Automatic learning of motion primitives for modular robots

Vojtěch Vonásek[1], Martin Saska[1]

***Abstract*— Modular robots consist of many mechatronic modules that can be connected into various shapes and therefore adapted for a given task and environment. Motion of the robots can be achieved by locomotion generators that control joints connecting the modules. A robot can be equipped with several locomotion generators that provide basic motion primitives. A sequence of motion primitives is found using a motion planner in order to visit a given goal position. Each primitive needs to be prepared by an optimization process where parameters of a locomotion generator are tuned. Due to the possibility to create robots of various shapes, it is not easy to estimate in advance what kind of motions a robot can perform. Moreover, motion capabilities can also be influenced by failures of individual modules. Human operators may prefer motions like 'crawl-forward' or 'move-left' but such primitives might not be achievable by all robots. In this paper, we discuss how to automatically learn motion primitives that are suitable for a given robot and task. Experimental verification in simulation as well as with physical robots is presented.**

I. Introduction

Modular robots are versatile robotic systems that are built from many basic mechatronic modules. The modules can be reconfigured to various shapes (Fig. 1), which allows modular robots to adapt for a given task and to recover from failures [10]. Modular robots can move using the concept of self-reconfiguration [1] or using joint-control locomotion that is achieved by changing angles of the joints connecting the modules. The joint-control locomotion, which is considered in this paper, can be modeled using Central Pattern Generators (CPG) [2] providing rhythmic signals for the actuators. This is inspired by evidence from nature, where locomotion is generated by coupled neuro-oscillators [11]. CPGs have been used to generate locomotion of snake-like robots, to control swimming robots, to generate biped locomotion of humanoid robots and also to control modular robots. In order to produce a gait, CPG parameters need to be tuned, which can be solved using evolutionary-based approaches like Genetic algorithms (GA) [4], [9], [7], [12], [15].

Locomotion generators are efficient in providing basic motions, but they do not consider overall situation in an environment that is necessary to avoid obstacles, other robots and to visit a desired goal position. Although the locomotion generators can be extended to cope with altering terrain, finding suitable parameters for such models is time consuming. In our previous work, we propose to equip a robot with multiple motion primitives modeled by locomotion generators and to utilize them in a motion planner [15].

The success of the motion planning with motion primitives is determined by the quality of motion primitives. In the case of simple and fully functional robots, the motion primitives are determined by a human operator who designs a cost function to be optimized. As the modular robots can reconfigure to various shapes, it is not always easy to estimate motions, that a robot can perform. Moreover, robot behavior is influenced by failures of the modules. A motion that can be efficiently performed by a fully functional robot can be prohibited or performed inefficiently in the case of failures.

To find motion suitable for robots of unusual shapes as well as for robots under failures, the motion primitives should be learned automatically rather than designed by a human operator. In this paper, we propose a system to automatically learn a set of motion primitives for modular robots. The assumed task of the robot is to move between arbitrary places in the environment while avoiding obstacles.

II. Motion primitives

Motion primitives are basic gaits such as 'crawl-forward' or 'climb-step'. Each motion primitive provides a control signal $a_i(t), i = 1, \ldots, m$, where m is the number of joints (modules). The signal defines desired angles of the joints. The signals are obtained using a Central Pattern Generator that is parametrized by a vector $\mathbf{x}^p$. Depending on the used CPG, the parameters define e.g. an intrinsic frequency or weights of coupling between neighboring modules. To move a robot using a primitive p, the corresponding parameters $\mathbf{x}^p$ are loaded to the generator, which produces signals $a_i(t)$ for the actuators. The robot is driven by the primitive over time τ^p. A robot is equipped with a set of k motion primitives $P = \{\mathbf{x}^1, \ldots, \mathbf{x}^k\}$ that represent basic motion skills suitable for a given task. For example, primitives like 'crawl-forward'

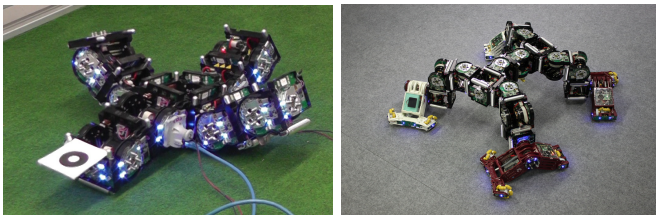


Fig. 1. Examples of modular robots developed withing Symbrion/Replicator EU projects [8].

[1]Dept. of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, Technicka 2, 166 27, Prague 6, Czech Republic; {vonasek,saska}@labe.felk.cvut.cz The work was supported by Grant Agency of the Czech Technical University in Prague, grant No. SGS15/157/OHK3/2T/13. Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005), is greatly appreciated.

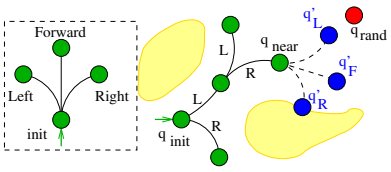


Fig. 2. Example of the configuration tree built by RRT-MP using three motion primitives (blue nodes). The tree will be extended using q'_L, as it is the closest configuration towards q_{rand}.

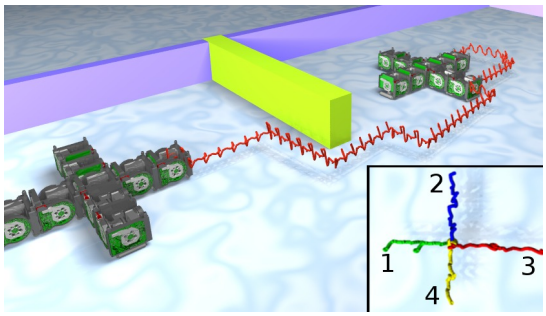


Fig. 3. Example of motion plan generated by RRT-MP for Quadropod robot equipped with four primitives.

and 'turn' can be used in a large environment without obstacles, and 'climb-step' can be added if the climbing over a stepped obstacle is expected.

III. Motion Planning with Motion Primitives

The task of the motion planner is to find a sequence of motion primitives leading from an initial configuration $q_{init} \in \mathcal{C}_{free}$ to a goal configuration $q_{goal} \in \mathcal{C}_{free}$, where the set $\mathcal{C}_{free} \subseteq \mathcal{C}$ consists of free (feasible) configurations of the configuration space $\mathcal{C}$. A configuration $q = (x, y, z, \alpha, \beta, \gamma, a_1, \ldots, a_m)$, where m is the number of modules, describes 3D position (x, y, z) and 3D rotation (α, β, γ) of its pivot module and angles a_i of all joints.

The plan is found using the RRT-MP (Rapidly Exploring Random Tree with Motion Primitives) [15] planner. RRT-MP iteratively builds a tree $\mathcal{T}$ of free configurations of the robot rooted at q_{init}. In each iteration, a random point $q_{rand} \in \mathcal{C}$ is generated and its nearest neighbor $q_{near} \in \mathcal{T}$ in the tree is found. Each motion primitive $p \in P$ is then applied to the simulated robot starting from the configuration q_{near} over time τ^p to get a new configuration q'_p. An example of the expansion step is depicted in Fig. 2. The tree is extended by such a configuration q'_p, that is closest to the random configuration q_{rand}. This procedure repeats until the goal region is reached or the number of allowed iterations K_{max} is achieved. The final plan, which is a sequence of the motion primitives, is sent to the robot for execution. An example of a plan is depicted in Fig. 3.

The overall situation in the environment is considered on the motion planning level. This allows us to equip a robot with simple locomotion generators providing only basic motions in the vicinity of the robot. The obstacle avoidance and reaching of the goal position is ensured by the motion planner.

Algorithm 1: Main loop of RRT-MP

Input: initial configuration q_{init}, goal configuration q_{goal}, set of motion primitives $P = (\mathbf{x}^1, \ldots, \mathbf{x}^n)$, probability of creating random pattern p_r
Output: plan or failure

$\mathcal{T}$.initialize(); // create empty configuration tree
$\mathcal{T}$.add(q_{init});
for $iterator = 1 : K_{max}$ **do**
  q_{rand} = generate random configuration;
  q_{near} = find nearest configuration to q_{rand} in $\mathcal{T}$;
  $R = \emptyset$;
  foreach $\mathbf{x}^p \in P$ **do** // P is the set of motion primitives
    if $rand() < p_r$ **then**
      $\mathbf{x}^p$ = random settings of CPG's parameters;
    end
    $(a_1(t), a_2(t), \ldots, a_m(t))$ = control signal generated by CPG with setting $\mathbf{x}^p$, $0 < t \leq \tau^p$;
    q = apply $a_i(t)$ to motion model starting from q_{near};
    $R = R \cup \{q\}$;
  end
  q_{new} = nearest from R to q_{rand};
  $\mathcal{T}$.add(q_{new});
  $\mathcal{T}$.addEdge(q_{near}, q_{new});
  if $\varrho(q_{new}, q_{goal}) < d_{goal}$ **then** //goal reached
    return plan from $\mathcal{T}$;
  end
end
return failure;

IV. Learning of Motion Primitives

The motion primitives are prepared by an optimization of the parameters $\mathbf{x}^p$ according to a cost function $f^p(\mathbf{x}^p)$. The cost functions f^p are specific for each motion primitive, as the primitives provide different motions.

In this paper, the parameters $\mathbf{x}^p$ are found using Particle Swarm Optimization (PSO) [5]. PSO is inspired by behaviors of bird flocks or fish schools. In PSO, each particle represents a candidate solution, whose quality is evaluated using a cost function. Let $\bar{\mathbf{x}}_i^p$ denote the best solution of a particle i in its history and let $\hat{\mathbf{x}}^p$ denote the best solution among all particles. At the beginning, the particles $\mathbf{x}_i^p$ are randomly placed into the search space and their velocities $\mathbf{v}_i(0)$ are set randomly. In each iteration, new velocity $\mathbf{v}_i(k+1)$ and position $\mathbf{x}_i^p(k+1)$ of i-th particle are computed as

$$\begin{aligned} \mathbf{v}_i(k+1) &= \mathbf{v}_i(k) + \varphi_1 r_1(\bar{\mathbf{x}}_i^p - \mathbf{x}_i^p) + \varphi_2 r_2(\hat{\mathbf{x}}^p - \mathbf{x}_i^p) \\ \mathbf{x}_i^p(k+1) &= \mathbf{x}_i^p(k) + \mathbf{v}_i(k+1), \end{aligned} \quad (1)$$

where φ_1, φ_2 influence speed of the exploration of search space and r_1, r_2 are random numbers from $U(0, 1)$.

The particles represent settings of a CPG, therefore their quality is determined by the quality of the motion provided by the CPG. As the optimization on real robots is time consuming, the optimization starts in a simulation before the best solutions are applied to real robots. The advantage of PSO is a fast convergence to a suitable solution. Moreover, it is not necessary to design any crossover operators as in the case of GA-based methods [3], [4]. The quality of the particle is measured using a cost function $f^p(\mathbf{x}^p)$ that needs to be defined to learn a desired primitive p. In this paper, we

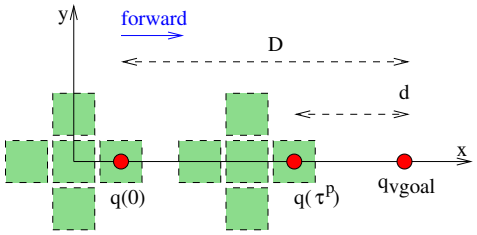


Fig. 4. Fitness function for predefined primitives. $q(0)$ is the initial position of the robot, $q(\tau^p)$ is the final position after time τ^p.

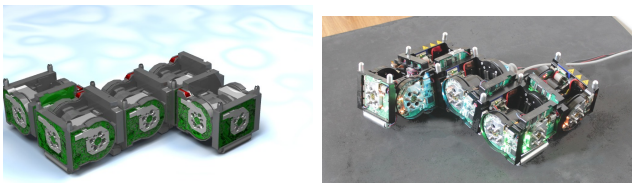


Fig. 5. S-Bot modular robot.

investigate two basic paradigms to design the cost functions: a) predefined primitives and b) task-based primitives.

A. Predefined primitives

In the first case, the cost functions are designed for each primitive separately by a human operator. The design is made based on the experience with the robot. The primitives can be relatively simple, as the motion planner combines the primitives in order to achieve a more complex task. The cost function can be based e.g. on the speed of motion. In this paper, the robots operate in a flat environment, where fast motions are preferred. Such primitives can be achieved e.g. using the cost function $f^p(\mathbf{x}^p) = D - d$, where D is the initial distance between robot and a virtual goal and d is the distance between the final position of the robot and the goal. Both D and d are measured as 3D Euclidean distance between pivot modules (Fig. 4). The virtual goal is placed in the direction defined by the primitive. For example, the goal is placed in front of the robot in the case of 'move-forward' primitive. This cost function forces the robots to move in the desired direction as fast as possible.

B. Task-based primitives

Optimization of the predefined primitives my be inefficient if a robot is not able to perform the desired motions due to its unusual shape or even due to module failures. Failures of modules decrease motion capabilities of the whole robot. The effect of failures depends both on the amount of failed modules but also on their relative positions in the robot. The number of possible configurations of failed modules grows exponentially with the amount of modules. If the robot cannot recover from failures by reconfiguration, its motions should be adapted considering effect of the broken modules [14].

In the case of unusual shapes as well as in the case of failures, it could be inefficient to optimize predefined primitives. Moreover, humans may tend to choose nice primitives (e.g. 'walk-forward') rather than primitives suitable for actual

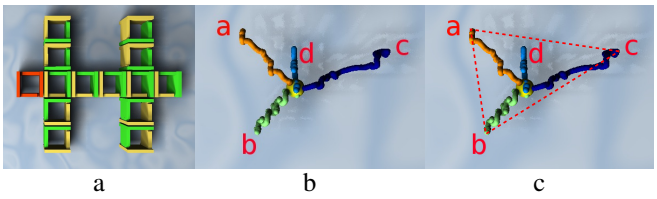


Fig. 6. Example of task-based primitives ($K = 4$) for Lizard robot with one broken module (the broken module is depicted in red) (a). The letters denote end points of the primitives (b). The cost function of these primitives is computed as the area of the convex hull of these points (points a, b, c) in this case (c).

configuration of the robot (e.g. 'walk-forward-while-rotating-left'). For example, straight forward and backward motions cannot be easily performed on the S-bot robot depicted in Fig. 5 because of the single side modules. However, S-bot can easily rotate and it can easily achieve motions with allowed rotation.

In such situations, primitives should be learned automatically in order to allow the RRT-MP planner to fulfill a given task. To find these task-based primitives, they have to be learned simultaneously. This requires to collect the parameter vectors $\mathbf{x}^i$ of each employed locomotion generator i into a single optimization vector $\hat{\mathbf{x}}$, $\hat{\mathbf{x}} = (\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^K)$, where K is the number of primitives being learned. The task-based primitives described by $\hat{\mathbf{x}}$ can be optimized by PSO.

This requires to define a task-based cost function describing an ability of the robot to fulfill a given task. The task considered in this paper requires a robot to move in the environment, therefore the cost function can be defined as the quality of the motion plans constructed using the primitives described by $\hat{\mathbf{x}}$. However, the evaluation of this cost function would be time consuming due to necessity to run the motion planner repeatedly.

A faster method to evaluate the task-based cost function for a robot moving in an environment is based on assumptions that the motion primitives should provide motions in various directions. This is suitable in the considered task, where the motion plans need to be constructed between various start/goal positions. The task-based cost function should favor fast motions but it has to ensure that these motions are oriented to different directions. Such a cost function can be computed as the area of the convex hull of the end points of the primitives, which is suitable for robots operating in flat environment, where amplitude of motion is not important. Example of this cost function is depicted in Fig. 6. The motion planning with task-based primitives derived by this cost-function is referred to as RRT-MP$_m$ in the rest of the paper.

V. Experimental verification

A. Failure recovery scenario

The system for automatic learning of task-based motion primitives was verified in the failure recovery scenario. We assume joint failures, i.e., a failed joint is stuck in its last position and it cannot be controlled. When a failure occurs,

the motion primitives should be adapted to enable the robot to finish its task or to visit a repair station.

Besides the RRT-MP$_m$ strategy, that derives the motion primitives automatically, two other strategies were tested: RRT-MP$_r$ and RRT-MP$_a$. These two strategies utilize a set of predefined primitives. The strategy RRT-MP$_r$ utilizes the predefined primitives learned for a fully functional robot. Such a naive strategy can be useful in situations, where failures influence motion abilities only slightly. If the performance of motion primitives decreases significantly, no plan can be found, which indicates, that the primitives should be adapted or that new primitives should be used.

In the RRT-MP$_a$ strategy, the predefined primitives designed for a fully functional robot are adapted to failures before they are used in the motion planner. The adaptation (optimization) is run using a physical simulation, where failures of joints are taken into account. Parameters $\mathbf{x}^p$ of each primitive are optimized using the original cost function defined for the healthy robot.

The predefined primitives used in RRT-MP$_r$ and RRT-MP$_a$ provide basic motion in four directions: $P = \{$'move-left','move-forward','move-right','move-back'$\}$. The primitives are realized using a sine CPG providing control signals $a_i(t) = A_i \sin(\omega_i t + \varphi_i) + O_i$. Each primitive p is represented by vector $\mathbf{x}^p = (A_i, O_i, \omega_i, \varphi_i), i = 1, \ldots, m$. The motion primitives were optimized using PSO with 20 particles and 100 iterations (we refer to [13] for details about the optimization). The predefined primitives were optimized using the cost function described in section IV-A (Fig. 4). The task-based primitives used in RRT-MP$_m$ were realized by the same locomotion generator. RRT-MP$_m$ is run with $K = 4$ primitives.

Lizard ($m = 14$) Quadropod ($m = 9$)

Fig. 7. Robots used in the simulation experiment.

The proposed failure recovery strategies RRT-MP$_r$, RRT-MP$_a$ and RRT-MP$_m$ have been experimentally verified in a scenario with obstacles (Fig. 8) using CoSMO [8] modular robots (Fig. 7). The task of failure recovery is to find a plan to a distant place behind the obstacle. Performance is evaluated using the success rate s-ratiocomputed as percentage of trials where the robot reached the goal position to distance $d_{goal} = 1\times$ modules size or less. For each robot (with m modules), the planner was run for all combinations of failures $B_i = \binom{m}{i}$ considering $i = 0, 1, 2, 3$ broken modules. The planners were run 20 times for each of these combinations with maximum allowed number of iterations $K_{max} = 150$. The performance of failure recovery with i broken modules is measured as the percentage of cases where the RRT-MP achieved s-ratio greater or equal to a given threshold out of all B_i cases. This performance is depicted by graphs for each amount of failed modules in Fig. 10 (values of B_i are in each graph). A better failure recovery strategy is indicated by higher percentage of combinations where a plan was found with the given probability.

The failures of one, two or three modules have smallest impact to the Lizard robot. Both strategies utilizing original primitives (RRT-MP$_r$ and RRT-MP$_a$) were able to construct motion plans with s-ratio$= 100$ % for almost all B_i cases of broken modules. The RRT-MP$_m$ strategy creates a successful plan with s-ratio$= 100$ % for more than 70 % of combinations of failed modules of Lizard robot. The Quadropod robot is influenced by failures more than Lizard, especially if two or three modules are broken. In these cases, RRT-MP$_a$ is superior to other methods. This can be caused by the symmetrical shape of Quadropod, that is able to perform some of the predefined motions even under failures. The effect of failures to speed of motion is depicted in Fig. 9. Generally, the speed decreases with the increasing number of broken modules. Examples of constructed plans for Quadropod are depicted in Fig. 8.

Motion patterns influenced by failures are visualized in Fig. 11. The four predefined patterns can be easily performed by Quadropod and Lizard robots, but two of these primitives are not easy to achieve on the Dog robot (second column). When failures are introduced, Quadropod and Dog cannot be driven by the predefined primitives (column RRT-MP$_r$).

B. Primitive learning on HW robots

To use RRT-MP on physical robots, motion primitives have to be prepared. As the optimization of the primitives on physical robots can be time consuming, a simulation should be used to find suitable candidates, that are finalized on the real robots. Several CPGs were tested in [12], and the simple sine-based CPG was used for HW experiments as it provides suitable motions with less amount of parameters. The comparison of optimization run on physical robots from scratch, i.e., from random motions, and optimization initialized using best solutions from the simulation is depicted in Fig. 12. In this case, the quality of the primitives (cost) was measured as the traversed distance after 30 s of motion using a localization system [6]. The uninitialized optimization provided a solution with cost function ~ 20 cm after 80 iterations (~ 40 minutes), while the optimization initialized from the physical simulation provided a solution with cost ~ 35 cm after 25 iterations (~ 13 minutes).

RRT-MP algorithm was verified using the Quadropod modular robot (Fig. 13a) that was equipped with four motion primitives: 'move-left/right/ahead/back'. The task was to find a trajectory to a place ~ 2 m far away from start position (snapshot from the plan execution is depicted in Fig. 13b). The planning took ~ 5 s running on-board on Blackfin CPU, the execution of the plan took ~ 2 minutes. The video from the experiment is available at http://mrs.felk.cvut.cz/iros2015workshop.

VI. Conclusion

The paper describes a system to learn task-based motion primitives for modular robot. Contrary to the predefined mo-

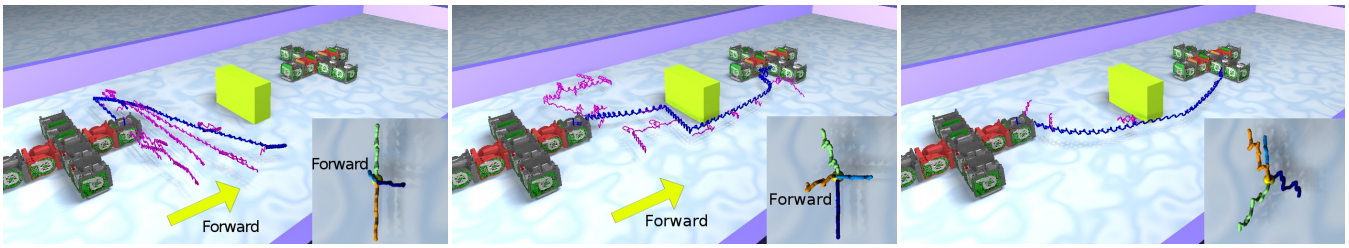


Fig. 8. Examples of configuration trees constructed using tested failure recovery strategies for Quadropod robot with two broken modules (broken modules are depicted in red). Robots are placed in start and goal configurations. RRT-MP$_r$ could not create a plan, as both 'move-forward' and 'move-back' primitives were almost disabled by the failures. RRT-MP$_a$ optimizes these primitives so the 'move-forward' primitive can be used in the plan.

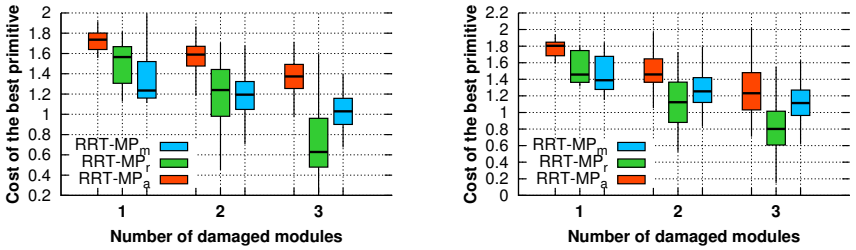


Fig. 9. Comparison of motion primitives used in the failure recovery strategies. The boxplots shows value of the cost function (speed of motion) of the best primitives for each configuration of broken modules. The value 1 equals to the size of one module.

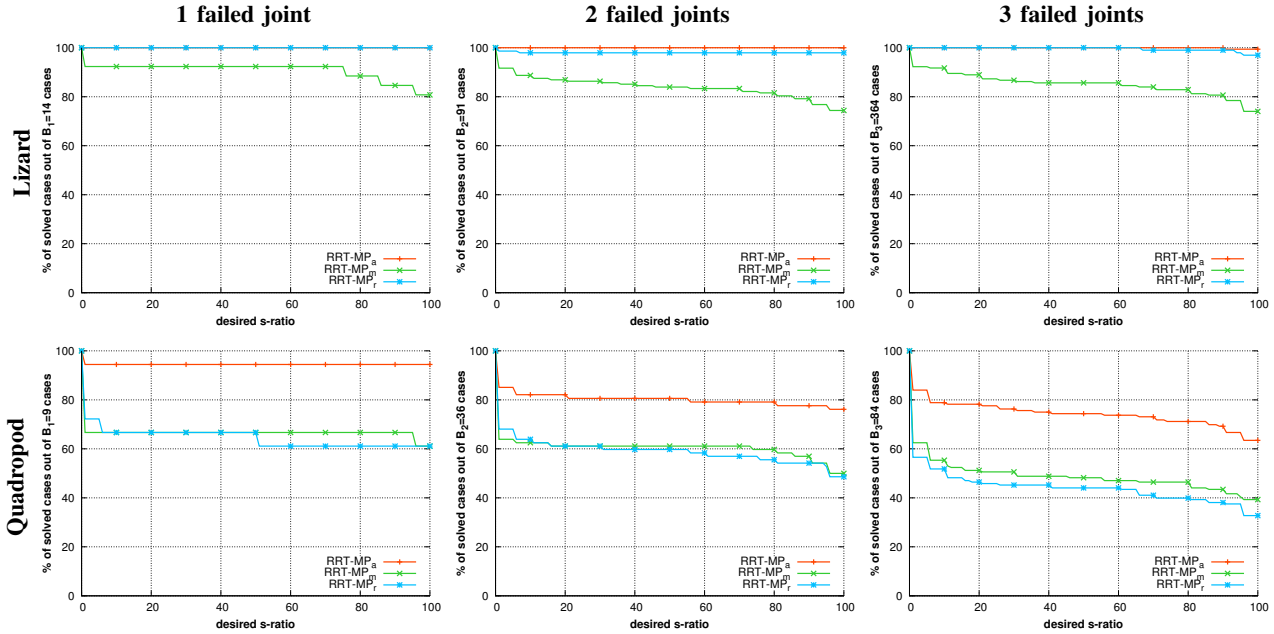


Fig. 10. Performance of the failure recovery strategies.

tion primitives, that need to be determined by a human operator before a mission considering capabilities of the robot, the task-based primitives are derived in a close cooperation with the motion planner. Several task-based primitives are derived simultaneously and their overall quality is determined by the quality of the generated motion plans. The advantage of the proposed system is the ability to derive motion primitives for unusual configurations of modular robots as well as for robots with failed modules.

The proposed system was verified in a failure recovery scenario. The experiments have shown that the pure adaptation of motion primitives considering the effect of broken modules is superior to both original primitives and the task-based primitives. A possible reason could be that the speed of task-based primitives was lower than the speed of adapted primitives and therefore the planner needs more iterations to find a solution.

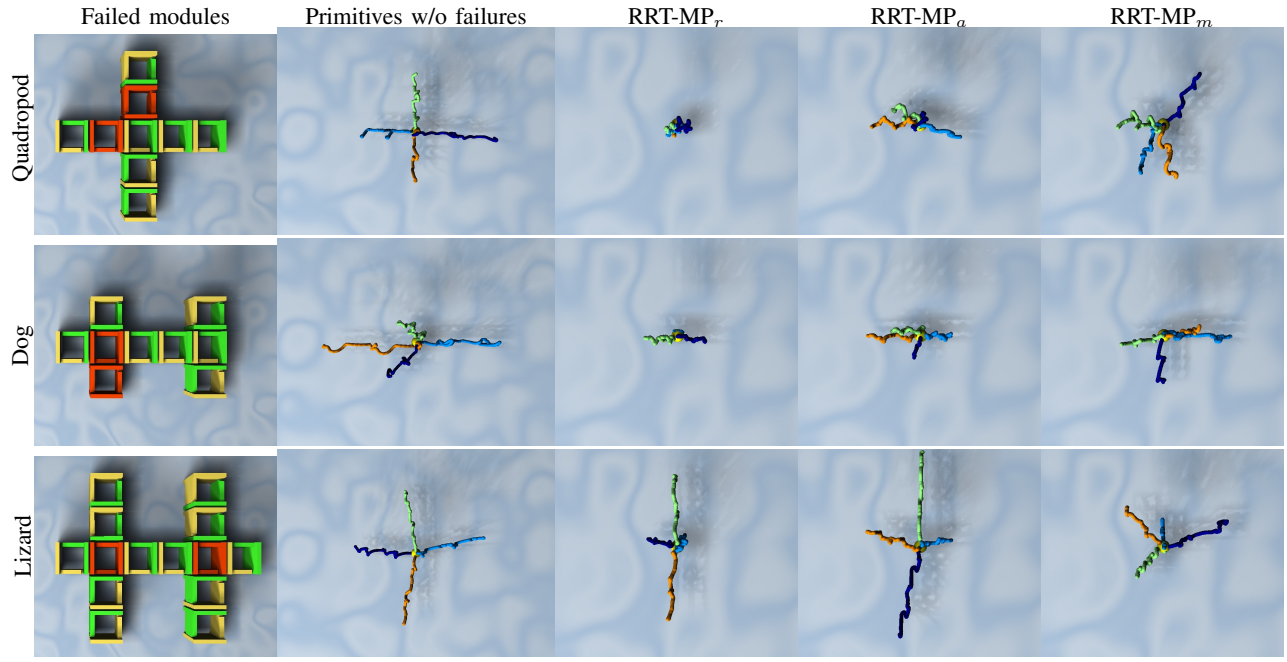


Fig. 11. Example of motion primitives for robot under failures. The first column shows configuration of broken modules (red), the second column shows trajectories of predefined primitives learned for a fully functional robot. The predefined primitives executed on failed robot are shown in the column RRT-MP$_r$ and the adapted primitives are in the column RRT-MP$_a$. Task-based primitives are shown in the last column.

REFERENCES

[1] Robert Fitch and Zack Butler. Million module march: Scalable locomotion for large self-reconfiguring robots. *International Journal of Robotic Research*, 27(3–4):331–343, 2008.

[2] A. J. Ijspeert. Central pattern generators for locomotion control in animals and robots: A review. *Neural Networks*, 21(4):642–653, 2008.

[3] A. Kamimura, H. Kurokawa, E. Toshida, K. Tomita, S. Murata, and S. Kokaji. Automatic locomotion pattern generation for modular robots. In *IEEE ICRA*, 2003.

[4] A. Kamimura, H. Kurokawa, E. Yoshida, K. Tomita, S. Kokaji, and S. Murata. Distributed adaptive locomotion by a modular robotic system, M-TRAN II. In *IEEE IROS*, 2004.

[5] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE International conference on Neural Networks*, 1995.

[6] T. Krajník, M. Nitsche, J. Faigl, P. Vaněk, Ma. Saska, L. Přeučil, T. Duckett, and M. Mejail. A practical multirobot localization system. *J. Intell. Robotics Syst.*, 76(3-4):539–562, December 2014.

[7] H. Kurokawa, E. Yoshida, K. Tomita, A. Kamimura, S. Murata, and S. Kokaji. Self-reconfigurable M-TRAN structures and walker generation. *Robotics and Autonomous Systems*, 54(2):142–149, 2006.

[8] J. Liedke, R. Matthias, L. Winkler, and H. Woern. The collective self-reconfigurable modular organism (CoSMO). In *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, 2013.

[9] Daniel Marbach and Auke Jan Ijspeert. Online optimization of modular robot locomotion. In *IEEE International Conference on Mechatronics and Automation*, 2005.

[10] P. Moubarak and P. Ben-Tzvi. Modular and reconfigurable mobile robotics. *Robotics and Autonomous Systems*, 60(12):1648–1663, 2012.

[11] T. Mulder, J. Duysens, and Henri W.A.A Van De Crommert. Neural control of locomotion: sensory control of the central pattern generator and its relation to treadmill training. *Gait & Posture*, 7(3):251–263, 1998.

[12] V. Vonásek, O. Penc, K. Košnar, and L. Přeučil. Optimization of Motion Primitives for High-Level Motion Planning of Modular Robots. In *Mobile Service Robotics: CLAWAR 2014: 17th International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines*, 2014.

[13] V. Vonásek, M. Saska, K. Košnar, and L. Přeučil. Global Motion Planning for Modular Robots with Local Motion Primitives. In *ICRA2013: Proceedings of 2013 IEEE International Conference on Robotics and Automation*, pages –, Piscataway, 2013. IEEE.

[14] Vojtech Vonasek, Sergej Neumann, David Oertel, and Heinz Worn. Online motion planning for failure recovery of modular robotic systems. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 1905–1910, May 2015.

[15] Vojtech Vonsek, Martin Saska, Lutz Winkler, and Libor Preucil. High-level motion planning for cpg-driven modular robots. *Robotics and Autonomous Systems*, 68(0):116 – 128, 2015.

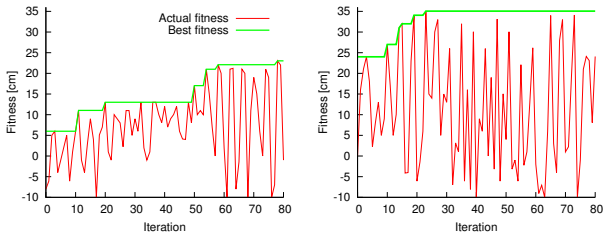
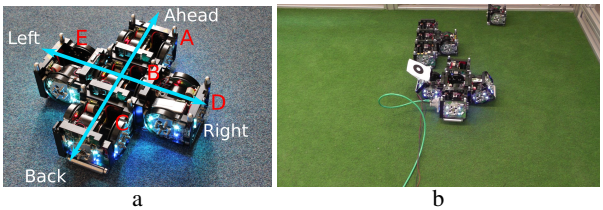


Fig. 12. Progress of cost function on a real robot in optimization run from scratch (a) and using initial solution from the simulation (b). In the latter case, the robot starts with initial solution with cost ~ 25 which is higher than the result of the optimization from scratch, which is ~ 23.



a b

Fig. 13. Configuration of the Quadropod robot in the HW scenario (a). Snapshot from the plan execution (b).