be solved by our future work.

## References

[1] A. Okano, H. Matsubara, and H. Inoue, "Design and implementation of a task-oriented robot language," *Advanced Robotics*, vol. 3, no. 3, pp. 177–191, 1988.

[2] H. Kimura, T. Yamashita, and S. Kobayashi, "Reinforcement learning of walking behavior for a four-legged robot," in *Proceedings of the 40th IEEE Conference on Decision and Control*, 2001.

[3] J. Zhang and B. Rössler, "Self-valuing learning and generalization with application in visually guided grasping of complex objects," *Robotics and Autonomous Systems*, vol. 47, no. 2-3, pp. 117–127, 2004.

[4] J. Kober, A. Wilhelm, E. Oztop, and J. Peters, "Reinforcement learning to adjust parametrized motor primitives to new situations," *Autonomous Robots*, vol. 33, pp. 361–379, 2012, 10.1007/s10514-012-9290-3.

[5] E. Theodorou, J. Buchli, and S. Schaal, "Reinforcement learning of motor skills in high dimensions: A path integral approach," in *the IEEE International Conference on Robotics and Automation (ICRA'10)*, may 2010, pp. 2397–2403.

[6] J. Kober and J. Peters, "Learning motor primitives for robotics," in *the IEEE International Conference on Robotics and Automation (ICRA'09)*, 2009, pp. 2509–2515.

[7] A. Yamaguchi and T. Ogasawara, "Skyai: Highly modularized reinforcement learning library —concepts, requirements, and implementation—," in *the 10th IEEE-RAS International Conference on Humanoid Robots (Humanoids'10)*, Nashville, TN, US, 2010, pp. 118–123.

[8] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "Pybrain," *Journal of Machine Learning Research*, vol. 11, pp. 743–746, 2010.

[9] B. Tanner and A. White, "Rl-glue: Language-independent software for reinforcement-learning experiments," *Journal of Machine Learning Research*, vol. 10, pp. 2133–2136, 2009.

[10] J. Peng and R. J. Williams, "Incremental multi-step Q-learning," in *International Conference on Machine Learning*, 1994, pp. 226–232.

[11] A. Yamaguchi, J. Takamatsu, and T. Ogasawara, "Constructing action set from basis functions for reinforcement learning of robot control," in *the IEEE International Conference on Robotics and Automation (ICRA'09)*, Kobe, Japan, 2009, pp. 2525–2532.

[12] M. Sato and S. Ishii, "On-line EM algorithm for the normalized Gaussian network," *Neural Computation*, vol. 12, no. 2, pp. 407–432, 2000.

[13] A. Yamaguchi, J. Takamatsu, and T. Ogasawara, "Learning strategy fusion to acquire dynamic motion," in *the 11th IEEE-RAS International Conference on Humanoid Robots (Humanoids'11)*, Bled, Slovenia, 2011, pp. 247–254.

[14] A. Tenorio-Gonzalez, E. Morales, and L. Villaseñor Pineda, "Dynamic reward shaping: Training a robot by voice," in *Advances in Artificial Intelligence – IBERAMIA 2010*, ser. Lecture Notes in Computer Science, 2010, vol. 6433, pp. 483–492.