Auton Robot (2012) 32:149–164
DOI 10.1007/s10514-011-9268-6

Automated synthesis of action selection policies for unmanned vehicles operating in adverse environments

Petr Svec · Satyandra K. Gupta

Received: 23 September 2010 / Accepted: 30 November 2011 / Published online: 30 December 2011

Abstract We address the problem of automated action selection policy synthesis for unmanned vehicles operating in adverse environments. We introduce a new evolutionary computation-based approach using which an initial version of the policy is automatically generated and then gradually refined by detecting and fixing its shortcomings. The synthesis technique consists of the automated extraction of the vehicle's exception states and Genetic Programming (GP) for automated composition and optimization of corrective sequences of commands in the form of macro-actions to be applied locally.

The focus is specifically on automated synthesis of a policy for Unmanned Surface Vehicle (USV) to efficiently block the advancement of an intruder boat toward a valuable target. This task requires the USV to utilize reactive planning complemented by short-term forward planning to generate specific maneuvers for blocking. The intruder is human-competitive and exhibits a deceptive behavior so that the USV cannot exploit regularity in its attacking behavior.

We compared the performance of a hand-coded blocking policy to the performance of a policy that was automatically synthesized. Our results show that the performance of the automatically generated policy exceeds the performance of the hand-coded policy and thus demonstrates the feasibility of the proposed approach.

Keywords Artificial evolution · Policy synthesis · Unmanned vehicles · Unmanned surface vehicle (USV)

P. Svec (✉)
Simulation Based System Design Laboratory, Department of Mechanical Engineering, University of Maryland, College Park, MD 20742, USA
e-mail: petrsvec@umd.edu

S.K. Gupta
Simulation Based System Design Laboratory, Maryland Robotics Center, Department of Mechanical Engineering and Institute for Systems Research, University of Maryland, College Park, MD 20742, USA
e-mail: skgupta@umd.edu

1 Introduction

Manual development of a truly robust robotic system operating in an environment with an adversary exhibiting a deceptive behavior is a challenge. This scenario is typical for combat mission tasks where even a single mistake in the decision of the unmanned vehicle can have fatal consequences. In such scenarios, the vehicle has to be prepared to rapidly execute specialized maneuvers in addition to its default strategy in specific situations as defined by its control algorithm, or action selection policy to successfully accomplish its task. The difficulty in the development of such a policy consists in manual handling of the vehicle's failure states that arise in the encountered environments. This requires intensive repeatable testing of the overall vehicle's behavior using a large suite of different test scenarios, identifying the shortcomings, and implementing various contingency-handling behaviors (Baker et al. 2008).

In this article, we introduce a new approach for automated synthesis of an action selection policy for unmanned vehicles operating in a continuous state-action space. This approach can be viewed as an iterative synthesis process during which an initial version of the policy is automatically generated and then gradually improved by detecting and fixing those shortcomings that have a high potential of causing various task failures. The presented technique belongs to the class of evolutionary methods that directly search for the policy (Whiteson 2010), as opposed to computing a value function (Sutton and Barto 1998). In contrast to the majority of the direct policy search methods, our technique utilizes a dedicated local search procedure that finds specific additional macro-actions (Theocharous and Kaelbling 2004) (in the form of action plans or maneuvers) allowing the vehicle to preemptively avoid the failure states that cause substantial decrease in the total performance of its default policy.

The iterative detection of failure states and refinement of the policy helped us handle the large, continuous, and in large part fragmented (Kohl and Miikkulainen 2008) state space of the considered reinforcement learning problem. The fracture of the state space presents a significant challenge for standard evolutionary algorithms (Floreano and Mattiussi 2008) to evolve a well-performing policy, since the vehicle may be required to execute very different actions as it moves from one state to another. By explicitly detecting the failure states, we are able to identify the subspaces of the state space that possess the characteristics of high fracture that hampers the generalization of policy actions. Once the failure states are identified using multiple evaluation runs, they are traced back in time to find exception states from which new macro-actions can be executed. The use of macro-actions in contrast to using primitive actions simplifies and speeds up the synthesis, as there is no need to generate primitive actions for a greater number of states to successfully handle the failure states. The new action plans are readily incorporated into the policy, since they operate over a set of locally bounded and overlapping state space regions. The technique takes advantage of the fact that a large proportion of the state space is not encountered during the actual policy execution, so that the most critical failure states are always handled first. This is similar to the idea of the Prioritized Sweeping technique (Andre et al. 1998), using which the computation of the value function is focused on important states according to some criteria. We use Genetic Programming (GP) (Koza 2003) as a discovery component of the macro-actions to be applied locally in the exception states. During the whole process, no external human input on how the policy should be synthesized is therefore needed.

The policy is internally represented as a composite of one default high-level controller and a set of specialized action plans. The default controller is used to control the vehicle's behavior in all states except the states for which specific macro-actions in the form of action plans are needed. The action space of the vehicle is represented by a set of primitive commands, each having continuous parameters. The commands are combined, parametrized, and composed into a structure by the synthesis process to perform the overall task. The inherent symbolic representation of the policy greatly simplifies the analysis of its behavior. In addition, the symbolic representation allows integrating human knowledge and the analysis of the policy can provide the basis for improving the code.

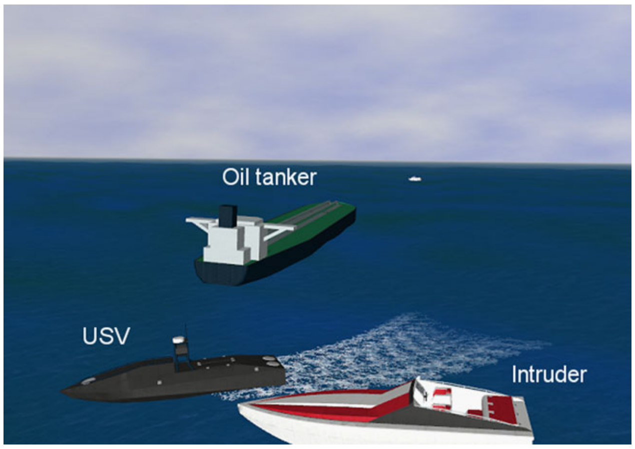


Fig. 1 Virtual environment: Unmanned Surface Vehicle (USV) is protecting an oil tanker by blocking the advancement of an intruder boat

Our approach was tested in the context of a large project aimed at the development of a general mission planning system (Schwartz et al. 2009; Svec et al. 2010) for automated synthesis of action selection policies for Unmanned Surface Vehicles (USVs) (Corfield and Young 2006; Finn and Scheding 2010). In this paper, we specifically focus on automated synthesis of a policy used for blocking the advancement of an intruder boat toward a valuable target (see Fig. 1). This task requires the USV to utilize reactive planning complemented by short-term forward planning to generate local action plans describing specific maneuvers for the USV. The intruder is human-competitive in the sense that its attacking efficiency approaches the attacking efficiency of deceptive strategies exhibited by human operators. Our aim is to reach the level 3 of autonomy as defined in Board (2005). In this level, the unmanned vehicle automatically executes mission-related commands when response times are too short for operator intervention.

An overview of the overall approach is shown in Fig. 2. First, we developed a physics-based meta-model using a detailed dynamics model of the USV to be able to test the policy in a simulation environment in real-time (Thakur and Gupta 2011). Second, we developed a mission planning system that contains a policy synthesis module (see Sect. 5). The necessary system architecture of the USV including the policy and state representation is described in Sect. 4. In order to combine the elements of the project into a cohesive system, we designed a USV simulation environment (Svec et al. 2010). The USV simulation environment integrates various components of the project into a complete simulation system and acts as a simulation platform for the synthesis module. One of the components of the simulation environment is the virtual environment (VE) based simulator (see Fig. 1) which serves as an emulator of the real USV environment that allows human players to play against each other or against the computer. Finally, we present an exper-

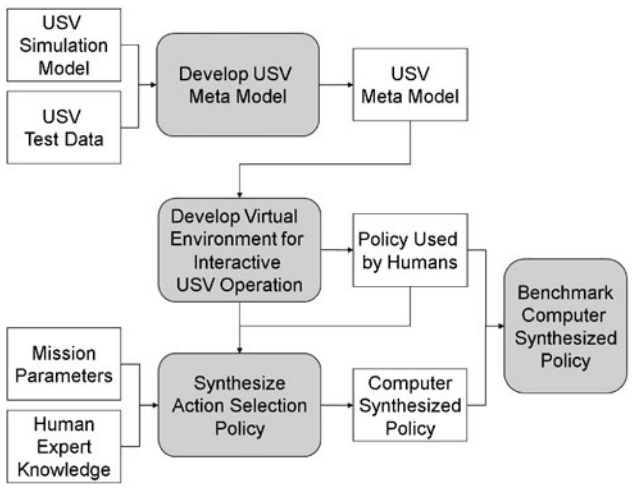

USV Simulation Model
USV Test Data
Develop USV Meta Model
USV Meta Model
Develop Virtual Environment for Interactive USV Operation
Policy Used by Humans
Benchmark Computer Synthesized Policy
Mission Parameters
Human Expert Knowledge
Synthesize Action Selection Policy
Computer Synthesized Policy

Fig. 2 Overview of the overall approach for synthesis of an action selection policy for USVs

imental evaluation of the approach in a challenging simulated combat-like scenario to demonstrate the feasibility of the proposed approach (see Sect. 6).

2 Related work

Computational synthesis (Lipson et al. 2003) deals with the problem of how to automatically compose and parametrize a set of functional building blocks into a hierarchical solution structure with the desired functionality. This is in contrast to classical optimization, in which the number and structure of modules and parameters being optimized is known in advance.

Evolutionary Robotics (ER) (Floreano and Mattiussi 2008) is a methodology that uses evolutionary algorithms to automatically synthesize controllers and body configuration for autonomous robots. As opposed to the use of standard temporal difference methods to approximate a value function (Sutton and Barto 1998), artificial evolution searches directly for a policy that maps states to actions.

In the literature, there are many successful applications of evolutionary techniques to robot controller synthesis. In many cases, the representation of the evolved controllers is a neural network. In the domain of neuroevolution, a popular method is the Neuroevolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen 2002), which was successfully applied to many controller synthesis problems. The main issue with the neural network representation, however, is the difficulty of analyzing the evolved solutions.

Besides the possibility of using methods for rules extraction from the evolved neural networks (Diederich et al. 2010), controllers can also be directly synthesized in a symbolic form. One of the techniques used to generate symbolic controllers is Genetic Programming (GP) (Koza 2003). GP as one of the robust evolutionary techniques has been used for automatically generating computer programs that usually have a tree structure and are generated using an algorithm similar to the traditional genetic algorithm (GA) (Goldberg 1989). Most of the controllers were successfully evolved for a wide variety of behaviors, such as obstacle avoidance (Barate and Manzanera 2007; Nehmzow 2002), wall-following (Dain 1998), line following (Dupuis and Parizeau 2006), light seeking, robot seeking (Nehmzow 2002), box pushing (Koza and Rice 1992), vision-driven navigation (Gajda and Krawiec 2008), homing and circling (Barlow and Oh 2008), predator versus prey strategies (Haynes and Sen 1996), co-evolution of control and bodies morphologies (Buason et al. 2005), game playing (Jaskowski et al. 2008; Togelius et al. 2007; Doherty and O'Riordan 2006) or group control for survey missions (Richards et al. 2005). GP was also utilized for the automated synthesis of human-competitive strategies for robotic tanks run in a closed simulation area to fight other human-designed tanks in international leagues (Shichel et al. 2005). There was also some progress on development of limited machine intelligence for classical strategic games like backgammon or chess endgames (Sipper et al. 2007).

However, for high-level strategy problems, e.g. the keep-away soccer (Kohl and Miikkulainen 2008), the discontinuous structure of the state-action space prohibits the standard evolutionary algorithms from generating good solutions. The monolithicity of the evolved controllers limit their usage for complex fractured domains (Kohl and Miikkulainen 2008), in which the best actions of the robot can radically change as it is moving continuously from one state to another.

In the domain of symbolic controllers, the Learning Classifier System (LCS) (Bacardit et al. 2008; Lanzi 2008; Urbanowicz and Moore 2009) can be considered one of the best options to cope with the fractured state-action space problem in robot learning. LCS represents a comprehensive class of evolutionary systems that solve reinforcement learning problems by evolving a set of classifiers or rules responsible for handling different local parts of input spaces of the problems. In LCS, the genetic algorithm searches for an adequate decomposition of the problem into a set of subproblems by evolving classifier conditions. This is similar to our approach; however, we explicitly decompose the problem by iteratively detecting states in which the robot exhibits deficiencies in fulfilling its task. We then find new states from which synthesized action plans can be executed to avoid the failure states. We use GP as a discovery component for the action part of the classifiers.

In the neuroevolution domain, there are examples of methods used for synthesizing different behaviors for different circumstances (so-called multi-modal behaviors) (Schrum and Miikkulainen 2009) arising in continuous state-action spaces. However, it is difficult to learn such behaviors reliably and without extensive human intervention, e.g. as is necessary in the case of the layered evolution combined with the subsumption architecture (van Hoorn et al. 2009). Moreover, the number of behaviors to evolve is usually determined in advance by a human designer. In most applications, the behaviors are learned and the control mechanism is hand-coded, or the control mechanism is learned and the behaviors are hand-coded. In some cases, both behaviors and the control mechanism are learned, but separately and with extensive human help.

Recently, neuroevolutionary methods have been developed that discover multi-modal behaviors automatically and do not depend on knowledge of the task hierarchy. For example, a special mutation operator was proposed in Schrum and Miikkulainen (2009) for evolving special output nodes that control the mode of the behavior. However, the potential number of modes needed to be evolved can be very high for some tasks. Another work (Kohl and Miikkulainen 2008) extends NEAT to use radial basis function (RBF) (Buhmann 2001) nodes to evolve controllers with better performance for complex problems. The above mentioned neuroevolutionary approaches make localized changes to the policy by directly modifying the structure of a neural network. In our case, we (1) represent the policy as a decision tree, and (2) indirectly modify the policy by evolving additional components that are activated in explicitly detected sub-regions of the state space, as opposed to directly modifying the structure of the tree by the use of special mutation operators or nodes.

In summary, our work is related to the above mentioned approaches by making localized changes to the policy using evolutionary search. However, in contrast to LCS or evolution of multi-modal behaviors in the domain of neuroevolution, we have developed a dedicated technique for incrementally finding and repairing functional shortcomings in the policy. Using this technique, we can identify critical subspaces of the state space in which the vehicle exhibits large deficiencies in fulfilling its task. We employ GP for evolution of the policy components represented as macro-actions to avoid the failure states from a special set of exception states. This allows us to overcome the inherent fracture and magnitude of the state space of the presented reinforcement learning problem.

3 Problem formulation

We are interested in synthesizing action selection policies suitable for solving sequential decision tasks that have highly fragmented state spaces (Kohl and Miikkulainen 2008) and require high reactivity in planning. The policies should thus allow the unmanned vehicles to quickly respond to the current state of the environment without any long-term reasoning about what actions to execute. In addition, the policies need to be represented symbolically in order to simplify their audit.

The task for the synthesis is to find a policy $\pi : S \to A$ that maps macro-actions A (also known as temporally extended actions that last for one or more time steps N) (Sammut and Webb 2011) to states S, such that the expected total discounted reward r for the task is maximized. The total reward r is computed as cumulative reward by taking into account rewards r_{s_t,a_t} of individual macro-actions $a_t \in A$ taken from states $s_t \in S$. The policy π consists of one default policy π_D and a set $\{\pi_i\}_{i=1}^{n}$ of n additional specialized policy components $\pi_i : S_i \to A$ that map macro-actions A to a specific set of overlapping regions $\Pi = \{S_1, S_2, \dots, S_n\}$ of the state space S.

We define a failure state $s_F \in S$ as a state in which the vehicle acquires a large negative reward due to violation of one or more hard C_h or soft C_s task constraints during the execution of a given task. The violation of even a single hard task constraint $c_h \in C_h$ (e.g., hitting an obstacle) guarantees that the vehicle will not be able to recover from its failure, and thus will not successfully finish its task. On the other hand, the violation of a soft task constraint C_s (e.g., losing a chance to block the intruder for a small amount of time) also results in a negative reward but does not terminate the task execution, since the vehicle can compensate for this type of task constraint violation in further planning. In both cases, the vehicle can avoid the violation of the constraints by executing appropriate pre-emptive macro-actions A from a special set of representative exception states $S_{E,REP} \subset S$.

More formally, there exist m sequences

$$P = \{p_i | p_i = (s_{i,1}, a_{i,1}, s_{i,2}, a_{i,2}, \dots, s_{i,l-1}, a_{i,l-1}, s_{F,i,l})\}$$

of states and atomic actions (as special cases of macro-actions with the length of one time step) representing transition paths in the state-action space that gradually attract the vehicle to failure states

$$S_F = \{s_{F,1}, s_{F,2}, \dots, s_{F,m}\} \subset S$$

if no additional macro-actions A are executed. We define a state of exception $s_{E,i}$ for a path p_i as the most suitable state from which a corrective macro-action $a_c \in A$ can be executed to deviate p_i from that state towards the states that result in the highest achievable expected reward. The state of exception $s_{E,i}$ can be found by searching in reverse from its corresponding failure state $s_{F,i}$ in p_i, which is equivalent to going back in time. Further, we define S_E as a set of exception states for all paths P leading to failures, and $S_{E,REP}$ as a representative set of close-by exception states from which the greatest number of failure states can be avoided.

The above formulation presented a general policy synthesis problem. Now, we will define a special case for the blocking task. In this case, the task is to automatically synthesize an action selection policy $\pi_U : S \rightarrow A_U$ for a USV to block the advancement of an intruder boat toward a particular target. The policy π_U should maximize the expected total reward r expressed as the maximum pure time delay the USV can impose on the intruder. The intruder executes a policy $\pi_I : S \rightarrow A_I$ that exhibits a deceptive attacking behavior. This prevents the USV from exploiting regularity in the intruder's actions. For this task, the USV needs to utilize reactive planning to be able to immediately respond to the current pose of the intruder boat by executing appropriate blocking maneuvers. The blocking maneuvers implement macro-actions A_U as defined by our framework. In this particular case, we define a failure state s_F as a state in which the intruder is closer to the target than the USV. The defined failure state thus covers both the soft and hard task constraints violations.

4 USV system architecture

The developed policy synthesis approach is closely coupled to the underlying system architecture of the USV. This architecture consists of several modules that are responsible for different tasks, e.g. sensing, localization, navigation, planning, behavior control, communication, human interaction, or monitoring (Corfield and Young 2006; Finn and Scheding 2010).

The USV utilizes a reactive controller with short-term forward planning for quick execution of maneuvers to be able to immediately respond to the state of the environment. This is in contrast to using a purely planning controller that deliberates about what sequence of actions to execute. The reasoning process of this type of controller would consume considerable amount of time since the planner would need to compute a number of candidate sequences of actions, evaluate each of them, and choose the one with the highest utility. In this work, we use the term short-term forward planning to emphasize the fact that the vehicle's action selection policy produces a sequence of time-varying actions (i.e., in the form of macro-actions) without any deliberation (as it is in the case of the purely planning controller), as opposed to directly executing individual primitive actions in each vehicle's state. This speeds up and simplifies the policy synthesis (see Sect. 5) since there is no need to repeatedly generate multiple primitive actions for a single representative exception state to successfully handle the corresponding failure state. The macro-actions cover a larger subset of the state-action space and thus simplify and increase the speed of the policy synthesis process.

The high simulation speed of the USV's dynamics model is critical for policy synthesis and therefore we use its simplified version with 3 degrees of freedom. This simplified

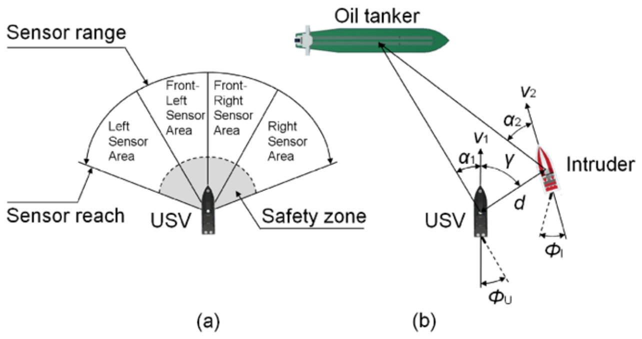


Fig. 3 (**a**) USV visibility sensor model, (**b**) USV's state in respect to the intruder and target

model has been adapted from the full-blown 6 degrees of freedom USV dynamics model as described in Thakur and Gupta (2011). The full-blown model considers ocean disturbances and is used for high-fidelity physics-based real-time simulations inside the virtual environment.

4.1 Virtual sensor models

The planning system needs to utilize only relevant key sensory information abstracted from the raw sensor data. This information is represented as a vector of features of different types (e.g., numerical or boolean) required for a successful fulfillment of the mission task. The values of the features are computed by a predefined set of virtual sensors (LaValle 2009) that process raw sensory data. The features can have one or more parameters using which their values are computed (e.g., features of the boolean type).

The planning system uses virtual visibility, relational, and velocity sensors. The virtual visibility sensor is a detection sensor with cone-shaped detection regions (see Fig. 3a). The size of the overall sensor area is defined by its reach and range parameters. Each region returns a boolean value expressing the presence of other objects and a normalized distance to the closest object. The relational virtual sensor provides relevant information on how other objects are situated with respect to the USV or to each other. It computes boolean values to be stored as values of the relational features. The velocity virtual sensor returns the velocities of other objects inside the environment.

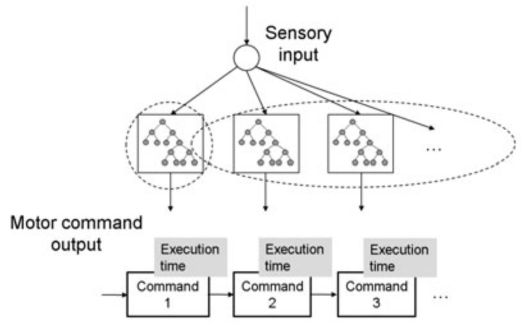
4.2 Planning architecture

4.2.1 USV state definition

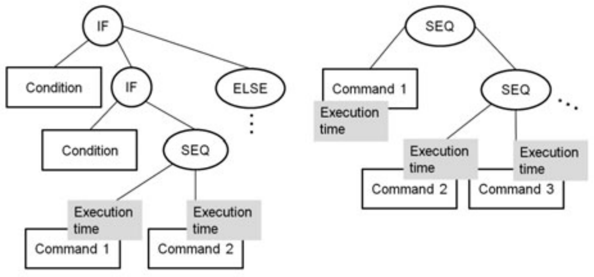
The full state of the USV (see Fig. 3b) is represented by an 8-dimensional vector $s = [\alpha_1, \alpha_2, \phi_U, \phi_I, v_1, v_2, \gamma, d]$ of state variables that encode attributes of the environment for the task. The angle α_1 represents the angle between the USV's heading and the direction to the target, α_2 is the angle between the intruder's heading and the direction to the target,

Table 1 Policy primitives

NC	go-intruder-front (front-left, front-right, left, right)
	turn-left (right)
	go-straight
CV	intruder-on-the-left (right, front, front-left, front-right, at-the-back, at-the-back-left, at-the-back-right)
	intruder-has-target-on-the-left (right)
	usv-has-target-on-the-left (right)
	usv-intruder-distance-le-than
	usv-closer-to-target-than-intruder
	usv-facing-intruder
BVO	if, true, false, and, or, not
PB	seq2, seq3
SC	usv-sensor, usv-velocity, usv-match-intruder-velocity



(a) Overall policy that consists of one default controller and a set of action plans. The product of the execution is a sequence of action commands (bottom)



(b) Example of a default controller (left) and an action plan (right)

Fig. 4 Policy representation

ϕ_U is the USV's steering angle, ϕ_I is the intruder's steering angle, v_1 is the USV's translational velocity, v_2 is the intruder's translational velocity, γ is the angle between the USV's heading and the direction to the intruder, and d is the distance between the USV and the intruder.

4.2.2 *Policy representation*

The policy π_U allows the USV to make a decision from a set of allowable actions based on the observed world state. The default policy π_D is represented as a high-level controller (see Fig. 4b left). This controller is made by a decision tree that consists of high-level parametrized navigation commands *NC*, conditional variables *CV*, standard boolean values and operators *BVO*, program blocks *PB*, and system commands *SC* (see Table 1). The leaves of the decision tree can be conditional variables, boolean values, navigation commands, or system commands. The inner nodes can be boolean operators or program blocks.

The additional policy components are represented as action plans that are sequences of parametrized navigation *NC* and system *SC* commands, and can be grouped into program blocks *PB*. The action plans are internally represented as trees so that the same synthesis mechanism used for generating the default controller can also be used for generating the plans. The leaves of the tree can be navigation or system commands.

The navigation and system commands have parameters, for example, the positional navigation commands (e.g., *go-intruder-front*) are defined using five parameters, where the first two parameters represent the USV's relative goal position (in polar coordinates) around the intruder. This effectively allows the vehicle to cover all feasible positions, as defined by its policy in a certain area around the intruder. The next two parameters represent a cone-shaped blocking area around the relative goal position. Once the vehicle gets inside the blocking area, it starts slowing down to limit the intruder's movement. The last parameter represents the length of the command execution. The turning left/right action commands have two parameters that represent the desired steering angle and the length of the command execution. The translational velocity of the vehicle is explicitly controlled by the velocity commands, where the *usv-velocity* command sets an absolute velocity given as a parameter, whereas the command *usv-match-intruder-velocity* sets a velocity given as a parameter relatively to the current velocity of the intruder. The *usv-sensor* system command changes the parameters of the virtual visibility sensor and thus allows it to explicitly control the risk level of the obstacle avoidance behavior (see Sect. 4.2.5 for further details).

4.2.3 *Policy execution*

During the mission, the USV periodically senses its surroundings and classifies its current state with respect to the intruder and the target. The classification mechanism of the policy executor decides whether the current state is close enough to one of the states for which a corresponding action plan exists. If such a plan exists, the policy executor directly executes the plan, otherwise it executes the default controller to generate a new sequence of actions. The decision whether to execute a specific plan depends on the activation distance parameter δ. This parameter defines the minimal distance that has to be achieved between the current USV's state and any state in the predefined set to activate a specific plan. The state space $S = S_1 \cup S_2$ is thus divided into two regions where in the first region S_1 the USV generates and executes plans using the default controller, whereas in the other region S_2 the USV directly executes previously generated or manually defined plans. The distance between normalized states is computed using the standard Euclidean distance metric.

The input into the policy is data from the virtual sensors that compute the values of all conditional variables. So, for example, the boolean value of the *intruder-on-the-left* variable is directly provided by the virtual relation sensor while the value of the *intruder-velocity-le-than* parametrized variable is provided by the virtual velocity sensor.

4.2.4 Planning and control architecture

By acquiring and processing sensor data in short-term cycles, and planning, the planning and control system (see Fig. 5) determines an action command to be executed to direct the vehicle inside the environment. The policy executor of the system takes as inputs sensor data, mission parameters, the USV meta model, and the action selection policy. It decides which component of the policy to execute to generate an action plan based on the current state of the vehicle. The plan consists of a number of action commands, each being executed for a certain amount of time. The ultimate output of an activated command is a sequence of motion goals $G = \{g_1, \dots, g_n\}$, $g_i = [x, y, \theta, v, t]$, where $[x, y, \theta]$ is the desired pose of the vehicle, v is the velocity, and t is the time of arrival. The motion goals are directly processed by the trajectory planner. The computed trajectories are consequently executed by a low-level controller to generate corresponding motor commands for device drivers of a particular actuator.

The planning and control architecture consists of planning, behavior, trajectory planning, and low-level control layers (see Fig. 5). The planning layer is responsible for interpreting the stored policy, which results in a number of commands queued for execution. The command processor inputs a command from the queue and either activates a corresponding behavior that interprets the command (e.g., in case of navigation commands), or modifies the properties of the trajectory planner (e.g., in case of system commands) in order to change the velocity or sensitivity of the obstacle avoidance mechanism of the vehicle. The commands are usually planned for short-term execution, such as planning of strategic maneuvers. The length of the execution is defined as a parameter of the command. The policy executor remains inactive until all the commands from the queue are processed, in which case the command processor requests new commands from the policy executor.

Each navigation command corresponds to the behavior selected by the behavior selector in the behavior layer. The behaviors produce sequences of motion goals when executed by the behavior executor. In the current architecture, the behaviors are *turn-left/right*, *go-straight*, and *go-to-intruder*. The trajectory planner receives a motion goal from the behavior executor and computes a feasible trajectory for the vehicle to reach the pose defined by the motion goal with a predefined velocity and within the given time.

The behavior executor is also responsible for monitoring execution of the trajectory and handling exceptions. An exception occurs if the vehicle loses its chance to successfully reach the motion goal as defined by the corresponding behavior. In that case, the command queue is emptied and new commands are supplied by executing the policy.

By default, the behaviors always choose a translational velocity that maximizes the USV's performance. So for example, the *go-straight* behavior uses maximum translational velocity to get to the requested position in the shortest amount of time. The policy can override this velocity by calling the *usv-velocity* or *usv-match-intruder-velocity* system commands.

4.2.5 Obstacle avoidance

The action plans executed by the behavior layer (that produces motion goals) and trajectory planning (that processes the motion goals) can be always overridden by the underlying reactive OA mechanism. The planning layer, however, can explicitly control the balance between a safe and risky avoidance behavior of the boat by modifying the parameters of the OA mechanism. These parameters particularly define how much steering should be applied in a close vicinity to an obstacle positioned at a certain distance and angle, and what should be the velocity of the vehicle in that situation. Hence, for our mission task, the resulting OA mechanism can be quite different with different parameters essentially controlling the vehicle's safe distance from the adversary and blocking efficiency at the same time. Insufficient avoidance steering can lead to collisions. On the other hand, too much steering will veer the vehicle away from the adversary, leading to ineffective blocking.

The reactive OA component of the trajectory planner uses the virtual visibility sensor (see Fig. 3a) in order to identify the locations of detectable obstacles. It directly produces the desired translational velocity v_d and steering angle $s_{a,d}$ to safely steer the vehicle away from the closest identified obstacles. The desired steering angle increases with the proximity to the obstacles while the translational velocity decreases. The desired steering angle is computed as $s_{a,d} =$

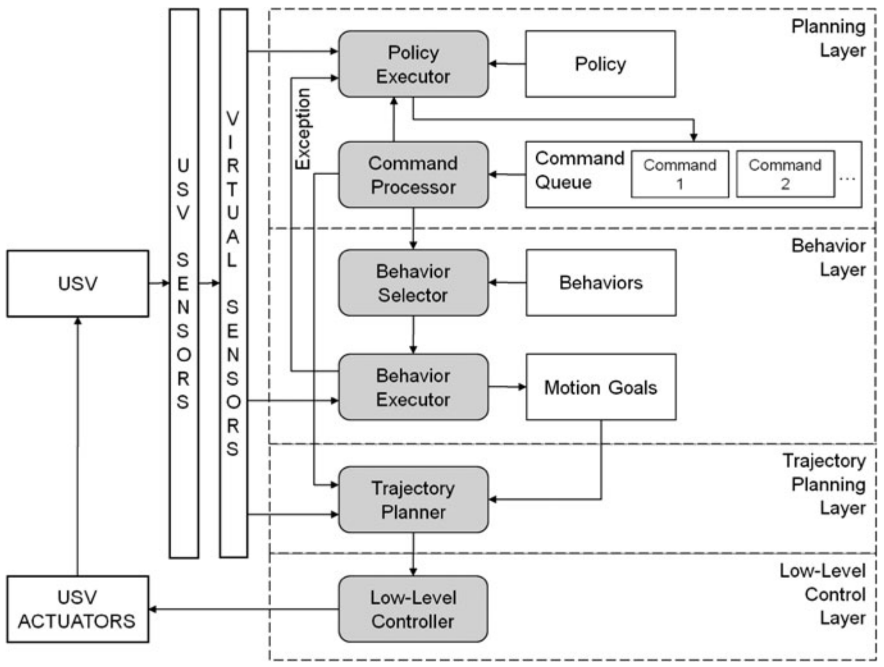


Fig. 5 USV planning and control architecture

$ks_{a,max}$, where $s_{a,max}$ is the maximum allowable steering angle for a particular steering direction and k is a scaling factor. The scaling factor is computed as $k = 1 - (d/s_{r,max})$, where d is the minimum distance to an obstacle in a particular visibility region around the USV and $s_{r,max}$ is the maximum reach of the virtual visibility sensor. The sensitivity of the OA mechanism can thus be indirectly modified by changing the reach and range parameters of the visibility sensor. In this way, the USV can get closer to obstacles than it would otherwise be possible and thus effectively define a balance between safe and aggressive maneuvering. The command used for setting the reach and range parameters of the virtual visibility sensor cones is the *usv-sensor*. The reach and range parameters can be modified by the synthesis process as described in Sect. 5.2.

5 Approach

In this section, we describe our approach to policy synthesis, which belongs to the class of evolutionary techniques (Whiteson 2010) that directly search for the policy in contrast to computing a value function (Sutton and Barto 1998). In addition to the evolution of a default policy, the presented iterative technique utilizes a specific local evolutionary search that finds additional macro-actions using which the vehicle can avoid frequent task execution failures.

The overall process is completely automated and starts by the synthesis of a default version of the policy π_D (see Fig. 6). The policy is then gradually refined by detecting failure states in which the vehicle has a high probability of failing in its task, and generating new pre-emptive action plans incorporated into this policy to avoid the failure states in future planning.

The synthesis technique exploits the fact that a large proportion of the state space is not frequently encountered during the actual policy execution. This observation together with the use of the default policy makes it possible to partially handle the combinatorial state explosion (Floreano and Mattiussi 2008) by evolving locally bounded optimal actions for only specific subsets of the continuous state space. This is further supported by generating macro-actions of variable size (defined by the number of commands and their execution time) in contrast to generating primitive actions, which would require identification of a substantial greater number of exception states to successfully handle the detected failure states. Hence, this results in faster synthesis as there is no need to generate primitive actions for every single state to handle the failure states.

The actual algorithm iteratively computes a stationary policy $\pi_U : S \to A_U$ that attempts to maximize the expected accumulated reward. The main steps of the algorithm are as follows (for a diagram depicting the whole process see Fig. 6):

1. Employ GP to evolve an initial version π_D of the policy π_U. The initial version may consists of a hand-coded portion to speed up the synthesis process.
2. Evaluate π_U inside the simulator using m distinct evaluation test cases T. The evaluation returns a set of failure states $S_F = \{s_{F,1}, s_{F,2}, \ldots, s_{F,n}\}$, $n \leq m$ in which the ve-

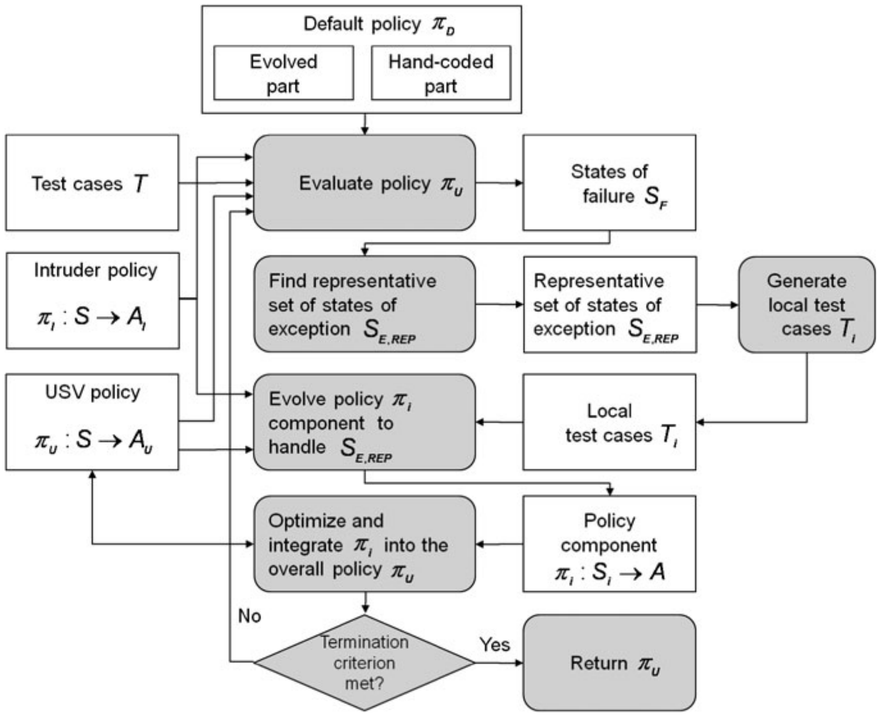

Default policy π_D
Evolved part
Hand-coded part
Test cases T
Evaluate policy π_U
States of failure S_F
Intruder policy $\pi_I : S \to A_I$
Find representative set of states of exception $S_{E,REP}$
Representative set of states of exception $S_{E,REP}$
Generate local test cases T_i
USV policy $\pi_U : S \to A_U$
Evolve policy π_i component to handle $S_{E,REP}$
Local test cases T_i
Optimize and integrate π_i into the overall policy π_U
Policy component $\pi_i : S_i \to A$
No
Termination criterion met?
Yes
Return π_U

Fig. 6 Policy synthesis overview

hicle fails its mission task. Each test case $t \in T$ defines the initial states of the vehicles and determines the maximum number of time steps N for evaluation of the policy.
3. Given S_F, find a corresponding set of exception states $S_E = \{s_{E,1}, s_{E,2}, \dots, s_{E,n}\}$ in whose proximity (given by the activation distance parameter δ) the vehicle has the potential to avoid the failure states S_F if it executes appropriate action plans.
4. Extract a representative set of exception states $S_{E,REP} \subseteq S_E$, as described in detail in Sect. 5.1, in which the vehicle has the potential to avoid the largest number of the detected failure states. In this way, the focus is always restricted to the most critical failure states first while leaving the rest for possible later processing.
5. Generate a set of local test cases T_i to be used during the evolution of a new action plan π_i as described in the next step. Each test case $t \in T_i$ encapsulates a corresponding exception state and determines the maximum number of time steps $N_i \leq N$ for evaluation of the plan.
6. Employ GP to evolve a new action plan $\pi_i : S_i \to A$ for the representative set of exception states $S_{E,REP}$. This prevents the over-specialization of the plan by evaluating its performance using a sample of states from this set (presented as the test cases in T_i).
7. Optimize the new plan π_i and integrate it into the policy π_U. If the termination condition is not satisfied, continue to step 2. The distance between the normalized states is computed using the Euclidean distance metric.

5.1 Extraction of a representative set of exception states

A diagram that describes the extraction process of representative exception states is shown in Fig. 7. First, the algorithm finds corresponding exception states S_E for given failure states S_F by reverting back in time for a predefined number of time steps N_τ.

Second, given S_E, the algorithm iterates over all exception states $s_E \in S_E$ and for each of them finds its neighboring states $S_{E,\delta}$ within the activation distance δ. Then, for s_E and the states from its immediate neighborhood $S_{E,\delta}$, the algorithm finds corresponding failure states $S_{F,E}$ together with all their neighbors $S_{F,E,\delta}$ within the distance δ. The algorithm terminates by returning the representative set of exception states $S_{E,REP}$ that is associated with the largest number of corresponding failure states $S_{F,E} \cup S_{F,E,\delta}$. The set $S_{E,REP}$ consists of a representative exception state $s_{E,REP}$ and its immediate neighboring states $S_{E,REP,\delta}$ within the distance δ.

Figure 8 shows an example of a detected representative set of exception states $S_{E,REP}$ (marked by the indicated set of circles) and their corresponding failure states (marked as crosses) connected by relational links. The states are projected to 2D plane by multidimensional scaling that uses the Kruskal's normalized stress criterion (Cox and Cox 2008).

In the context of our mission task, a failure state s_F defines a situation in which the intruder is closer to the target than the USV. Its corresponding exception state s_E is found by reverting back in time for a predefined number of time

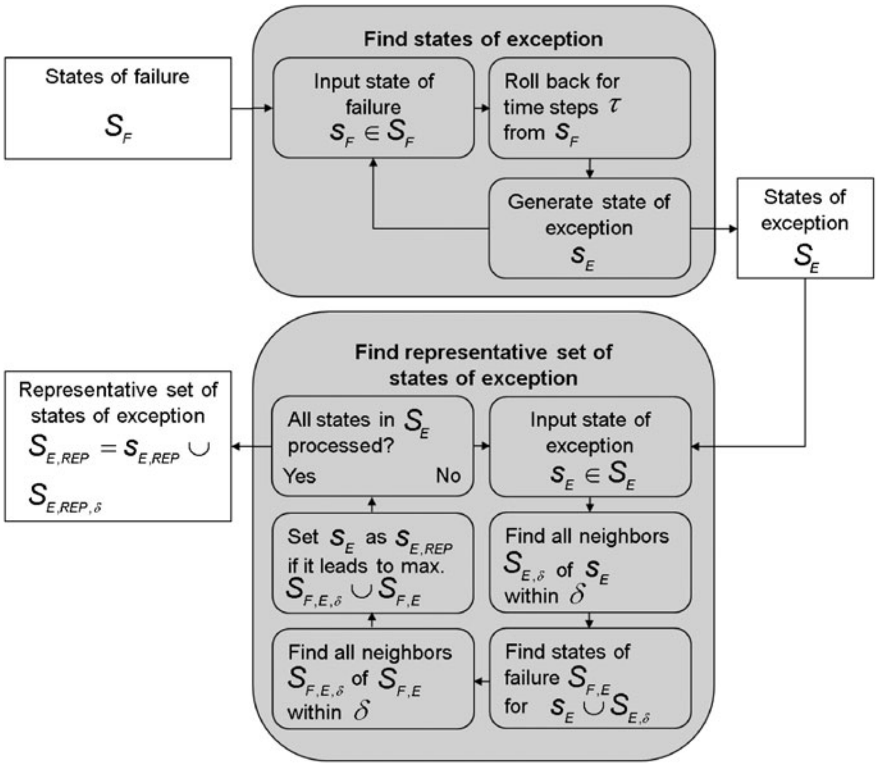

States of failure
S_F
Find states of exception
Input state of failure $s_F \in S_F$
Roll back for time steps τ from s_F
Generate state of exception s_E
States of exception S_E
Find representative set of states of exception
All states in S_E processed?
Yes
No
Input state of exception $s_E \in S_E$
Find all neighbors $S_{E,\delta}$ of s_E within δ
Find states of failure $S_{F,E}$ for $s_E \cup S_{E,\delta}$
Find all neighbors $S_{F,E,\delta}$ of $S_{F,E}$ within δ
Set s_E as $s_{E,REP}$ if it leads to max. $S_{F,E,\delta} \cup S_{F,E}$
Representative set of states of exception $S_{E,REP} = S_{E,REP} \cup S_{E,REP,\delta}$

Fig. 7 Extraction of a representative set of exception states $S_{E,REP}$

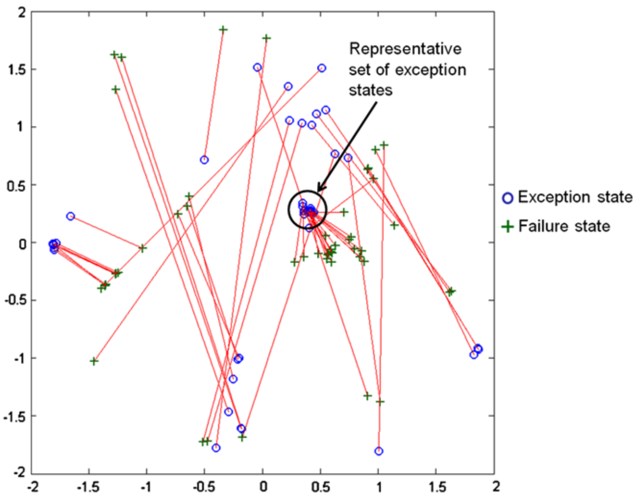


Fig. 8 Representative set of exception states. The failure states (marked as *crosses*) and exception states (marked as *circles*) are projected to 2D plane using multidimensional scaling with the Kruskal's normalized stress criterion. The links between each pair of states express the failure-exception state bindings

steps N_τ to record a state from which a new specific action plan can be executed to prevent a possible future failure. The simple way of determining s_E can be further improved by developing a special task-related heuristic that precisely determines a state from which the vehicle will have the highest chance of successfully avoiding the largest number of possible failure states.

5.2 Evolution

The evolution searches through the space of possible plan configurations to find the best possible action plan for a particular state. Both the default policy π_D (represented as a controller) and specialized action plans $\{\pi_i\}_{i=1}^n$ as components of the policy are automatically generated using separate evolutionary processes. The specific evolutionary method we used is the strongly-typed Genetic Programming imposing type constraints on the generated Lisp trees (Poli et al. 2008). This is a robust stochastic optimization method that searches a large space of candidate program trees while looking for the one with the best performance (so-called the fitness value).

The evolutionary process starts by randomly generating an initial population of individuals represented as GP trees using the Ramped half-and-half method (Poli et al. 2008). The initial values of parameters of all action commands and conditionals are either seeded or randomly generated. The default controller π_D of the policy π_U is generated using a human-written template for which GP supplies basic blocking logic. The first portion of the template encodes a maneuver using which the vehicle effectively approaches the intruder at the beginning of the run, as there is no need for it to be explicitly evolved.

The terminal and function sets T and F consist of action commands, system commands, conditional variables, boolean values and operators, and program blocks as defined in Sect. 4.2.2. The sets are defined as

$$T_{default\text{-}controller} = T_{plan} = NC \cup SC$$

$$F_{default\text{-}controller} = CV \cup BVO \cup PB; \qquad F_{plan} = PB$$

Within the population, each individual has a different structure responsible for different ways of responding to its environment. The individual plans are evaluated in the context of the whole policy inside the simulator. The sensory-motor coupling of the individual influences the vehicle's behavior, resulting in a specific fitness value that represents how well the USV blocks the intruder.

We favor individuals which can rapidly establish basic blocking capabilities and optimize them to push the intruder away from the target over the entire trial duration. To do so, the fitness f is defined as the sum of normalized squared distances of the USV from the target over all time steps. If a collision occurs, either caused by the USV or the intruder, the zero fitness value is assigned to the individual, and the selection pressure eliminates the policy component with low-safety guarantee. The fitness function is as follows:

$$f = \frac{1}{N}\sum_{t=1}^{N}\left(\frac{d_i}{d_0}\right)^2 \qquad (1)$$

where N is the total number of time steps, d_i is the distance of the intruder from the target at time step t, and d_0 is the initial distance of the intruder from the target in a particular test scenario. The total fitness value of the individual is maximized and computed as an average of the fitness values resulting from all scenarios.

The default controller π_D is evaluated using a hand-coded human-competitive intruder in 8 different scenarios. In each scenario, the intruder has a different initial orientation, and the USV always starts from an initial position closer to the target. The evaluation lasts for a maximum number of time steps equal to 300 seconds in real time. The maximum speed of the USV is set to be 10% higher than the speed of the intruder, but other properties of the vehicles are the same. The action plan is evaluated using a sample of states from $S_{E,REP}$ found within the activation distance δ of its corresponding $s_{E,REP}$. The evaluation lasts for a maximum number of time steps equal to 10 seconds in real time.

The individuals in the initial population mostly exhibit a random behavior. By selecting and further refining the individuals with high fitness, their quality gradually improves in subsequent generations. During this process, the individuals are randomly recombined, mutated, or directly propagated to the next generation. These operations are applied with the predefined probabilities (see Table 2). The following evolutionary operators are used:

1. Reproduction—copies one individual directly to the next generation without any modification. We use a strong elitism to propagate the best individual directly into the next generation. This makes sure that the best individual is not lost during the evolutionary process as a consequence of recombination.
2. Mutation—we use three types of mutation operators: structural mutation of a randomly selected sub-tree, preventing bloat (Poli et al. 2008) by shrinking a randomly chosen sub-tree to a single node, and Gaussian mutation of chosen parameters.
3. Crossover—randomly selects sub-trees from two input trees and swaps them.

Table 2 GP parameters

Population size/ number of generations	500/100 (controller) 50/30 (plan)
Tournament size	2
Elite set size	1
Min. and max. initial GP tree depth	3 and 6 (controller) 2 and 4 (plan)
Maximum GP tree depth	30 (controller) 10 (plan)
Reproduction prob.	0.1
Crossover prob.	0.84
Structure mutation prob.	0.05
Shrink structure mutation prob.	0.01
Mutation prob. of parameters of action commands	0.5

During the policy synthesis, the USV learns the balance between safe and dangerous maneuvering by mutating the reach and range parameters of its virtual visibility sensor. The policy is thus co-evolved with the sensor parameters of the vehicle to control the obstacle avoidance mechanism.

The optimization of the generated default controller π_D removes all branches of the code that have not been executed during evaluation scenarios. Moreover, each generated action plan π_i is truncated to contain only the action commands that do not exceed the maximum execution time of the plan.

A detailed description of the functionality of all the operators used can be found in Koza (2003). The control parameters of the evolutionary process used for evolution of the default controller and plans are summarized in Table 2.

6 Computational experiments

6.1 General setup

We tested the developed approach in the context of a combat mission task during which the USV protects a valuable target against an intruder boat. In this task, the intruder boat has to reach the target as quickly as possible while the USV has to block and delay the intruder for as long a time as possible. We set up an experiment to compare the performance of the automatically generated USV's policy to the USV's policy coded by hand. We compare the performance in terms of pure time delay imposed by the USV on the intruder. To get a fair assessment of the USV performance, the time values being compared must be normalized by a 40-second baseline. This baseline represents the amount of time needed to reach the target if the intruder is completely unobstructed. Any additional time above this baseline thus represents the effective delay time of the intruder when being blocked by the USV.

The policy of the USV is evaluated in 800 runs to account for the intruder's nondeterministic behavior. Each evaluation run lasts for a maximum number of time steps equal to 300 seconds in real time. The dimension of the scene is 800×800 m with the target positioned in the center. At the beginning of each run, the USV and the intruder are oriented toward each other with a random deviation of up to 10 degrees and the USV is positioned on a straight line between the intruder and the target. The initial distance of the USV from the target is approximately 240 m while the intruder's initial distance is 360 m. The USV's maximum velocity is 10 m/s while the intruder's maximum velocity is 9 m/s.

6.2 Experimental protocol

First, we manually implemented an initial version of the intruder's attacking policy and tested it against human players to evaluate its performance in the virtual environment. The policy was further improved in multiple iterations over a period of 6 weeks. Its overall size reached 485 lines of Lisp code. The outline of the policy functionality is described in the next section. We evaluated the performance of the policy by pitting human players against it playing as USVs. The human players achieved 90 seconds of pure time delay on average imposed on the intruder. This shows that the intruder's attacking policy is quite sophisticated as it exhibits a deceptive behavior. If the intruder's behavior was not deceptive, the human players would have been able to quickly find a motion pattern in the behavior that could be exploited for "indefinite" blocking, and thus not useful for the synthesis.

Second, we implemented the USV's blocking policy against the hand-coded intruder. The policy was improved iteratively over a period of 3 weeks. Its overall size reached 500 lines of Lisp code. The main difficulty when implementing the policy by hand was the manual identification of the most critical cases (or exceptions) in which the policy had a low performance. This is generally non-trivial and requires substantial human effort.

Third, we used the mission planning system to automatically generate the USV's blocking policy using the hand-coded intruder as the competitor. The activation distance parameter δ was set to 0.2 for all action plans. In the current version of the approach, a failure state is determined to be the state in which the intruder is closer to the target than the USV. An exception state is computed by going back in time for 150 time steps from a given failure state.

Finally, we compared the performance of the automatically synthesized USV's policy to the hand-coded one.

6.3 Intruder's policy

The design of the intruder's attacking policy was a crucial step during the initial stages of the experimental setup. The level of aggressiveness of the intruder's attacking behavior is defined such that the intruder presents a high challenge for human players playing as USVs but at the same time executes relatively safe obstacle avoidance.

The evaluation scenario is nondeterministic, i.e., the intruder may execute different actions in the same state that may lead to different outcomes. The nondeterminism of actions the intruder can execute thus allows it to repeatedly deceive the USV during the combat so that the USV is not able to find a motion pattern in the intruder's behavior that can be easily exploited for blocking (as would be the case of deterministic intruder's policy).

The nondeterminism of the intruder's behavior poses a great challenge to the synthesis of local USV policies since the same policies may acquire different fitness values when evaluated in the same test scenario. Thus, for the purpose of fair evaluation, we use one specific type of the intruder's policy for computing the fitness values of all USV individuals (represented as local policies) from one specific population. The behavior expressed by the intruder is influenced by a set of randomized action commands that are parts of its policy. The randomized commands take a random number as one of their inputs, based on which they produce an action. The random number is generated by a random number generator that is initiated by an explicitly provided random seed. The random seed thus indirectly defines a particular type of the intruder's policy and is kept the same during evaluation of individuals within the same population.

The hand-coded intruder's policy is represented as a single decision tree that contains standard action commands as well as their randomized versions. The intruder's policy is divided into five main sections. Each of these sections handles a different group of situations that can arise during the combat. The first section handles situations in which the distance of the intruder from the target is greater than 130 m and the angle between its translation direction and the target is more than 80 degrees. In these situations, the intruder attempts to rapidly change its direction of movement toward the target by aggressively turning left or right, depending on the relative position of the USV.

The second section handles situations in which the USV is very close to the intruder, positioned relatively to its front left, and the target is on the intruder's left side (see Fig. 9a left). In this case, the intruder has two options. Either it executes a random turn with probability 0.9 or it proceeds with a complete turn. In both cases, the intruder can slow down rapidly with probability 0.3 to further confuse the adversary. This section also handles a symmetric type of situations when the USV is on the front right of the intruder and the target is on the right.

The logic of the third section is very similar to the logic of the second section with the exception that it handles the situations when the USV is directly on the left or right side of the intruder (see Fig. 9a right). In these cases, the intruder deceives the USV by randomly slowing down to get an advantageous position, proceeding with a complete turn, or executing a partial turn. The probability of the complete turn is 0.1 and the probability of slowing down is 0.2.

The fourth section deals with situations in which the intruder is positioned behind the USV inside the rear grey area as shown in Fig. 9b left. The greater distance of the intruder from the USV gives it opportunity to exploit the USV's tendency to overshoot a little in the process of blocking. In this case, if the USV has high velocity, the intruder slows down and turns toward the stern of the blocking USV, passing the USV from behind. Otherwise, the intruder randomly turns with probability 0.7 or it proceeds with a complete turn (see Fig. 9b right). Again, the intruder can rapidly slow down with probability 0.3.

Finally, if the intruder is not in close proximity to the USV, it computes the best sequence of actions in order to get to the target as quickly as possible.

The intruder's policy modifies the reach and range parameters of its virtual visibility sensor to indirectly control the balance between a safe and aggressive obstacle avoidance mechanism. For example, if the intruder wants to make an aggressive turn in close proximity to the USV, it has to take risk by decreasing the reach of the sensor to be able to quickly proceed with the turn. In this case, the sensitivity of the obstacle avoidance behavior is reduced for a short period of time so that the intruder can easily pass the USV from behind. If the intruder always aimed to safely avoid the adversary, it would not get any chance to get to the target, especially if it competes against a human player.

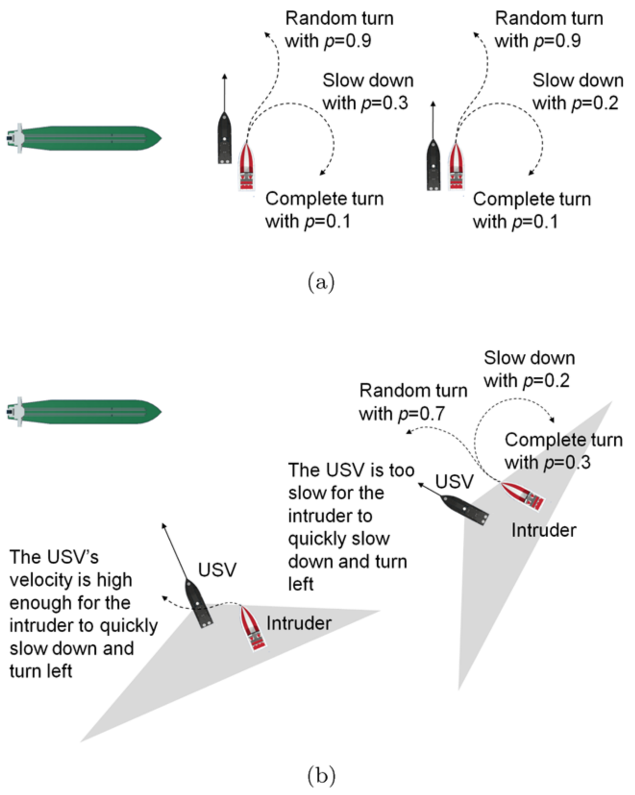

Random turn with p=0.9
Slow down with p=0.3
Complete turn with p=0.1
Random turn with p=0.9
Slow down with p=0.2
Complete turn with p=0.1
(a)
Slow down with p=0.2
Random turn with p=0.7
Complete turn with p=0.3
The USV is too slow for the intruder to quickly slow down and turn left
USV
Intruder
The USV's velocity is high enough for the intruder to quickly slow down and turn left
USV
Intruder
(b)

Fig. 9 Representative portions of intruder's policy

6.4 Results and discussion

The experimental run that generated the blocking policy with the highest performance is shown in Fig. 10. The horizontal axis of the graph shows different versions of the policy consisting of a gradually increasing number of action plans. The vertical axis shows the blocking performance in terms of the intruder's pure time delay for each version of the USV's policy. The best performance is reached by version 249 of the policy and amounts to 65 seconds of pure time delay on average and a median of 53 seconds. This can be compared to the pure time delay of 53 seconds on average and a median of 46 seconds imposed by the hand-coded USV on the same intruder. This result thus shows that the best performance of the automatically generated policy exceeds the blocking performance of the hand-coded policy.

The automated generation of the policy took approximately 1 day to generate the default controller and approximately 23 days on average to generate action plans for 300 automatically defined exception states on a machine with configuration Intel(R) Core(TM)2 Quad CPU, 2.83 GHz. From the set of 10 experimental runs, only 2 were able to find a policy with similar performance to the best one. The remaining 8 runs prematurely stagnated due to overspecialization of some of the evolved action plans and imprecise extraction of exception states. Even a single defective action plan synthesized for one of the key situations can significantly influence the performance of the whole policy. This shows that the synthesis of a policy for the USV to block the intruder utilizing a nondeterministic attacking policy is a challenging task.

The results show that the performance of the first few versions of the policy is low as they contain only a few action plans describing specialized maneuvers for a small number of key situations. However, as the synthesis process progresses, more action plans handling new situations are added and the overall performance gradually improves. This way the initial policy becomes sophisticated due to newly

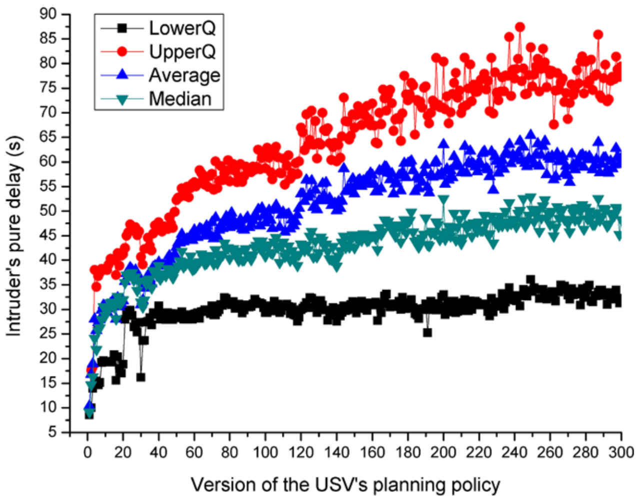


Fig. 10 Evaluation of the USV's blocking performance. The performance is expressed as a pure time delay imposed on the intruder. Each version of the USV's policy was evaluated in 800 runs

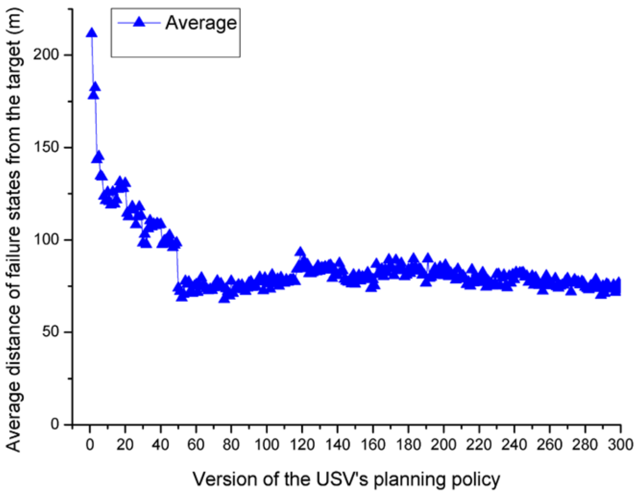


Fig. 11 The average distance of failure states from the target during the synthesis of the USV's policy

evolved action plans. The synthesis process continues until version 249 of the policy after which the performance stagnates. This can be attributed to difficulty in solving new complex situations in which problems with the generalization of action plans and correct detection of exception states arise.

The graph in Fig. 11 illustrates the average distance of failure states from the target during the synthesis process. It is shown that at the beginning of the synthesis (up to the version 47 of the policy), the failure states occur farther from the target, while most of them appear closer to the target at an average distance of 75 meters, where most intense combats happen.

Evolution of a single policy against an adversary exhibiting a nondeterministic behavior thus generates a highly sub-optimal solution. To further improve the performance of the policy, distinctive states are automatically isolated for which short-term action plans are generated. As a result of that, the final policy demonstrates a clear performance advantage over the policy without the high performing substitutes.

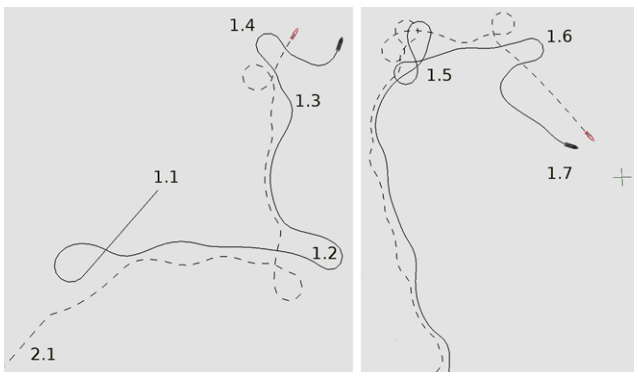


Fig. 12 Example of a run in which the USV managed to block the intruder for additional 45 seconds. The start position of the USV is marked as 1.1 while the start position of the intruder is marked as 2.1

An example of a run in which the USV reached 45 seconds of pure time delay imposed on the intruder is shown in Fig. 12. The USV starts at the location 1.1 while the intruder starts at the location 2.1. The first situation in which the USV executes a specific maneuver is marked as 1.2. In this situation, the USV steers sharply to the left in an attempt to intercept the intruder. The run continues until 1.3 where the USV attempts to deflect the intruder's heading by first carefully turning to the left and then aggressively blocking from the side. The intruder, however, instantly responds by executing a sharp left turn, which makes the USV attempt again to intercept him in the situation 1.4. Yet the USV overshoots in the process of blocking. The run continues for the next 23 seconds all the way up to the target. In the situation 1.5, the intruder executes a random sequence of two sharp turns to deceive the USV and thus to increase its chances for the attack. The USV, however, successfully follows and makes another attempt to intercept the intruder but overshoots in 1.6, and the intruder finally reaches its goal 1.7.

7 Conclusions and future work

We have introduced a new approach for automated action selection policy synthesis for unmanned vehicles operating in adverse environments. The main idea behind this approach is to generate an initial version of the policy first and then gradually improve its performance by evolving additional policy components. These components are in the form of macro-actions that allow the vehicle to pre-emptively avoid frequent task execution failures. The combination of (1) explicitly detecting subspaces of the state space that lead to frequent failure states, (2) discovering macro-actions as variable sequences of primitive actions for avoiding these failure states, and (3) having elaborated control and planning mechanisms, allowed us to successfully solve a non-trivial reinforcement learning problem.

Our particular focus was on the synthesis of a symbolic policy for an autonomous USV operating in a continuous state-action space with a deceptive adversary. We have developed a mission planning system to automatically generate a policy for the USV to block the advancement of an intruder boat toward a valuable target. The intruder is human-competitive and exhibits a deceptive behavior so that the USV cannot exploit regularity in its attacking action rules for blocking. The USV's policy consists of a high-level controller and multiple specialized action plans that allow the vehicle to rapidly execute sequences of high-level commands in the form of maneuvers in specific situations.

In our experiments, we have compared the performance of a hand-coded USV's blocking policy to the performance of the policy that was automatically generated. The results show that the performance of the synthesized policy (65 seconds of pure delay on average, a median of 53 seconds) exceeds the performance of the hand-coded policy (53 seconds on average, a median of 46 seconds). Hence, the approach described in this paper is viable for automatically synthesizing a symbolic policy to be used by autonomous unmanned vehicles for various tasks.

We have validated the presented approach on a specific task; however, it can be applied to a broad range of other tasks as well. The type of tasks for which the presented approach is applicable is described in Sect. 3. Examples of other application domains include variations on the pursuit and evasion task during which a robotic vehicle attempts to keep track of another moving object operating in a certain area (Gerkey et al. 2006), and robotic soccer (Kohl and Miikkulainen 2008).

Besides deploying the presented approach to other application domains, we would like to exploit the regularity (e.g., repetitiveness, symmetry, and symmetry with modifications) (Lipson 2007) of the tasks in other to effectively reuse already evolved macro-actions with possible modifications for similar states. This will increase the speed of the synthesis process. Other ways of increasing the speed of the synthesis are the physics-based model simplification that allows fast simulations while trying to preserve the underlying physics of the vehicle-terrain simulated interactions (Thakur and Gupta 2011), and utilization of distributed and parallel computing on dedicated computer clusters (e.g., using Berkeley Open Infrastructure for Network Computing) (Anderson 2004). Moreover, the sub-regions of the state space with associated macro-actions currently have a fixed size. It would be beneficial, however, if these regions could be of different shapes with a variable size. This would allow the evolution of macro-actions of greater complexity for better generalization of the policy. Finally, an interesting possibility for further research would be the development of a general methodology for efficient detection of failure states and computation of adequate exception states by exploiting the structure of a given task.

Acknowledgements This research has been supported by Office of Naval Research N00014-10-1-0585. Opinions expressed in this paper are those of the authors and do not necessarily reflect opinions of the sponsors.

References

Anderson, D. (2004). Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM international workshop on grid computing* (pp. 4–10). IEEE Computer Society: Los Alamitos.

Andre, D., Friedman, N., & Parr, R. (1998). Generalized prioritized sweeping. *Advances in Neural Information Processing Systems*, 1001–1007.

Bacardit, J., Bernadó-Mansilla, E., & Butz, M. (2008). Learning classifier systems: looking back and glimpsing ahead. *Learning Classifier Systems*, 1–21.

Baker, C., Ferguson, D., & Dolan, J. (2008). Robust Mission Execution for Autonomous Urban Driving. *Intelligent Autonomous Systems*, *10*, 155.

Barate, R., & Manzanera, A. (2007). Automatic design of vision-based obstacle avoidance controllers using genetic programming. In *Proceedings of the evolution artificielle, 8th international conference on Artificial evolution* (pp. 25–36). Berlin: Springer.

Barlow, G., & Oh, C. (2008). Evolved navigation control for unmanned aerial vehicles. *Frontiers in Evolutionary Robotics*, 596–621.

Board, N. (2005). *Autonomous vehicles in support of naval operations*. Washington: National Research Council.

Buason, G., Bergfeldt, N., & Ziemke, T. (2005). Brains, bodies, and beyond: Competitive co-evolution of robot controllers, morphologies and environments. *Genetic Programming and Evolvable Machines*, *6*(1), 25–51.

Buhmann, M. (2001). Radial basis functions. *Acta Numerica*, *9*, 1–38.

Corfield, S., & Young, J. (2006). Unmanned surface vehicles–game changing technology for naval operations. *Advances in Unmanned Marine Vehicles*, 311–328.

Cox, M., & Cox, T. (2008). *Multidimensional scaling*. *Handbook of data visualization* (pp. 315–347).

Dain, R. (1998). Developing mobile robot wall-following algorithms using genetic programming. *Applied Intelligence*, *8*(1), 33–41.

Diederich, J., Tickle, A., & Geva, S. (2010). Quo vadis? Reliable and practical rule extraction from neural networks. *Advances in Machine Learning*, *I*, 479–490.

Doherty, D., & O'Riordan, C. (2006). Evolving agent-based team tactics for combative computer games. In *AICS 2006 17th Irish artificial intelligence and cognitive science conference*.

Dupuis, J., & Parizeau, M. (2006). Evolving a vision-based line-following robot controller. In *IEEE proceedings*.

Finn, A., & Scheding, S. (2010). *Developments and challenges for autonomous unmanned vehicles: a compendium*. Berlin: Springer.

Floreano, D., & Mattiussi, C. (2008). *Bio-inspired artificial intelligence: theories, methods, and technologies*.

Gajda, P., & Krawiec, K. (2008). Evolving a vision-driven robot controller for real-world indoor navigation. In *Proceedings of the 2008 conference on applications of evolutionary computing* (pp. 184–193). Berlin: Springer.

Gerkey, B., Thrun, S., & Gordon, G. (2006). Visibility-based pursuit-evasion with limited field of view. *The International Journal of Robotics Research*, *25*(4), 299.

Goldberg, D. (1989). *Genetic algorithms in search and optimization*.

Haynes, T., & Sen, S. (1996). Evolving behavioral strategies in predators and prey. *Adaption and Learning in Multi-Agent Systems*, 113–126.

Jaskowski, W., Krawiec, K., & Wieloch, B. (2008). Winning ant wars: Evolving a human-competitive game strategy using fitnessless selection. In *Genetic programming: Proceedings of 11th European conference, EuroGP 2008* (p. 13). Naples, Italy, 26–28 March 2008. New York: Springer.

Kohl, N., & Miikkulainen, R. (2008). Evolving neural networks for fractured domains. In *Proceedings of the 10th annual conference on genetic and evolutionary computation* (pp. 1405–1412). New York: ACM.

Koza, J. (2003). *Genetic programming IV: Routine human-competitive machine intelligence*. Dordrecht: Kluwer Academic.

Koza, J., & Rice, J. (1992). Automatic programming of robots using genetic programming. In *Proceedings of the national conference on artificial intelligence* (p. 194).

Lanzi, P. (2008). Learning classifier systems: then and now. *Evolutionary Intelligence*, *1*(1), 63–82.

LaValle, S. (2009). *Filtering and planning in information spaces* (IROS tutorial notes).

Lipson, H. (2007). Principles of modularity, regularity, and hierarchy for scalable systems. *Journal of Biological Physics and Chemistry*, *7*(4), 125.

Lipson, H., Antonsson, E., Koza, J., Bentley, P., & Michod, R. (2003). Computational synthesis: from basic building blocks to high level functionality. In *Proc. assoc. adv. artif. intell. symp.* (pp. 24–31).

Nehmzow, U. (2002). Physically embedded genetic algorithm learning in multi-robot scenarios: The PEGA algorithm. In *2nd international workshop on epigenetic robotics: modelling cognitive development in robotic systems*.

Poli, R., Langdon, W., & McPhee, N. (2008). *A field guide to genetic programming*. Lulu Enterprises UK Ltd.

Richards, M., Whitley, D., Beveridge, J., Mytkowicz, T., Nguyen, D., & Rome, D. (2005). Evolving cooperative strategies for UAV teams. In *Proceedings of the 2005 conference on genetic and evolutionary computation* (p. 1728). New York: ACM.

Sammut, C., & Webb, G. (2011). *Encyclopedia of machine learning*. New York: Springer.

Schrum, J., & Miikkulainen, R. (2009). Evolving multi-modal behavior in NPCs. In *Proceedings of the 5th international conference on computational intelligence and games* (pp. 325–332). New York: IEEE Press.

Schwartz, M., Svec, P., Thakur, A., & Gupta, S. K. (2009). Evaluation of automatically generated reactive planning logic for unmanned surface vehicles. In *Performance metrics for intelligent systems workshop (PERMIS'09)*.

Shichel, Y., Ziserman, E., & Sipper, M. (2005). GP-robocode: Using genetic programming to evolve robocode players. *Genetic Programming*, 143–154.

Sipper, M., Azaria, Y., Hauptman, A., & Shichel, Y. (2007). Designing an evolutionary strategizing machine for game playing and beyond. *IEEE Transactions on Systems, Man and Cybernetics. Part C, Applications and Reviews*, *37*(4), 583–593.

Stanley, K., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99–127.

Sutton, R., & Barto, A. (1998). *Reinforcement learning: an introduction. Adaptive computation and machine learning*. Cambridge: MIT Press.

Svec, P., Schwartz, M., Thakur, A., Anand, D. K., & Gupta, S. K. (2010). A simulation based framework for discovering planning logic for Unmanned Surface Vehicles. In *ASME engineering systems design and analysis conference*, Istanbul, Turkey.

Thakur, A., & Gupta, S. (2011). Real-time dynamics simulation of unmanned sea surface vehicle for virtual environments. *Journal of Computing and Information Science in Engineering*, *11*, 031005.

Theocharous, G., & Kaelbling, L. (2004). Approximate planning in pomdps with macro-actions. *Advances in Neural Information Processing Systems*, *16*.

Togelius, J., Burrow, P., & Lucas, S. (2007). Multi-population competitive co-evolution of car racing controllers. In *IEEE congress on evolutionary computation, CEC 2007* (pp. 4043–4050). New York: IEEE.

Urbanowicz, R., & Moore, J. (2009). Learning classifier systems: a complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*, *2009*, 1.

van Hoorn, N., Togelius, J., & Schmidhuber, J. (2009). Hierarchical controller learning in a first-person shooter. In *IEEE symposium on computational intelligence and games (CIG 2009)* (pp. 294–301).

Whiteson, S. (2010). *Adaptive representations for reinforcement learning* (Vol. 291). Berlin: Springer.

Petr Svec is an Assistant Research Scientist at the University of Maryland, College Park. His research interests are in the area of Robotics, Machine Learning, Motion Planning, Computational Geometry, and Graph Theory. Dr. Svec received a Master's degree in Computer Science from the Faculty of Mechanical Engineering at Brno University of Technology in 2003, followed by Ph.D. with a focus on robot motion planning from the same institute. He also received a second Master's degree in Computer Science in 2006 from Masaryk University. In addition, he spent one year as a Graduate Research Assistant at the Department of Computer Science at the University of Bristol in the UK.

Satyandra K. Gupta is a Professor in the Department of Mechanical Engineering and the Institute for Systems Research at the University of Maryland. He is also the director of Maryland Robotics Center. Prior to joining the University of Maryland, he was a Research Scientist in the Robotics Institute at Carnegie Mellon University. He received a Bachelor of Engineering (B.E.) degree in Mechanical Engineering from the University of Roorkee (presently known as the Indian Institute of Technology, Roorkee) in 1988. Dr. Gupta's research interest is broadly in the area of automation. He is specifically interested in automation problems arising in Design, Manufacturing, and Robotics. He has authored or coauthored more than two hundred articles in journals, conference proceedings, and book chapters. Dr. Gupta is a fellow of the American Society of Mechanical Engineers (ASME).