Engineering Approaches and Methods to Verify Software in Autonomous Systems

G. Cicala[2], A. Khalili[1 2], G. Metta[1], L. Natale[1], S. Pathak[1 2], L. Pulina[3], and A. Tacchella[2]

[1] iCub Facility, Istituto Italiano di Tecnologia (IIT), Via Morego, 30 – 16163 Genova – Italy
Giorgio.Metta@iit.it - Lorenzo.Natale@iit.it - Shashank.Pathak@iit.it
[2] DIBRIS, Università degli Studi di Genova, Via Opera Pia, 13 – 16145 Genova – Italy
Giuseppe.Cicala@unige.it - Ali.Khalili@edu.unige.it - Armando.Tacchella@unige.it
[3] POLCOMING, Università degli Studi di Sassari, Viale Mancini 5 – 07100 Sassari – Italy
lpulina@uniss.it

Abstract. We present three computer-augmented software engineering approaches to ensure dependability at different levels of control architectures in autonomous robots. For each approach, we outline the methodological framework, our current achievements, and open issues. Albeit our results are still preliminary, we believe that furthering research along these lines can provide cost-effective techniques to make autonomous robots safe and thus fit for commercial purposes.

Keywords: Dependable Control Architectures, Software Verification and Testing, Autonomous Robots.

1 Introduction

A great deal of current research is focused on making robots accomplish complex tasks in unstructured environments with increasing degrees of autonomy — see, e.g., [1, 2], and the impressive results obtained in the DARPA robotics challenge [3]. These operational scenarios require rich and complex control architectures, which are usually organized in several levels, from those closest to hardware, e.g., motor control loops, to those farthest from it, e.g., object recognition, manipulation, locomotion, speech and combinations thereof. Since robots must be operated safely, layered control architectures must be dependable, but this is a major challenge when autonomy clashes with basic safety requirements.

In this paper, we present three computer-augmented software engineering approaches aimed to improve on safety. These approaches are targeted to different levels and different kinds of components inside the control architecture, and they rely on different formal models and techniques. However, they share the fundamental vision that formal models can be automatically (i) extracted, (ii) analyzed and (iii) exploited to obtain additional confidence in the properties of the control architecture. Our basic philosophy is to keep the amount of additional developer's knowledge as small as possible, while at the same time ensuring a precise analysis about whether the architecture matches its requirements. The final goal is to analyze control architectures automatically with state-of-the-art verification techniques. In the remainder of this Section, we briefly describe the context, the motivation and the key ideas behind each of the approaches that we present.

Verification of Embedded Control Software Powerful embedded controllers are commonly adopted to enable the implementation of sophisticated planning and control strategies — see, e.g., [4]. The growing complexity of control strategies entails a growing complexity of embedded software and, in turn, a growing number of programming bugs. We would like to apply software model checking — see [5] for a recent survey — to ensure that bugs in control programs cannot drive the robot to unwanted states. This is challenging, because the correctness of the control software relies on properties of the controlled system. In [6] a methodology to enable embedded software model checking is introduced. The main idea is to apply system identification techniques to obtain a computational model of the physical system which can be checked together with the control software. In Section 2, we present an experience report along the lines of [6], where we consider the verification of an embedded control program in a two-wheeled self-balancing robot. The goal of the report is to highlight the current limitations of this methodology, and to propose further research to improve its feasibility and applicability.

Middleware identification Developing (formal) models can be difficult for "black-box" components, i.e., overly-complex, poorly-documented, or closed-source software. This is critical in middleware APIs used, e.g., to orchestrate uniform access to hardware resources. Automata-based identification techniques – see, e.g., [7] for a comprehensive list of references – enable the inference of the internal structure of a black-box component by analyzing its interactions with an embedding context. Learning algorithms supply the component with suitable input test patterns to populate a "conjecture" automaton by observing the corresponding outputs; then, they check whether the conjecture is behaviorally equivalent to the actual component. When an abstract model is obtained, it can be used as a stub to test and/or verify software components relying on it. Practical identification of middleware is enabled by our tool AIDE (Automata IDentification Engine) [8], an open-source software written in C#. In Section 3, we sketch the design and the implementation of AIDE, we show the results of an experiment about the identification of a YARP [9] component, and we provide an example to demonstrate how the identified models can support bug-finding in control software relying on YARP.

Safe Reinforcement Learning Reinforcement Learning (RL) [10] is one of the most widely adopted paradigms to obtain intelligent behavior from interactive robots. RL methods have shown robust and efficient learning on a variety of robot-control problems — see, e.g., [11]. However, "the asymptotic nature of guarantees about RL performances makes it difficult to bound the probability of damaging the controlled robot and/or the environment" [12]. How to guarantee that, given a control policy synthesized by RL, such policy will have a very low probability of yielding undesirable behaviors? Our answer leverages Probabilistic Model Checking techniques — see, e.g., [13] – by describing robot-environment interactions using Markov chains, and the related safety properties using probabilistic logic. Both the encoding of the interaction models and their verification are fully automated, and only the properties have to be manually specified. Our research goes beyond automating verification, to consider the problem of automating repair, i.e., if the policy is found unsatisfactory, it is fixed with no manual intervention. Further details about this approach and some experimental results are presented in Section 4.

2 Verification of Control Software

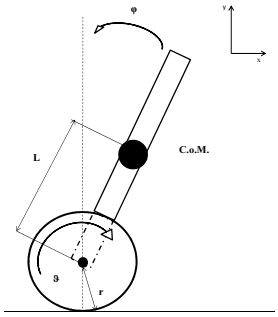

Fig. 1. Lumped-parameters model of a self-balancing two-wheeled robot.

As we mentioned in the Introduction, our approach amounts to consider programs as a language to model the whole system, i.e. ,the physical system plus its control software. The main advantage is that we can exploit the expressive power as well as the (formal) semantics of programming languages to describe the physical reality in a precise, yet developer-friendly way. Moreover, following [6], it is possible to apply software model checking to automate the verification of properties that may depend on the controlled system, and the discovery of bugs that arise because of the interaction between the control program and the physical system. As a case study, we consider a prototype of a self-balancing two-wheeled robot whose physical model is shown in Figure 1. This kind of robot provides a compelling test case for several reasons. Firstly, while the complexity of the mathematical modeling is relatively low, the control system must reach beyond the basic single-input, single-output (SISO) framework, to take into account multiple outputs — body angle φ and position x. Secondly, the control problem is a variation of the well-known inverted pendulum control problem; while inherently non-linear, the inverted pendulum can be linearized around its unstable equilibrium point so as to allow the design of linear regulators usings simple and robust techniques. Lastly, a system prototype is easy to build, requiring only two motors equipped with encoders, a gyroscope, a control board, plus some support hardware to build the inverted pendulum configuration.

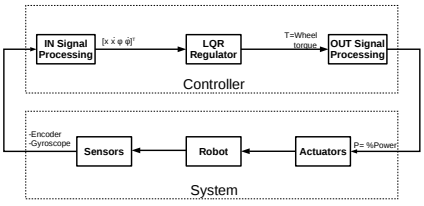

Fig. 2. Block diagram of our robot and its control software.

In more details, our prototype is built using a solid-state sensor to provide rate of rotation around the vertical axis y, i.e., the reading of the sensor is proportional to $\dot{\varphi}$. Two DC motors are connected to the drive wheels to provide torque to the axles for upright balancing and navigation of the vehicle. The motor speed is controlled using Pulse Width Modulation (PWM) with a fixed duty-cycle expressed as a percentage, from 0% (no power) to 100% (full power). The rotation angle θ is calculated using the encoders built in the motors, and from θ we reconstruct the horizontal position x using odometry. At $\varphi = 0$, the robot has its quasi-equilibrium state, so we can linearize the model under the assumption that the variation of φ is small enough to have small errors. Us-

```
W := wheel radius
I := moment of inertia
T := 10ms

MAIN()
 1 while (TRUE) do
 2   UPDATEGYRODATA()
 3   UPDATEMOTORDATA()
 4   u ← - K · x̂
 5   p ← TORQUETOPOWER(u)
 6   SETMOTORPOWER(p)

TORQUETOPOWER(torque)
 1   angAccel ← torque · 1/I
 2   angVelSum ← angVelSum +
                 angAccell · T
 3   angVel ← angVelSum
 4   return α · angVel + β

UPDATEGYRODATA()
 1   gyroRaw ← GETGYROREADING()
 2   gOffset ← G · gyroRaw + (1 − G) · gOffset
 3   gyroSpeed ← RPM2RAD(gyroRaw - gOffset)
 4   gAngleSum ← gAngleSum + gyroSpeed · T
 5   x̂[φ̇] ← gyroSpeed
 6   x̂[φ] ← gAngleSum

UPDATEMOTORDATA()
 1   encRaw = ← GETENCODERREADING()
 2   eOffset = E · encRaw + (1 − E) · eOffset
 3   mrc ← encRaw - eOffset
 4   mrcPrev ← mrcCurr
 5   mrcCurr ← mrc
 6   mrcDelta ← mrcCurr - mrcPrev
 7   xDelta ← W * RPM2RAD(mrcDelta)
 8   x̂[x] ← x̂[x] + xDelta
 9   x̂[ẋ] ← xDelta · 1/T
```

Fig. 3. Sketch of the main control loop, and accessory functions TORQUETOPOWER (left), UPDATEGYRODATA, UPDATEMOTORDATA (right). We assume that $\mathbf{K}$ is a global constant corresponding to the LQR weights, and $\hat{\mathbf{x}}$ is a global variable containing the current state estimate. The constants $G, E \in (0, 1)$ are parameters for the exponential moving averages used to correct drifts in sensor readings — line 2 in UPDATEGYRODATA and UPDATEMOTORDATA. α and β are constants defininng the linear relationship between desired motor velocity and PWM value – line 4 in TORQUETOPOWER. Underlined functions are those defined in the embedded controller API to interface with sensors and actuators.

ing this assumption, we can consider the dynamics of the robot in state space form as $\dot{\mathbf{x}} = A\mathbf{x} + Bu$, where $\mathbf{x} = [\theta, \dot{\theta}, x, \dot{x}]$ is the state vector, u is the input torque, and A and B are the matrices obtained considering physical parameters. We implement a Linear Quadratic Regulator (LQR) [14] to provide a stabilizing torque u given the state $\mathbf{x}$. The regulator provides proportional feedback in the form $u = -\mathbf{K}\mathbf{x}$, where $\mathbf{K}$ is the solution of the Algebraic Riccati Equation [15].

As shown in Figure 2 (left), the control software consists of three sequential tasks: input signal processing to provide an estimate $\hat{\mathbf{x}}$ of the state from sensor readings, torque computation, i.e., the scalar product between $\mathbf{K}$ and $\hat{\mathbf{x}}$, and output signal processing to convert the torque value to a suitable PWM control signal. The control software is developed in C and it runs on an embedded computer equipped with a 32-bit ARM7TDMI core, 256KB of FLASH memory, 64KB of RAM and an 8-bit Atmel AVR ATmega48 microcontroller. The duration of the whole control loop, including input and output signal processing, is approximately 10ms. The pseudo code of the main control loop is sketched in Figure 3, including MAIN control task, and accessory functions UPDATEGYRODATA, UPDATEMOTORDATA and TORQUETOPOWER. In more detail, the function UPDATEGYRODATA shown in Figure 3 (top-right) reads values from the gyroscope (line 1), corrects them for drifting due to, e.g., thermal noise (line 2), and finally it estimates the robot angular velocity $\dot{\varphi}$, and the pitch angle φ using numerical integration (line 4). The function UPDATEMOTORDATA shown in Figure 3 (bottom-right) reads data from motor encoders (line 1), corrects them (line 2), and it estimates (i) the linear displacement x using odometry (line 7), and (ii) the linear velocity $\dot{x}$ by numerical derivation (line 9). Both UPDATEGYRODATA and UPDATEMOTORDATA are implemented in the "IN Signal Processing" block depicted in Figure 2 (left). In line 4 of function MAIN, the torque value is computed based on equation $u = -\mathbf{K} \cdot \hat{\mathbf{x}}$ where u is the torque, $\mathbf{K}$ is the vector of LQR weights, and $\hat{\mathbf{x}}$ is the state vector estimate. The

function TORQUETOPOWER computes the PWM value to drive the actuators from the feedback torque value. This is obtained by considerig the relation between the *angular velocity* of the motors and the PWM value, which is linear as long as the motors are far from the saturation torque. In our setting, this is always the case, so we can compute the PWM signal by obtaining the desidered angular acceleration (line 1), then the desired angular velocity by numerical integration (lines 2,3), and applying an experimentally-derived linear relationship to obtain the PWM value. The function TORQUETOPOWER is implemented in the "OUT Signal Processing" block in Figure 2 (left).

Our goal is to provide a computational model of the physical system, i.e., a piece of code that can provide a realistic implementation of GETGYROREADING and GETENCODERREADING given the state of the system and the PWM feeded through SETMOTORPOWER. To this purpose, we consider the simplest form of time-domain system identification, i.e., AutoRegressive with eXogenous input (ARX) models. The ARX model has the following structure:

$$\sum_{k=0}^{N} a_k y(t-k) = \sum_{k=0}^{M} b_k u(t-k) + e(t) \tag{1}$$

where $y(t)$ is the output, $u(t)$ is the input, $e(t)$ is the error, M is the number of zeroes and N is the number of poles. Using the Z-transform, the transfer function from input to output can be written as

$$H(z) = \frac{Y(z)}{U(z)} = \frac{b_0 + b_1 z^{-1} + ... + b_M z^{-M}}{1 + a_1 z^{-1} + ... + a_N z^{-N}} = \frac{B(z)}{A(z)} \tag{2}$$

The goal of system identification thus becomes the extrapolation of the coefficients $a_1, \ldots, a_N$ and $b_0, \ldots, b_M$ from experimental input-output data. In our case, we resorted to the Matlab System Identification Toolbox [16] in order to experiment with various settings of M and N, and to extract a reasonable model of the robot around its unstable equilibrium point. Using the superposition principle, we considered two different systems to identify the gyroscope and the encoder given the PWM signal. After detrending and filtering the collected input-output data, we estimated a model with one zero and three poles for the encoder, and a model with one zero and four poles for the gyroscope. Both models have been implemented in C code as a direct form representation of IIR filters, which act as "stubs" to replace the original embedded controller API functions. Notice that this form of identification is very crude, in the sense that the system exhibits non-linear dynamics as soon as the pitch angle φ is not close to 0. An identificaton using, e.g., NARX models with recurrent Neural Networks would be more appropriate in this case. However, our point here is not about the quality of the identification process, but rather about showing that identification yields a "system program" which can be composed with the "control program" for model checking purposes.

Once the physical system plus its control software are represented as C code, we can consider software model checking techniques to assess various properties of the system as a whole. In particular, we focus on the problem of guaranteeing that the PWM signal sent to the actuators will be bounded within a given range, even assuming that the sensor readings are affected by noise. In particular, we assume that noise

Table 1. ESBMC CPU time (in seconds) to verify different configurations.

PWM bound	n=2	n=4	n=6	n=8
-10;10	0.030	13.126	FAILED	FAILED
-12;12	0.030	16.401	370.632	4721.857
-14;14	0.028	12.855	360.600	4562.389
-16;16	0.028	10.783	311.470	5338.552
-18;18	0.030	11.239	325.786	5892.505
-20;20	0.032	11.324	269.531	3962.351
-22;22	0.031	11.445	263.744	5729.603
-24;24	0.031	11.115	266.816	2812.533
-26;26	0.028	10.765	268.889	5791.778
-28;28	0.030	11.250	263.010	3340.451

can be any value chosen non-deterministically in a given range — [-2.5,2.5] for the gyroscope and [-4,4] for the encoder, In practice, this choice encompasses any stochastic noise process whose probability of generating values outside the stated bounds is negligible, which is indeed what we observed in our experiments with the robot. Our tool of choice for verification is ESBMC [17]. ESBMC is a context-bounded model checker for embedded C/C++ software based on Satisfiability Modulo Theories (SMT) solvers. As such, ESBMC is able to deal seamlessly with the source code of the embedded controller, and it is also able to deal with the mixed linear integer-real arithmetic operations that make up the bulk of the code. ESBMC checks automatically for a variety of bugs, including arithmetic under- and overflow, pointer safety, memory leaks, and array bounds. However, we can also customize the verification process by adding *assumptions*, i.e., preconditions stating that a given variable will always range within stated bounds, and *assertions*, i.e., conditions that should never become false (similar to the `assert` macro found in many C language implementations). We run ESBMC[4] varying two parameters: the PWM bound asserted — from [-10, 10] to [-28,28] — and the number of iterations performed in the main loop — from 2 to 8. Notice that, since the main loop is infinite, the program could not be verified because of termination issues. However, since we are trying to find bugs in the code, we are satisfied with traces of finite length obtained with limited loop unwiding. The issue becomes that, as shown in Table 1, even with a relatively small unwinding — 8 iterations in the main loop correspond to 80ms of system working time — the computational effort to run ESBMC to completion is substantial. For every bound value, growing from n to $n + 2$ in the main loop unwiding will cause ESBMC CPU time to grow at least one order of magnitude. Given this state of affairs, it is clear that running ESBMC for more than one hour in order to verify a "system life" of about 80ms is untenable. While ESBMC is able to find assertion violations for $n = 6$ and $n = 8$ given a PWM bound of [-10,10], we know from experiments that this can be due to physiological transient effects when the system is started. From these experiments we conclude that verification of embedded software requires more research effort to make model checking techniques viable in this context.

[4] Experiments with ESBMC are performed on an Intel Core i7-3770 quad core at 3.40 GHz with 32 GB RAM, equipped with Ubuntu 12.04 LTS 64 bit

3 Middleware Identification

Middleware is an important part of robotic control architectures — see [18] for a recent survey. Since robots are complex distributed systems, middleware tends to reflect this complexity, making it difficult to check the code that relies on top of it. On one hand, a precise model of the whole middleware may not be available. On the other, even if we had one, it could be simply too complex to cope with. Our motivation for investigating black-box software identification is the opportunity given by these techniques to obtain compact models of middleware from controlled experimentation. Our choice of YARP [9] as a case study is dictated by several reasons, including a deep knowledge of the platform, and a fairly large installed base due to the adoption of YARP as the standard middleware of the humanoid iCub [19]. Moreover, YARP is a publish-subscribe architecture quite similar to other middleware widely used in the robotics community such as ROS [20]. From the implementation point of view, YARP is a set of libraries written in C++ consisting of more than 150K lines of code. The purpose of YARP is to support modularity by abstracting algorithms and the interface to the hardware and operating systems. YARP abstractions are defined in terms of protocols. One of the main functionalities of YARP is to support inter-process communication using a "port" abstraction. The `port` C++ class follows an "Observer" design pattern by decoupling producers and consumers. Objects of the `port` class can deliver messages of any size and type across a network, using a number of underlying protocols — including shared memory, when available. In doing so, ports attempt to decouple as much as possible the behavior of the two sides of the communication channels, as a function of some user-defined parameters. Finally, port objects can be commanded at run time to connect and disconnect among each other.

Automated construction of behavioral models for black-box components can be obtained with automata-based identification techniques, also known as *automata learning* or *grammatical inference*. In particular, we consider *active* automata learning, wherin we assume that the system to be identified is available for controlled experimentation — as opposed to *passive* learning where only on-line traces of the system behavior are available. One of the earliest algorithmic contributions in active automata learning is a paper by Angluin [21] wherein the concept of *minimally adequate teacher* (MAT) is introduced, as well as the L^* algorithm to identify deterministic finite-state automata (DFA). The main idea is that the system to be learned can be modeled by a DFA and it comes along with a *teacher* (also *oracle*) which can answer two kinds of queries: *membership queries*, in the form of "is an input word accepted by the system?", and *equivalence queries*, in the form "is a given conjecture automaton equivalent to the actual system?". The answer to the first question is just "yes" or "no", whereas in the second case, the answer can be "yes", or a counterexample in the symmetric difference of the languages recognized by the system on one hand, and the conjecture automaton on the other. In practice, the teacher will be replaced by testing procedures which implement membership queries and approximate equivalence queries using, e.g., conformance testing – see Gargantinis review [22]. An adaptation of Angluin's L^* algorithm for identifying (deterministic) transducers was first developed by Niese [23] and it was further extended by Shahbaz [7]. In [24] identification of transducers is used as a basis to identify *interface automata* (IA). This model is relevant in our context

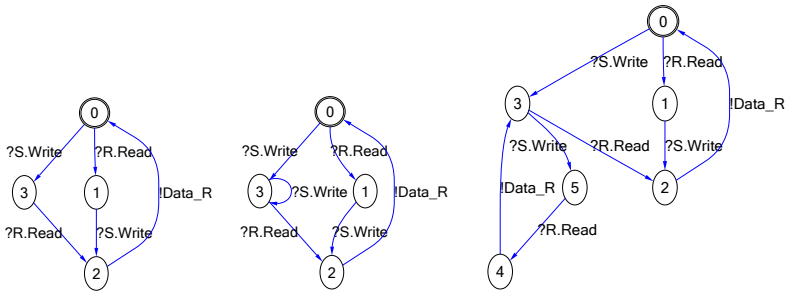

Fig. 4. Behavior of a (standard) port with one sender S and one receiver R (left), behavior of a buffered port non-strict (center) and strict (right) read. The input actions *R.Read* and *R.Write* denote the receiver thread reading from a port, and the sender thread writing to a port, respectively. The *Data_R* refers to the output action of delivering data to the receiver. By convention, we append to the name of a transition the symbol ? (resp. !) to denote that it is an input (resp. output) action, and the initial state is represented with a double circle.

because IA can model reactive systems that interact with their surrounding environment in an asynchronous way. Formally, an *interface automaton* (IA) as a quintuple $P = (I, O, Q, q_0, \rightarrow)$ where I is a set of *input actions*, O is a set of *output actions*, Q is a set of states, $q_0 \in Q$ is the *initial state* of the system, $\rightarrow \subset Q \times (I \cup O) \times Q$ is the *transition relation*, and the sets O, I and Q are finite, nonempty and mutually disjoint.

We have developed a tool called AIDE (Automata IDentification Engine) whose purpose is to enable black-box identification according to different models, including DFA, transducers and IA, and different algorithms, e.g., Angluin's L^* and Shahabaz L_M^+. AIDE provides also learning of (behavior non-deterministic) IAs. From an architectural point of view this requires several components. The first one is a wrapper around the system — YARP API functions in our case — which translates actual input/outputs to the symbolic representations used internally by AIDE. The second component implements a translation bewteen IAs and non-deterministic transducers. The algorithm implemented in AIDE is an extension of the translation presented in [24]. The core component of the identification process is our N^* algorithm — described in [25] — which extends Shahabaz's L_M^+ to deal with non-deterministic transducers. Finally, since we assume no a-priori knowledge about YARP components that we are trying to identify, a conformace testing engine provides approximate equivalence testing for conjecture automata — more details can be found in [25, 8].

Our goal is to identify the behavior of YARP ports in a specific setting using AIDE and IA as a formal model of YARP behavior. We assume a typical scenario inside a robotic control architecture where some component, say an *Object Detector*, is processing a stream of data, e.g., streamed images from the robot camera, to produce some result, e.g., coordinates of the detected object in the 3D space. The output of the first component is connected to a second one, say *Gaze Control*, which receives a the coordinates and, e.g., controls the head of the robot to gaze at the target point. To model this scenario, we consider the behavior of one port with just one sender and one receiver. In particular we used AIDE to identify the port model in three different configurations, and the resulting models are depicted in Figure 4. In the case of a (standard) YARP

port, the model identified by AIDE in a scenario with one sender S and one receiver R is shown in Figure 4 (left). The identification of this model required 50 CPU seconds[5]. In this case, the type of communication is of a "send/reply" type, wherein the sender and the receiver are tightly coupled. In the case of *buffered* ports, the sender and the receiver enjoy more decoupling, in the sense that YARP takes care of the lifetime of the objects being transmitted through the ports and it makes a pool of them, growing upon need. By default, a buffered port will just keep the newest message received[6]. Therefore, messages that come in between two successive calls to read, might be dropped. If the so called *strict* mode is enabled, YARP will keep all received messages — like a FIFO buffer. Notice that, in this mode, the state space of the system would be infinite. Therefore, to learn this model with AIDE, we limit the system to send no more than N packets, i.e., we assume that the buffer will not exceed the maximum size of N messages. In Figure 4 (center), we can see the inferred behavior of a buffered port in normal mode, and in Figure 4 (right), we can see a buffered port in strict mode with $N = 2$. In the latter case, 150 CPU seconds were required for the the IA learner to infer the model.

```
1: Initialize buffered ports P_1 and P_2
2: Connect P_1 to Q_1
3: while true do
4:     for i = 1 to N do
5:         Write message m to P_1
6:     end for
7:     Read message from P_2
8: end while
```

```
1: Initialize buffered ports Q_1 and Q_2
2: Set the reading mode of Q_1 as strict
3: Connect Q_2 to P_2
4: while true do
5:     for i = 1 to N do
6:         Read message m from Q_1
7:     end for
8:     Create a message and write it to Q_2
9: end while
```

Fig. 5. An example code client (left) and server (right). In the server it is easy to introduce a subtle bug: if the port is not configured to use strict read mode (i.e., line 2) the server may loose a packet and wait forever.

To see how identification of YARP components can help verification of code that relies on top of it, let us assume that we have two components *Client* and *Server* whose pseudo-code is presented in Figure 5. The idea is that the client sends some constant number of packets, say N, and the server waits for them, then processes them, and finally it returns a result to the client. After the result packet is received, the procedure starts again. However, a problem arises if the client sends packets too fastly through a YARP port configured for non-strict mode. In this case, there is a concrete chance that the server misses some of the packets, and a deadlock may occur. Both client and server programs can be translated into an interface automaton and together, they can be composed with one of the identified automata of the port component shown in Figure 4, considering a specific number of packets N. For $N = 2$, it is easy to see that the trace $?P_1.Send \rightarrow ?P_1.Send \rightarrow ?Q_1.Receive \rightarrow !Data\_Q_1$ may cause a deadlock in the composed model if the behavior of the YARP port is defined in non-strict mode as in Figure 4 (center), whereas the communication will always be fine if the port is operating in strict mode as in Figure 4 (right).

[5] Notice that most of the overall time is spent in network communication between the learner and system wrapper, in resetting the system, and other overhead tasks; less that 1% of time was actually needed for the "learning algorithm". All the identification experiments have been carried out on a Dell laptop with Core2Duo 2.53GHz CPU and 4GB of RAM on Ubuntu 12.04.

[6] This behavior is the default because in the implementation of control loops it is preferable to reduce the communication latency at the cost of dropping (late) packets.

4 Safe Reinforcement Learning

It is well-established in the robotics scientific literature that RL methods represent an effective solution for efficient learning on several visual-motorial and control tasks. These tasks can be found in complex, fast, and demanding environments representing some real-world applications, and they are often used as test bed platforms in robotics. The air hockey game surely represent a complex, fast, and demanding environment, as witnessed by its large usage as test bed in robotics – see, e.g., [26, 27]. Air hockey is played by two players. They use round paddles (mallets) to hit a flat round puck across a low-friction table. At each end of the table there is a goal area. The objective of the game is to hit the puck so that it goes into the opponent's goal (offense play), and to prevent the opponent to score into your own goal (defense play). In this environment, focusing on a robot playing against a human opponent, the *safety* of the control policies learned by using RL algorithms is a property of paramount importance in order to avoid dangerous behaviour – e.g., moving too fast or too close to the table's edges – and formal verification can be a key-enabler to check such property.

Formal verification of RL algorithms starts from the framework of Markov Decision Processes (MDPs), considering a set S of distinct *states*, and a set A of *actions* that can be performed. At each discrete *time step* t, the agent senses state s_t, and performs action a_t. The environment responds by giving the agent a *reward* $r_{t+1} = r(s_t, a_t)$ and by producing a *transition* to the succeeding state $s_{t+1} = \delta(s_t, a_t)$. While the functions r and δ may not be known, the founding assumption is that they depend only on the current state and action. A *policy* is any mapping from s_t to the action a_t. We consider *stochastic policies*, i.e., functions of the form $\pi : S \times A \to [0, 1]$[7] which return the probability of executing a given action in a state. The combination of the original MDP and a policy π computed by some RL algorithm yields a *discrete-time Markov chain* (DTMC) describing all possible interactions of the agent with the environment. Following [13], given a set of atomic propositions AP, a DTMC is defined as a tuple $(W, \overline{w}, \mathbf{P}, L)$ where W is a finite set of *states*, and $\overline{w} \in W$ is the *initial state*; $\mathbf{P} : W \times W \to [0, 1]$ is the *transition probability matrix*; and $L : W \to 2^{AP}$ is the *labeling function*. An element $\mathbf{P}(w, w')$ gives the probability of making a transition from state w to state w' with the requirement that $\sum_{w' \in W} \mathbf{P}(w, w') = 1$ for all states $w \in W$. A terminating (absorbing) state w is modeled by letting $\mathbf{P}(w, w) = 1$. The labeling function maps every state to a set of propositions from AP which are true in that state.

Safety assurance can be expressed in terms of Probabilistic Computational Tree Logic (PCTL) formulas, whose syntax is defined inductively as:

- if $\varphi \in AP$ then $\varphi \in \Sigma$;
- $\top \in \Sigma$; if $\alpha, \beta \in \Sigma$ then also $\alpha \wedge \beta \in \Sigma$ and $\neg\alpha \in \Sigma$
- $\mathcal{P}_{\bowtie p}[\psi] \in \Sigma$ where $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0, 1]$ and ψ is either $X\alpha$ (*next*), $\alpha \mathcal{U}^{\leq k} \beta$ (*bounded until*) and $\alpha \mathcal{U} \beta$ (*until*) with $\alpha, \beta \in \Sigma$ and $k \in \mathbb{N}$

The semantics of PCTL formulas is given over *paths*, where a path τ is defined as a non-empty sequence of states $w_0, w_1, w_2, \ldots$ where $w_i \in W$ and $\mathbf{P}(w_i, w_{i+1}) > 0$ for all $i \geq 0$. We denote the ith state of τ by $\tau(i)$. Paths can be either finite of infinite.

[7] $[0, 1]$ denotes a closed sub-interval of $\mathbb{R}$, i.e., every $x \in \mathbb{R}$ such that $0 \leq x \leq 1$.

In case of finite paths, the *length* of the path is just the number of states in the path, and we denote with $Path_w$ the set of (infinite) paths starting from w. Given a DTMC $\mathcal{M} = (W, \overline{w}, \mathbf{P}, L)$ and a state $w \in W$, let $\mathcal{M}, w \models \varphi$ denote that $\varphi \in \Sigma$ is satisfied in w. Intuitively, expression made up of atomic propositions, $\top$, and the connectives $\neg$ and $\wedge$ have the standard Boolean semantics; to assess whether $M, w \models \mathcal{P}_{\bowtie p}[\psi]$, we have to consider whether the condition specified by ψ is verified on a path starting from w in the model M. For the sake of our research, we will be concerned only with formulas of the form

$$\mathcal{M}, w \models \mathcal{P}_{<\sigma}[\mathcal{F}\ bad] \tag{3}$$

where $\mathcal{F}\beta$ (*eventually*) is an abbreviation of $\top \mathcal{U} \beta$, and $bad \in AP$ labels any state which is considered to be undesirable. Intuitively, this formula is satisfied if the total probability of reaching a state bad from state w along any path from w to bad is less than a threshold σ — see [13] for further details about PCTL and its semantics.

Let us assume that states and transitions explored during learning are logged in a table $T : S \times A \to 2^{S \times \mathbb{N}}$, where S is the state-space and A is the action-space. For each state $s \in S$ and action $a \in A$, we have that $T(s, a) = \{(s_1; n_1), \ldots, (s_k; n_k)\}$ where $s_i \in S$ for all $1 \leq i \leq k$, and n_i is the number of times state s_i occurred as next state in a transition from state s on action a; each cell (s, a) of T stores only the pairs $(s_{;}n_i)$ such that $n_i > 0$, i.e., state s_i appeared at least once in a transition from an explored state s on action a. For all actions $a \in A$, if the state $s \in S$ was explored, but had no outgoing transitions, then $T(s, a) = \emptyset$; if a state was not explored, then $T(s, a) = \perp$.[8] Depending on whether (un)safety is defined for states or transitions, we also must keep track of unsafe states/transitions that were visited/executed during learning. If unsafety is in terms of states, i.e., the state itself is undesirable, then we need a table $F : S \to \{\text{TRUE}, \text{FALSE}\}$, where $F(s) = \text{TRUE}$ exactly when state s is found to be unsafe during learning. On the other hand, if transitions can be unsafe, i.e., the action itself is undesirable, then we consider a table $F : S \times A \to \{\text{TRUE}, \text{FALSE}\}$ where, for each explored state $s \in S$ and action $a \in A$, we log whether the specific action a is found to be unsafe when started from state s. If we assume that the learning algorithm returns a *state-action quality table* Q, i.e., a function $Q : S \times A \to \mathbb{R}$ that expresses the value of executing a given action in a given state, then tables T and F can be combined with Q to obtain a DTMC $\mathcal{M} = (W, w_0, \mathbf{P}, L)$. The set AP of atomic propositions has just one element bad which denotes unwanted states. The construction of $\mathcal{M}$ is detailed in [28], but here we just mention that we take into account the problem of unsafe transitions, i.e., all the cases in which the policy learned by RL commands the manipulator to perform a move that would imply excessive speed. This kind of behavior, if repeated over and over again, can bring damage to the manipulator and thus is to be avoided, even if the RL algorithm may try to suggest it in the attempt to pursue greedily the optimality of the policy.

We define *policy repair* as a verification-based procedure that modifies the Q-table to fix the corresponding policy. Repair is driven by counterexamples that witness the violation of a safety property like (3) for some threshold σ. Since there are no infinite

[8] Table T can handle (i) deterministic actions, i.e., a is such that $\forall s \in S. |T(s, a)| = 1$; (ii) non-deterministic actions, i.e.,a is such that for all $s \in S$, both $|T(s, a)| = k$ and $n_i = 1/k$ for all $1 \leq i \leq k$; and (iii) probabilistic actions, i.e., arbitrary values of n_i.

paths in $\mathcal{M}$, property (3) is violated only if σ is less than the probability of choosing a path $\tau = w_0, w_1, \ldots, w_k$ such that $w_k = w_{bad}$ and $L(w_{bad}) = \{bad\}$. The *probability mass* of a path τ is defined as

$$Prob(\tau) = \Pi_{j=1}^{k} \mathbf{P}(w_{j-1}, w_j) \tag{4}$$

and extended to a set of paths $\mathcal{C} = \{\tau_1, \ldots, \tau_n\} \subseteq Path_{w_0}$ as $Prob(\mathcal{C}) = \sum_{i=1}^{n} Prob(\tau_i)$. If, for all $1 \leq i \leq n$, we have $\tau_i = w_{i,0}, w_{i,1}, \ldots, w_{i,k_i}$ with $w_{i,0} = w_0$, $w_{i,k_i} = w_{bad}$ and $Prob(\mathcal{C}) > \sigma$, then $\mathcal{C}$ is a counterexample of property (3). Notice that a counterexample $\mathcal{C}$ is unique only if it contains all paths starting with w_0 and ending with w_{bad}, and for all $\mathcal{C}' \subset \mathcal{C}$ we have that $Prob(\mathcal{C}') < \sigma$. Given our assumptions, the counterexample $\mathcal{C} = \{\tau_1, \ldots, \tau_n\}$ is a DAG such that $|\mathcal{C}| \leq |Path_{w_0}|$. Unless learning is highly ineffective, we expect that $|\mathcal{C}| \ll |Path_{w_0}|$, i.e., focusing on the counterexample is an effective way to limit the repair procedure to a handful of "critical" paths. However, focusing on the counterexample is just the first step in verification-based repair, and we also need to find an effective way to alter the policy, i.e., the Q-values, so that probability masses are shifted in the DTMC and property (3) is satisfied. Our implementation leverages two key ideas to accomplish this. The first one is that transitions between states which are "far" from w_{bad} should be discouraged less that those that are "near" w_{bad}. The second idea is that probabilities should "flow away" from paths ending with w_{bad} towards paths ending with w_{ok}. Altering transitions which are close to w_{bad} helps to keep changes to the policy as local as possible, and to avoid wasting useful learning results. Shifting probabilities towards paths ending with w_{ok} avoids moving probability mass from one unsafe path to another, yielding no global improvement — see [28] for further details.

In our current simulated air hockey setup, we model the air hockey environment and the related RL problem as follows. At the beginning of each episode, the puck is fired from a random point in vicinity of the opponent's goal area, with speed and inclination chosen uniformly at random within some practically feasible range. An episode ends in one of the following ways: (i) the puck hits the border on the opponent's side of the table, (ii) a time-out occurs, or (iii) a goal is scored. With reference to a Cartesian system centered in the robot's goal area, the position of the puck is represented by the couple (p, α), where p is the perpendicular dropped from the origin on the trajectory line of the puck, and α is the angle made by this perpendicular with respect to the abscissa. A state is a vector (p, α, θ), where θ is the current robot's mallet angle with respect to a polar system also centered in the robot's goal area.[9] The parameters p and α are quantized over 58 and 37 values, respectively, and θ is quantized over 31 values, yielding 66526 possible states.[10]

In Figure 6, we present the results about learning, where we consider the percentage of saved goals (plot on the left), and the percentage of unsafe transitions (plot on the right). Both plots report from two phases: learning across a number of episodes

[9] The radius ρ is not needed because we keep it fixed throughout learning and simulation. In a pure defense play, this choice does not hamper the robot's ability to defend the goal area.

[10] Simulation and learning are performed on an Intel Core i5-480M quad core at 2.67 GHz with 4GB RAM, equipped with Ubuntu 12.04 LTS 64 bit.

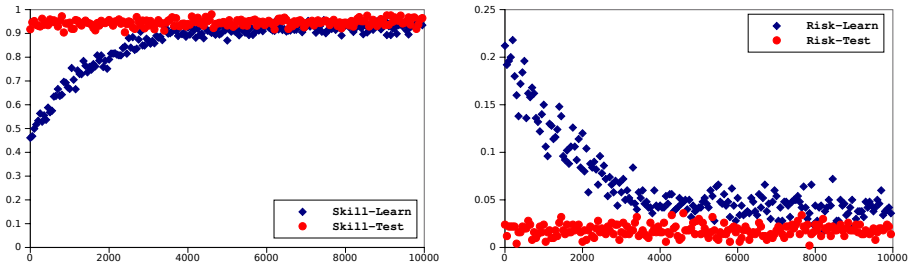

Fig. 6. Performance (left) and safety (right) while learning and during testing. The x-axis in both plots indicates the total number of episodes; the plot on the left reports the average percentage of saved goals, i.e., the y-axis indicates the ratio between total shots and goals scored. The plot on the right reports the average number of unsafe transitions on the y-axis. In both plots, each point is the average value over a batch of 50 episodes, diamonds denote values obtained while learning, and circles denote values obtained while testing.

(diamond dots), and testing the policy learned across several trials (circle dots). Notice that in both plots, sequence matters only for learning trials, because earlier trials can count on less accurate Q-values. In the case of testing, we just show the variance and the quality of the final policy across several trials, but order is not important. As we can see from the plots, learning converges to a reasonable performance after about 5000 episodes. The percentage of saved goals vs. potential goals is always greater than 0.8 after 2000 episodes, and this lower bound raises to 0.95 after 5000 episodes. As for safety, after 5000 episodes, the percentage of unsafe transitions is always less than 0.02, and this upper bound does not decrease in a substantial way even after 10000 learning episodes.

The DTMC encoding in the air hockey case study has 4313 states and 8850 transitions. All the model checkers that we consider[11] are able to compute the safety threshold σ for us, agreeing on a value $\sigma = 0.0243$. In terms of CPU time, their performance is quite different, in that COMICS and MRMC complete in less than 0.01s, whereas PRISM requires 3.18s to check the property — with additional 164.14s to build the model. In this regard, it is important to notice that COMICS and MRMC require an explicitly built DTMC as input, whereas PRISM allows for an implicit declaration which is more user friendly and compact, but requires extra time to be compiled into a DTMC.

In Table 2, we report the empirical results related to the automated repair procedure based on the results of COMICS only.[12] We checked three different safety thresholds, namely $\sigma = \{0.01, 0.001, 0.0001\}$ against three different values of decay coefficient δ, namely 1, 5, and 10. Intuitively, δ represents how strong it is the impact of repair along the path: the higher is δ, the more edges that are far away from the bad state will be modified, according to an exponential decay law. In this experiment, we re-check

[11] Verification and repair are performed on a Intel Core i3-2330M quad core at 2.20 GHz with similar RAM and OS. Verification of policies is carried out with state-of-the-art probabilistic model checkers, namely COMICS [29] (version 1.0), MRMC [30] (version 1.4.1), and PRISM [31] (version 4.0.3). All tools are run in their default configuration with the exception of COMICS, for which the option `--concrete` is selected instead of the default `--abstract`.

[12] MRMC does not implement counterexample generation, and in PRISM this is still a beta-stage feature.

σ \ δ	1			5			10		
	τ	Eff.	Time	τ	Eff.	Time	τ	Eff.	Time
0.01	17	0.967	12.93	7	0.968	5.26	7	0.968	5.57
0.001	48	0.967	37.22	25	0.970	18.95	25	0.961	19.09
0.0001	75	0.968	59.84	39	0.966	29.89	39	0.964	29.35

Table 2. Automated repair. Columns and rows are labeled with values of δ — the decay coefficient used when repairing along a path — and σ — the target safety threshold. Each group is composed of three columns. In the first one ("τ"), we report the total amount of counterexamples involved in the repair. In the second column ("Eff."), we report the effectiveness of the learning agent, i.e., the mean value over 10 random tests of the ratio of saved goals versus potential goals after repairing. Finally, in the last column ("Time") we report the CPU time (in seconds) spent by the repair procedure.

safety after taking into account every path of $\mathcal{C}$. As mentioned before, our starting point is $\sigma = 0.0243$, so every value of σ in Table 2 requires fixing the agent. The initial performance score of the agent is 0.967, i.e., nearly 97% of the potential goals are saved. Looking at Table 2, we can see that robot's performance does not change in a noticeable way, and thus the repair procedure is not invasive in these configurations. Furthermore, repair guarantees a safety level which is two orders of magnitude smaller than the starting one and, in this sense, it is also highly effective.

5 Conclusions

Summing up, in this paper we have presented three computer-augmented approaches aiming to ensure dependability of a control architecture. All three approaches are formally grounded, and thus they can provide precise and demonstrable statements about desired properties. Nevertheless, their usage is intended to be highly automated with a minimum impact on the developer. The results that we obtained are mixed. In the case of verification of control software (Section 2) our findings are that current software model checking technology is unable to cope with relatively simple systems and their control code. This partially contradicts what was shown in [6], wherein some interesting properties related to control code in autonomous robots could be demonstrated. The main reason seems to be in our case that the program to be checked (IIR filter + control code) is "data intensive", i.e., the control structure of the program is fairly simple, but the calculations to be performed are more involved than those appearing in [6], and the property we wish to prove critically depends on such calculations. Also, ESBMC is oblivious of any structural property of the code which could help to abstract the code to more concise, yet informative models — see, e.g., [32] for results with control code involving artificial neural networks. On the other hand, if we consider black-box identification (Section 3) and safe learning (Section 4) we can conclude that the results are more satisfying. In particular, for middleware identification we have shown that our tool AIDE can automatically construct abstract formal models of ports, a widely used YARP component. Since YARP is the middleware of choice in the humanoid iCub [19], AIDE can enable the adoption of precise techniques for testing and verification of relevant components in iCub's control architecture. In addition our approach can be readily adopted to model the behavior of any publish-subscribe architecture, e.g., the popular ROS middleware. Clearly, scaling to more complex components must be considered in

our future research agenda, but AIDE already enables developers to check their code against common errors such as, e.g., incorrect port flagging. As for safe learning, we have shown that we can automatically extract DTMCs considering (i) policies learned by agents using RL and (ii) recordings of the state-action space exploration performed by agents while learning. In this way, probabilistic model checking tools like COMICS, can calculate the probability of the policy to drive the robot to unwanted states. The size of the problem that can be handled by COMICS is realistic (thousands of states and transitions), and the computing time is in the order of milliseconds. Furthermore, when the policy is unsafe, i.e., bad states are reachable with probability higher than a given threshold, COMICS generates counterexamples that allow us to repair the original policy. This procedure is also cost-effective: less than 30 CPU seconds are enough to reach safety thresholds in the order of 10^{-4}. Our main research focus in this case, will be on adapting this methodology to more advanced RL methods, such as [33] and similar recent contributions requiring continuos state and action spaces.

References

1. M. Bajracharya, M. Maimone, and D. Helmick. Autonomy for mars rovers: Past, present, and future. *Computer*, 41(12):44–50, 2008.
2. M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mosenlechner, D. Pangercic, T. Ruhr, and M. Tenorth. Robotic roommates making pancakes. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 529–536. IEEE, 2011.
3. G. Pratt and J. Manzo. The DARPA Robotics Challenge [Competitions]. *Robotics & Automation Magazine, IEEE*, 20(2):10–12, 2013.
4. C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G.J. Pappas. Symbolic planning and control of robot motion [grand challenges of robotics]. *Robotics & Automation Magazine, IEEE*, 14(1):61–70, 2007.
5. R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.
6. S. Scherer, F. Lerda, and E. M. Clarke. Model checking of robotic control systems. In *Proceedings of ISAIRAS 2005 Conference*, pages 5–8, 2005.
7. M. Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing*. PhD thesis, Institut Polytechnique de Grenoble, Grenoble, France, 2008.
8. A. Khalili and A. Tacchella. AIDE: Automata-Identification Engine. `http://aide.codeplex.com`.
9. P. Fitzpatrick, G. Metta, and L. Natale. Towards long-lived robot genes. *Robotics and Autonomous systems*, 56(1):29–45, 2008.
10. R.S. Sutton and A.G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.
11. J.A. Bagnell and S. Schaal. Special issue on Machine Learning in Robotics (Editorial). *The International Journal of Robotics Research*, 27(2):155–156, 2008.
12. J.H. Gillula and C.J. Tomlin. Guaranteed Safe Online Learning via Reachability: tracking a ground target using a quadrotor. In *ICRA*, pages 2723–2730, 2012.
13. M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. *Formal methods for performance evaluation*, pages 220–270, 2007.
14. Rudolf Emil Kalman et al. Contributions to the theory of optimal control. *Bol. Soc. Mat. Mexicana*, 5(2):102–119, 1960.
15. P. Lancaster and L. Rodman. *Algebraic riccati equations*. Oxford University Press, 1995.

16. MATLAB. *version 8.1.0 (R2013a)*. The MathWorks Inc., Natick, Massachusetts, 2013.
17. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *Int.l Conf. on Automated Software Engineering*, pages 137–148, 2009.
18. N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Middleware for robotics: A survey. In *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*, pages 736–742. IEEE, 2008.
19. G. Metta, L. Natale, F. Nori, G. Sandini, D. Vernon, L. Fadiga, C. von Hofsten, K. Rosander, M. Lopes, J. Santos-Victor, et al. The iCub Humanoid Robot: An Open-Systems Platform for Research in Cognitive Development. *Neural networks: the official journal of the International Neural Network Society*, 2010.
20. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, 2009.
21. D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.
22. A. Gargantini. Conformance testing. *Model-Based Testing of Reactive Systems*, pages 87–111, 2005.
23. O. Niese. *An integrated approach to testing complex systems*. PhD thesis, Universität Dortmund, Dortmund, Germany, December 2003.
24. F. Aarts and F. Vaandrager. Learning I/O automata. *CONCUR 2010-Concurrency Theory*, pages 71–85, 2010.
25. A. Khalili and A. Tacchella. Learning nondeterministic Mealy machines. Technical report, University of Genoa, 2013.
26. D. C. Bentivegna, C. G. Atkeson A. Ude, and G. Cheng. Learning to Act from Observation and Practice. *International Journal of Humanoid Robotics*, 1(4), December 2004.
27. G. Metta, L. Natale, S. Pathak, L. Pulina, and A. Tacchella. Safe and effective learning: A case study. In *ICRA*, pages 4809–4814, 2010.
28. S. Pathak, L. Pulina, G. Metta, and A. Tacchella. Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the iCub. In *IROS*, pages 170–175, 2013.
29. E. Abrahám, N. Jansen, R. Wimmer, J. Katoen, and B. Becker. DTMC model checking by SCC reduction. In *Quantitative Evaluation of Systems (QEST), 2010 Seventh International Conference on the*, pages 37–46. IEEE, 2010.
30. J.P. Katoen, I.S. Zapreev, E.M. Hahn, H. Hermanns, and D.N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance evaluation*, 68(2):90–104, 2011.
31. M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 113–140, 2002.
32. L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *22nd International Conference on Computer Aided Verification (CAV 2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2010.
33. X.C. Ding, S.L. Smith, C. Belta, and D. Rus. MDP optimal control under temporal logic constraints. In *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*, pages 532–538. IEEE, 2011.