LETTER — **Communicated by Florentin Worgötter**

Formal Modeling of Robot Behavior with Learning

Ryan Kirwan
ryankirwan85@gmail.com
Alice Miller
alice.miller@glasgow.ac.uk
School of Computing Science, University of Glasgow, Glasgow G12 8RZ, Scotland

Bernd Porr
bernd.porr@glasgow.ac.uk
P. Di Prodi
robomotic@gmail.com
School of Engineering, University of Glasgow, Glasgow G12 8RZ, Scotland

We present formal specification and verification of a robot moving in a complex network, using temporal sequence learning to avoid obstacles. Our aim is to demonstrate the benefit of using a formal approach to analyze such a system as a complementary approach to simulation. We first describe a classical closed-loop simulation of the system and compare this approach to one in which the system is analyzed using formal verification. We show that the formal verification has some advantages over classical simulation and finds deficiencies our classical simulation did not identify. Specifically we present a formal specification of the system, defined in the Promela modeling language and show how the associated model is verified using the Spin model checker. We then introduce an abstract model that is suitable for verifying the same properties for any environment with obstacles under a given set of assumptions. We outline how we can prove that our abstraction is sound: any property that holds for the abstracted model will hold in the original (unabstracted) model.

1 Introduction

Simulation is commonly used to investigate closed-loop systems. Here, we focus on biologically inspired closed-loop systems where an agent interacts with its environment (Walter, 1953; Braitenberg, 1984; Verschure & Pfeifer, 1992). The loop here is established as a result of the agent responding to signals from its sensors by generating motor actions that change the agent's sensor inputs. More recently, simulation has been used to analyze closed-loop systems in which the response of an agent changes with time due

Neural Computation **25, 2976–3019** (2013) © 2013 Massachusetts Institute of Technology
doi:10.1162/NECO_a_00493

to learning from the environment (Verschure & Voegtlin, 1998; Kulvicius, Kolodziejski, Tamosiunaite, Porr, & Wörgötter, 2010). This adds a new layer of complexity where the dynamics of the closed loop will change and an initially stable system might become unstable over time.

Simulation is a relatively inexpensive method for determining the behavior of a system. Even single experiments are informative and by applying statistical methods to a series of experiments, researchers can draw inferences concerning overall trends in behavior. However, simulation alone is not sufficient to determine properties of the form: in all cases property P holds, or it is never true that property Q holds.

Formal methods have two major benefits when applied to this type of system. First, formal specification of the system requires precision in system description (e.g., in the rules determining an agent's response to a particular signal), thus avoiding redundancy and inconsistency. Second, verification of a system can allow us to prove properties that hold for any run of the system (i.e., that should hold for any experiment).

Formal methods have been used to analyze agent-based systems (Hilaire, Koukam, Gruer, & Müller, 2000; Hilaire, Simonin, Koukam, & Ferber, 2004; Da Silva & De Lucena, 2004; Wooldridge, Jennings, & Kinny, 2004; D'Inverno, Luck, Georgeff, Kinny, & Wooldridge, 2004; Fisher, 2005). These approaches involve new formal techniques for theoretical agent-based systems. In contrast, in this letter, we apply an appropriate automatic formal technique to a real system that has previously been analyzed using classical closed-loop simulation. We do this in order to compare the two approaches and demonstrate the effectiveness of the application of formal methods in this domain.

Model checking is even more important for agents that learn because most learning rules are inherently unstable (Oja, 1982; Miller, 1996), especially at high learning rates. There is always the risk that weights grow endlessly and eventually render the resulting system dysfunctional. Model checking can guarantee that under a given learning rate, the system will always learn successfully.

In this letter, we describe two formal models of a closed-loop system in which an agent's behavior adapts by temporal sequence learning (Sutton & Barto, 1987; Porr & Wörgötter, 2006). Our first model is obtained from a fairly low-level description of a particular environment. The second, which we refer to as the Abstract model, is obtained from a higher-level representation of a set of environments. The second model is more instructive but requires expert knowledge to construct. We give a brief overview of how we would prove that the Abstract model is sound, in that it preserves properties that hold for the underlying set of environments.

Our models are obtained from specifications defined in the model specification language Promela, and verified using the model checker Spin. We focus on the experiments described in Kulvicius et al. (2010) as an example of temporal sequence learning. We describe how we have reproduced

the experiments using classical closed-loop simulation and compare this classical approach to that using formal verification.

Our models are defined for a simplified environment or set of environments. The purpose of this letter is to provide a proof of concept for our approach. In section 7, we describe how to generate models for more complex scenarios (with a fixed boundary, with additional robots, or more closely packed obstacles, for example).

2 The System

We show how model checking can be used to verify properties of a system that has previously been analyzed using simulation. The system we focus on is that described in Kulvicius et al. (2010) in which simulation is used to investigate how learning is affected by an environment and by the perception of the learning agent. The ability of a robot to successfully navigate its environment is used to assess its learning algorithm and sensor configuration.

We first describe how the robot learns to move toward or away from objects. This is achieved by using the difference between the signals received from the left and right sensors, which can be interpreted as error signals (Braitenberg, 1984). At any time, an error signal x is generated of the form

$$x = sensor_{left} - sensor_{right}, \tag{2.1}$$

where $sensor_{left}$ and $sensor_{right}$ denote the signals from the left and right sensors, respectively. The error is then used to generate the steering angle v, where $v = \omega x$, for some constant ω. The polarity of ω determines whether the behavior is classed as attraction or avoidance (Walter, 1953). The steering then influences the sensor inputs of the robot, and we have formed a closed loop.

Having introduced the general concept of behavior-based robotics we now formalize the agent, the environment, and the closed loop formed by this setup. We have two loops, as shown in Figure 1C, because we have two pairs of sensors passing signals to the robot. One pair of sensors reacts to close-range impacts and is referred to as proximal sensors, with associated signals x_p. The other sensor pair reacts to more distant events and is referred to as distal sensors, with associated signals x_d. If the robot collides with an obstacle, there is an impact on the proximal sensor, which may be preceded by an impact on a distal sensor. This situation is referred to as a delay in the environment. This can happen at any time while the robot interacts with the environment and can be modeled as a stochastic process. It should be noted that this principle is not limited to avoidance but can also be used to learn attraction behavior (Porr & Wörgötter, 2006). However, here we focus on avoidance behavior and how this is learned.

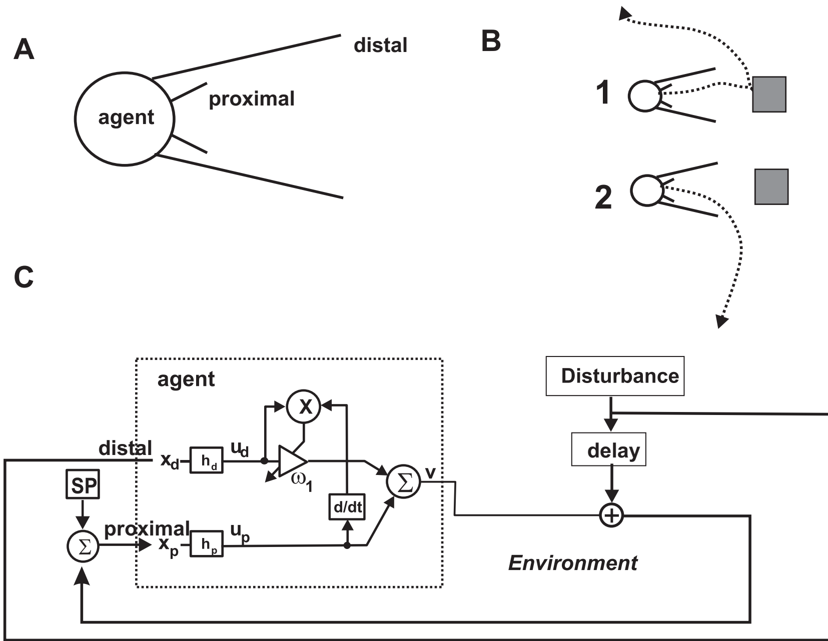

Figure 1: Generic closed-loop data flow with learning. (A) Sensor setup of the robot consisting of proximal and distal sensors. (B1) Reflex behavior. (B2) Proactive behavior. (C) Circuit diagram of the robot and its environment. SP = set point, X is a multiplication operation changing the weight ω_d, Σ is the summation operation, d/dt the derivative, and h_p, h_d low-pass filters.

We introduce learning with the goal of avoiding triggering the proximal sensors by using information from the distal sensors. In order to learn to use the distal sensor information we employ sequence learning; this works on the basis that different sensors react at different times to the presence of an obstacle, causing a sequence of reactions. Figure 1A shows the pairs of proximal and distal sensors at the front of the robot. Note that in our simulation, the proximal and distal sensors are in fact collinear, that is, both the right proximal and distal sensors point in same direction (and so on). For ease of viewing in Figure 1A, this is not the case. Note that although the sensors are collinear, there is no dual contact (i.e., with both sensors) at the places where the sensors overlap. The distal sensor is nonresponsive to contact over the overlapping sections.

The proximal signals cause a reactive behavior that is predefined (or "genetic") and guarantees success (see Figure 1B1). Specifically, when the proximal sensor is hit directly by an obstacle, the robot will behave in such a way as to ensure that it moves away from the obstacle (i.e., escapes).

Figure 1C shows the formalization of the learning system indicated by the box "agent," which contains a summation node $\sum$, which sums the low-pass (h_p, h_d) filtered input signals x_p and x_d where x_p and x_d are determined from the signals from the corresponding sensor pairs (see equation 2.1). The filtered input signals u_p and u_d and the angular response v after an impact to either sensor are determined by equations 2.2, 2.3, and 2.5,

$$u_p = x_p * h_p, \tag{2.2}$$

$$u_d = x_d * h_d, \tag{2.3}$$

where $*$ is the convolution operation and h_p, h_d are low-pass filters defined in discrete time as

$$h(n) = \frac{1}{b} e^{an} \sin(bn) \leftrightarrow H(z) = \frac{1}{(z - e^p)(z - e^{p^*})}. \tag{2.4}$$

The real and imaginary parts of p are defined as $a = \text{real}(p) = -\pi f/Q$ and $b = \text{imag}(p) = \sqrt{(2\pi f)^2 - a^2}$, respectively. $Q = 0.51$ and $f = 0.1$ are identical for both h_p and h_d, which results in a smoothing of the input over at least 10 time steps.

These smoothed distal and proximal signals are then summed to form the steering angle:

$$v = \omega_p u_p + \omega_d u_d. \tag{2.5}$$

The distal weight ω_d is set to zero at the start of the experiment so that only the reactive (predefined) loop via signal x_p and ω_p is active. This loop is set in such a way that the behavior when touching an obstacle is successful (proximal reflex). However, such behavior is suboptimal because the proximal sensor signal x_p first has to be triggered, which might be dangerous or even lethal for the agent. The task of learning is to use the distal sensors (with signal x_d) so that the agent can react earlier. Learning adjusts the weight ω_d in a way that the agent successfully performs this proactive behavior by using the loop via signal x_d. The learning we employ here is ICO learning (input correlation learning; Porr & Wörgötter, 2006), which uses low-pass filters to create correlations between distal and proximal events (see Figure 1C). Low-pass responses (see equations 2.2 and 2.3) smear out the signals and create a temporal overlap between the proximal and distal signals, which can then be correlated by our learning rule to adjust the predictive behavior:

$$\dot{\omega}_d = \lambda \dot{u}_p u_d. \tag{2.6}$$

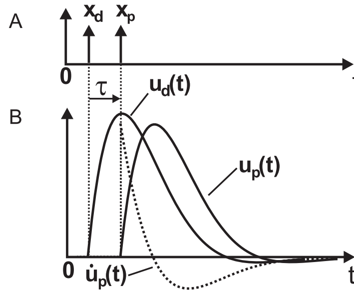

Figure 2: Impact signal correlation with the help of low-pass filters. (A) The input signals from both the distal and proximal sensors, which are τ temporal units apart. (B) The low-pass filtered signals u_p, u_d and the derivative of the proximal signal $\dot{u}_p$.

The derivative of the low-pass filtered proximal signal u_p is used to create a phase lead, which is equivalent to shifting its peak to an earlier moment in time so that a correlation can be performed at the moment the proximal input has been triggered. This is illustrated in Figure 2, which shows the signals from the distal and proximal sensors. Signals are represented as simple pulses in Figure 2A and low-pass filtered signals in Figure 2B. It can be seen that the smearing out of the signals is necessary to achieve a correlation. Learning stops if u_p is constant, which is the case when the proximal sensor is no longer triggered. A sequence of impacts consisting of at least one impact on a distal sensor followed by an impact on a proximal sensor causes an increase in the response (by a factor λ known as the *learning rate*). (See Porr & Wörgötter, 2006, for a more detailed elaboration of differential Hebbian learning.)

While in Kulvicius et al. (2010), the main objective was to measure the performance of the robot, here we concentrate on the performance of the closed loop, which can be benchmarked in different ways using the proximal sensor input x_p and the weight ω_d (the proactive weight). For model checking, satisfaction of the following properties would indicate correct behavior of the robot:

1. The sensor input x_p of the proximal sensor will eventually stay zero, indicating that the agent is using only its distal sensors.
2. The weight ω_d will eventually become constant, indicating that the agent has finished learning.

Note that the two properties are shown by simulation to be true in most cases. However, one simulation leads to a counterintuitive result. This is discussed further in section 3.1. In section 5.2, we discuss how formal verification showed us that this seemingly incorrect result was due to premature termination of the simulation run.

Figures 1B1 and 1B2 show two example behaviors before learning and after learning, respectively. The behavior shows a typical transformation from a purely reflexive behavior to proactive behavior. The agent begins with a zigzag movement by reacting to the collisions (see Figure 1B1), then progresses to smoother trajectories when it learns to respond to its distal antennae (see Figure 1B2). This behavior is generated by the growth of the weight ω_d, which represents the loop via the distal sensors. After successful learning, the proximal antennae will no longer be triggered, which causes the weight ω_d to stabilize.

Our goal is to verify the properties above using model checking. In section 5, we describe how the system is specified in Promela. In section 5.2, we show how these properties can be expressed in linear time temporal logic (LTL) (Pnueli, 1981) and describe the process of verification. We demonstrate that the LTL properties are satisfied for our model of the system.

3 Preliminaries

3.1 Simulation Environment. In order to recreate the simulation results of Kulvicius et al. (2010), we created our own simulation environment, using classical closed-loop simulation tools and ICO learning. In section 8 we compare this approach with that using formal verification. We focus our modeling on how learning is affected by the complexity of the environment. Note that the purpose of this letter is to present a proof of concept—that of using model checking combined with abstraction to verify properties of this type of environment. We have simplified our models, using a set of assumptions in order to clarify our exposition. In section 7, we indicate how our models could be extended to incorporate more realistic, or complex, scenarios.

We restricted the definition of environmental complexity to be a measure of the minimum spacing between obstacles (i.e., a more complex environment implies a smaller minimum space between obstacles). This makes our models simpler. Environment boundaries are removed: when a robot reaches the edge of the environment, it simply emerges again at the opposite point on the environment edge. This simplification is unlike the situation in Kulvicius et al. (2010) but agrees with the setup for our verifications, with which we are comparing results. Removing the boundaries also helps us to abstract our model in section 6. Figure 3 represents the geometry of the simulated environment.

In our behavior-based simulation environment the robot is positioned at coordinates $r_x(t), r_y(t)$ and moves in a grid of pixels. At every time step, the robot moves forward 1 pixel at angle θ (from north):

$$r_x(t) = r_x(t-1) + \cos(\theta), \tag{3.1}$$

$$r_y(t) = r_y(t-1) + \sin(\theta), \tag{3.2}$$

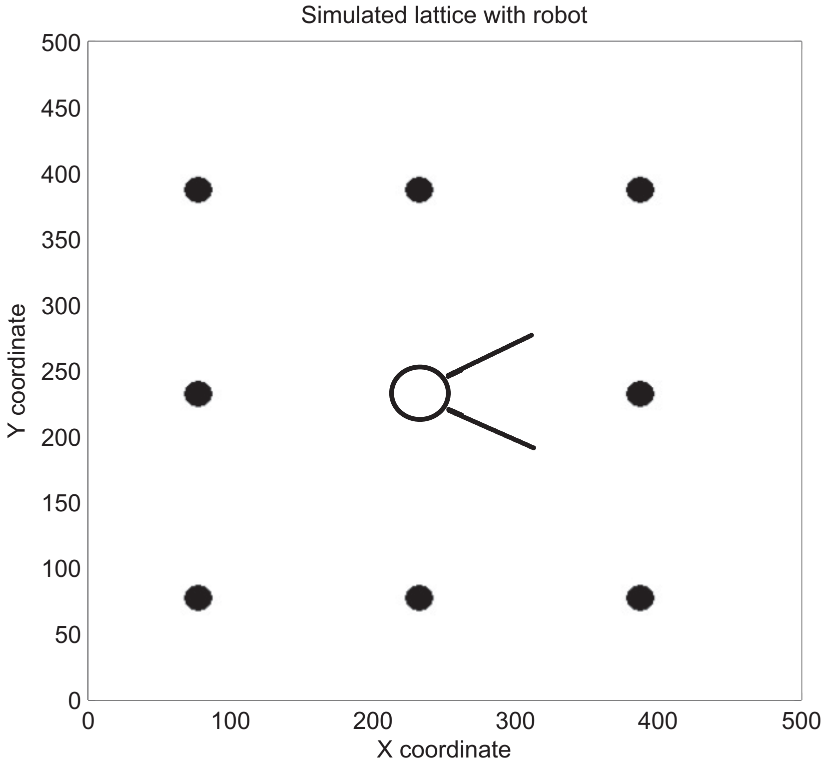

Figure 3: Example of the simulation setup.

where r_x, r_y and any derived coordinates for the sensor signals are stored as floating point values. The coordinates are rounded to integer values when used to determine whether a collision has occurred (for example), but their floating point values are retained for future calculations. The steering angle v is added to θ every time step: $\theta(t+1) = \theta + v$. Obstacles are coded as nonzero values in the grid. The sensor signals x_d and x_p are generated by probing the pixel values along the left and right antenna coordinates and calculating their differences (see equation 2.1). The resulting differences for the proximal and distal sensors are then fed into first-order low-pass filters (see equations 2.2 and 2.3) and then summed to generate the new steering angle (see equation 2.5). Learning is implemented using equation 2.6. The simulation environment is implemented in Matlab (Matlab, 2010).

Various simulations were run using this system. The results of some of these simulations are plotted in Figure 4. For each simulation, the agent is positioned in the center of the environment, facing a varying number of degrees clockwise from north.

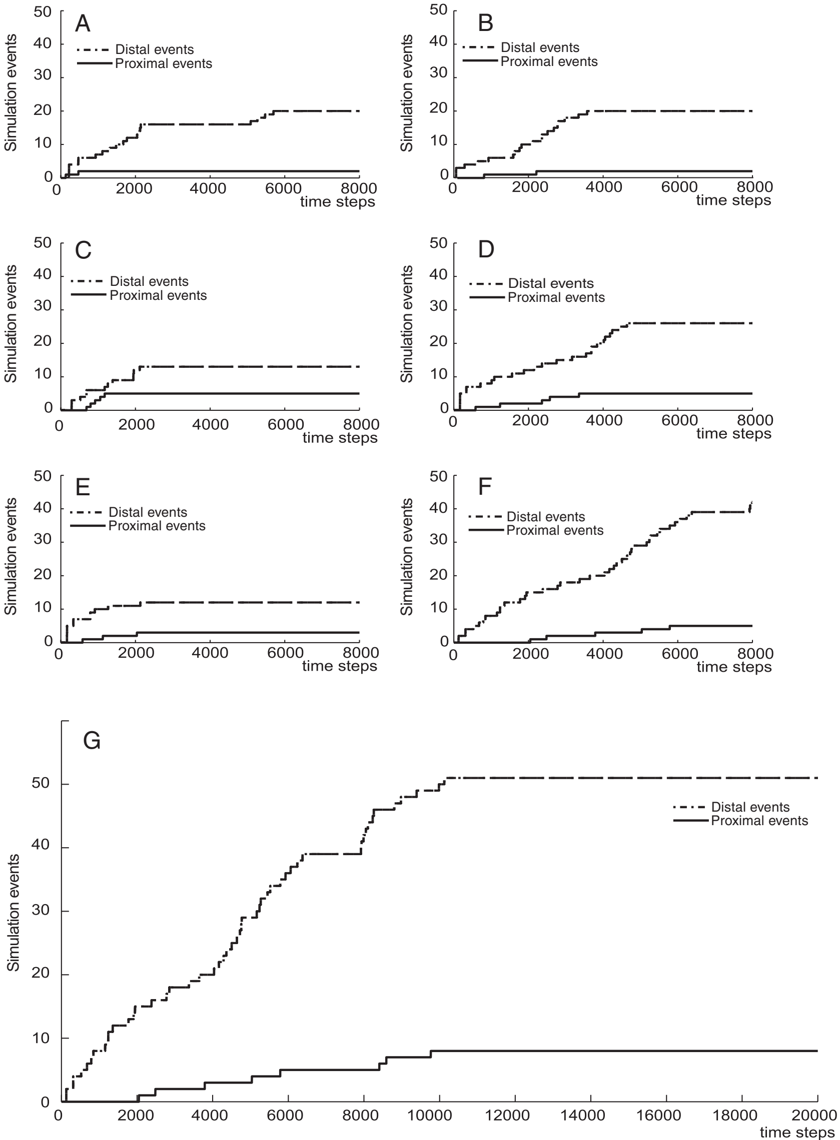

Figure 4: Simulation runs for a range of starting directions. (A) 0°, (B) 15°, (C) 30°, (D) 40°, (E) 50°, (F) 58°, (G) 58°, with extended running time.

The graphs in Figure 4 plot the total number of impacts on the distal and proximal antennae over time, for a range of starting directions (i.e., the angle from north faced by the robot). Note that the weight development

follows exactly the curve for the accumulated proximal events (scaled using λ). This is due to the fact that (in our simplified system) the weight increases every time the proximal sensor is been triggered.

In the graphs, the distal and proximal values are initially close together, as the system cannot yet avoid using its proximal signals. As the agent learns to use its distal antennae, the distal and proximal total impacts begin to diverge. This is an indication that the agent is avoiding colliding with its proximal antennae by learning avoidance behavior. Eventually the proximal total stays constant, indicating that the agent is no longer colliding. This is because the robot eventually finds a path between the obstacles and, after reaching the boundary, emerges along the same path. This causes continuous collision-avoiding behavior and is both desired and expected.

The simulation where the agent begins facing 58 degrees from north (see Figure 4F) appears to be an exceptional case. In this graph, the proximal value continues to rise. We consider this case in detail in section 5.2.

3.2 Model Checking versus Simulation. In this letter, we demonstrate how model checking can be used to verify properties of a system comprising a robot moving in an environment. The environment we use is identical to that used for simulation, as described in section 3.1, so we can compare the two approaches. Our system is simple and subject to a number of assumptions. Indeed either approach requires assumptions to be made. The important issue is that the same assumptions are made in all cases, so that a fair comparison can be made. Our goal here is to illustrate the technique rather than present a comprehensive suite of models. We explain how our approach can be extended to other environments, or to situations involving more robots, or a rigid boundary wall in section 7. We illustrate the relationship between classical closed-loop simulation and our approach in Figure 5.

Model checking involves analysis of a state-space (a graphical representation of all possible states reached by the system and the transitions between them). While in the classical simulation, we could implement all variables as floating point numbers (given analytical expressions of the entire environment), in model checking, we need to discretize variables so that we can set up our state-space. Generally the granularity of any discretization of a robot simulation is determined by the signal-to-noise ratio of the sensors of the real robot and imperfections of its actuators (Grana, 2007; Tronosco, Sanches, & Lopez, 2007; Chesi, 2009). This holds true for both classical simulation and model checking.

Simulation is equivalent to examining individual paths through a state-space. Exhaustive simulation (to cover all eventualities) is either very time-consuming or impossible. Model checking allows us to examine all possible paths and to precisely express the property of interest (rather than relying on observation of simulation output). As illustrated in Figure 5, a single simulation is equivalent to a single path in our model. Often a simulation

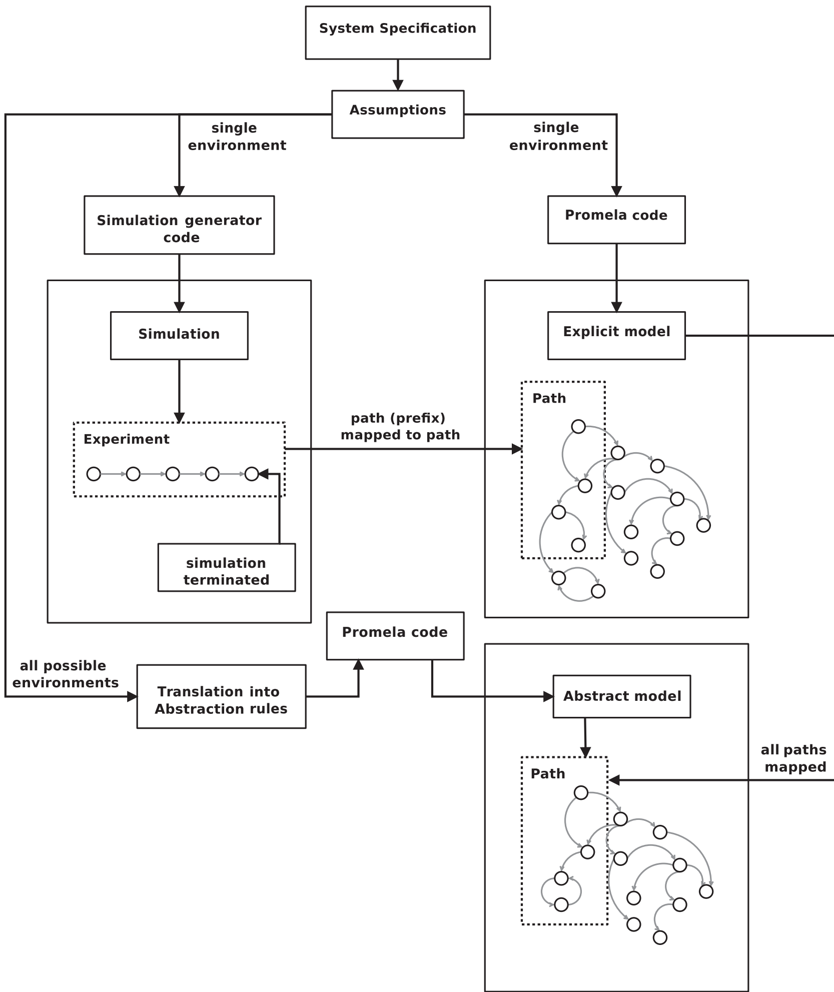

Figure 5: Comparison of approaches.

run is equivalent to a prefix of a path in our model (consisting of the first N states of the path for some finite number N). A simulation is necessarily terminated at some point, whereas verification involves exploring all paths until there are no further states or a cycle is detected.

Both simulation and model checking involve a degree of abstraction. The user of the technique must decide which aspect of the system to represent in the simulation or model. Our initial model (the Explicit model) is deliberately abstracted to the same degree as the simulation setup, so no additional information is lost. It is therefore straightforward to infer that we are

modeling the same thing in each case. The power of model checking in this case is that we can formally define a property and automatically check every path. Note that in the single robot model, there are few decision points in our model (and so there are few paths), but in general a state-space contains many paths. For example, if there were multiple robots, the ordering of steps taken by the different robots would lead to different paths with different outcomes.

Having demonstrated the power of model checking with our Explicit model, we introduce the Abstract model, which is a far more compact model and not only merges symmetrically equivalent views (from the robot's perspective) but combines several equivalent environments into the same model. There are two major benefits to this type of abstraction. The Abstract specification is a much neater representation than the Explicit specification; for example, fewer individual transitions need to be considered. In addition, results of verification hold for all environments considered in the single model, which avoids the need to repeat the same experiments for similar but different environments. A drawback of the approach is that it requires expert knowledge of the system (e.g., intimate prior experience with the Explicit model). In addition, it differs so greatly from the physical system (and the simulation environment) that complex mathematical proof is required to ensure that the abstraction is sound (i.e., that it preserves the properties in question).

A comparison of model checking and closed-loop simulation, applied to this system, is presented in section 8.

3.3 Formal Definitions. In order to be able to reason about our models, we need formal semantics. We define a Kripke structure (Kripke, 1963) as the formal model of our system. Note that for model checking, we do not need to be aware of the underlying semantics (indeed, the model checker represents the system as a Büchi automaton; Büchi, 1960; see section 4). However, to prove that our Abstract model preserves LTL (linear time temporal logic) properties (in section 6.1) we will reason about the underlying Kripke structures of our Promela programs.

Definition 1. *Let AP be a set of atomic propositions. A Kripke structure over AP is a tuple $\mathcal{M} = (S, s_0, R, L)$ where S is a finite set of states, s_0 is the initial state, $R \subseteq S \times S$ is a transition relation, and $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state. We assume that the transition is total, that is, for all $s \in S$, there is some $s' \in S$ such that $(s, s') \in R$.*

A path in $\mathcal{M}$ is a sequence of states $\pi = s_0, s_1, \ldots$, such that for all i, $0 \leq i$, $(s_i, s_{i+1}) \in R$.

When AP is a set of propositions defined over a set of variables X (e.g., $AP = \{(x == 4), (y + z <= 3)\}$, we say that $\mathcal{M}$ is a Kripke structure over X.

Table 1: Common LTL Properties.

Property	Common name	Description
$p \rightarrow q$		If p is true at a state, then q is true at that state
$[]p$	invariance	p is true at every state
$\langle\rangle q$		q is true eventually
$p \cup q$		q will eventually be true. p will be true at the initial state and will remain true until q becomes true
$p \rightarrow \langle\rangle q$	response	If p is true at a state, then q will be true either at that state or at a later state in the path

The logic CTL^* is defined as a set of state formulas (properties that hold from a given state) and a set of path formulas (i.e., properties that hold along a given path), which are defined inductively below. The quantifiers A and E are used to denote for all paths, and for some path, respectively (where, if $\neg$ denotes negation, for path property ϕ, $E\phi = \neg A \neg \phi$). In addition, X (*nexttime*) denotes *in the next state*, and $\langle\rangle$ and $[]$ represent the standard *eventually* and *always* operators (used to indicate that a proposition is true for every state in a path or true at some state in a path, respectively). The binary operator $\cup$ denotes *until*, where $p \cup q$ states that proposition p is true in the current state and continues to be true until a state is reached at which proposition q is true (and such a state will eventually be reached). Note that $\langle\rangle\phi = true \cup \phi$ and $[]\phi = \neg\langle\rangle\neg\phi$.

Let AP be a finite set of propositions. Then if $\wedge$ and $\vee$ denote the usual *and* and *or* respectively,

- For all $p \in AP$, p is a state formula.
- If ϕ and ψ are state formulas, then so are $\neg\phi$, $\phi \wedge \psi$ and $\phi \vee \psi$.
- If ϕ is a path formula, then $A\phi$ and $E\phi$ are state formulas.
- Any state formula ϕ is also a path formula.
- If ϕ and ψ are path formulas, then so are $\neg\phi$, $\phi \wedge \psi$ and $\phi \vee \psi$, $X\phi$, $\phi \cup \psi$, $\langle\rangle\phi$ and $[]\phi$.

The logic LTL (Pnueli, 1981) is obtained by restricting the set of (CTL^*) formulas to those of the form $A\phi$, where ϕ does not contain A or E. When referring to an LTL formula, one generally omits the A operator and instead interprets the formula ϕ as "for all paths ϕ."

For a model $\mathcal{M}$, if the LTL formula ϕ holds at a state $s \in S$, then we write $\mathcal{M}, s \models \phi$ (or simply $s \models \phi$ when the identity of the model is clear from the context).

We assume p, q, and r are propositions; some common LTL properties are given in Table 1. In each case, the common name for the property is given if such a name exists. Note that we omit the criterion *for every path* in the description of the properties, as this is implied for all LTL properties.

(For example, []p should be read as "for every path p holds at every state.") More examples of common LTL property patterns can be found in Dwyer, Avrunin, and Corbett (1998).

3.4 Büchi Automata and LTL. One of the most efficient algorithms for model checking LTL properties is the automata-theoretic approach (see section 4.1). Although we will not describe the algorithms in detail, we provide a little background theory here.

Definition 2. *A finite state automaton (FSA) $\mathcal{A}$ is a tuple $\mathcal{A} = (S, s_0, L, T, F)$ where:*

1. *S is a nonempty, finite set of states.*
2. *$s_0 \in S$ is an initial state.*
3. *L is a finite set of labels.*
4. *$T \subseteq S \times L \times S$ is a set of transitions.*
5. *$F \subseteq S$ is a set of final states.*

A run of $\mathcal{A}$ is an ordered, possibly infinite sequence of transitions

$$(s_0, l_0, s_1), (s_1, l_1, s_2), \ldots$$

where $s_i \in S$ and $l_i \in L$ for all $i > 0$. An accepting run of $\mathcal{A}$ is a finite run in which the final transition (s_{n-1}, l_{n-1}, s_n) has the property that $s_n \in F$.

In order to reason about infinite runs of an automaton, alternative notions of acceptance (e.g., Büchi acceptance) are required. We say that an infinite run (of an FSA) is an accepting ω-run (it satisfies Büchi acceptance) if and only if some state in F is visited infinitely often in the run. A Büchi automaton is an FSA defined over infinite runs (together with the associated notion of Büchi acceptance).

Every LTL formula can be represented as a Büchi automaton (see, Wolper, Vardi, & Sistla, 1983; Vardi & Wolper, 1994).

4 Model Checking

Errors in system design are often not detected until the final testing stage, when they are expensive to correct. Model checking (Clarke & Emerson, 1981; Clarke, Emerson, & Sistla, 1986; Clarke, Grumberg, & Peled, 1999) is a popular method that helps to find errors quickly by building small, logical models of a system that can be automatically checked.

Verification of a concurrent system design by temporal logic model checking involves first specifying the behavior of the system at an appropriate level of abstraction. The specification $\mathcal{P}$ is described using a high-level formalism (often similar to a programming language) from which an associated finite state model, $\mathcal{M}(\mathcal{P})$, representing the system is derived. A requirement of the system is specified as a temporal logic property, ϕ.

A software tool called a model checker then exhaustively searches the finite state model $\mathcal{M}(\mathcal{P})$, checking whether ϕ is true for the model. In LTL model checking, this involves checking that ϕ holds for all paths of the model. If ϕ does not hold for some path, an error trace or counterexample, is reported. Manual examination of this counterexample by the system designer can reveal that $\mathcal{P}$ does not adequately specify the behavior of the system, that ϕ does not accurately describe the given requirement, or that there is an error in the design. In this case, either $\mathcal{P}$, ϕ, or the system design (and thus also $\mathcal{P}$ and possibly ϕ) must be modified and rechecked. This process is repeated until the model checker reports that ϕ holds in every initial state of $\mathcal{M}(\mathcal{P})$, in which case we say $\mathcal{M}(\mathcal{P})$ satisfies ϕ, written $\mathcal{M}(\mathcal{P}) \models \phi$.

Assuming that the specification and temporal properties have been constructed with care, successful verification by model checking increases confidence in the system design, which can then be refined toward an implementation.

4.1 LTL Model Checking. The model checking problem for LTL can be restated as, "Given $\mathcal{M}$ and ϕ, does there exist a path of $\mathcal{M}$ that does not satisfy ϕ?" One approach to LTL model checking is the automata-theoretic approach (Lichtenstein & Pnueli, 1985; Vardi & Wolper, 1986).

In order to verify an LTL property ϕ, a model checker must show that all paths of a model $\mathcal{M}$ satisfy ϕ (alternatively, find a counterexample, namely, a path that does not satisfy ϕ). To do this, an automaton $\mathcal{A}$ representing the reachable states of $\mathcal{M}$ is constructed together with an automaton $\mathcal{B}_{\neg\phi}$, which accepts all paths for which $\neg\phi$ holds. The asynchronous product of the two automata, $\mathcal{A}'$ is constructed. In practice, $\mathcal{A}'$ is usually constructed implicitly by letting $\mathcal{A}$ and $\mathcal{B}_{\neg\phi}$ take alternate steps. Whenever a transition is executed in $\mathcal{A}$, the propositions in ϕ are evaluated to determine which transitions are enabled in $\mathcal{B}_{\neg\phi}$. If a path in $\mathcal{A}$ does not satisfy ϕ, the automaton $\mathcal{B}_{\neg\phi}$ may execute a trace along a path in which it repeatedly visits an acceptance state. This is known as an accepting run (of $\mathcal{A}'$), and it signifies an error. If there are no accepting runs, the property holds, and $\mathcal{M} \models \phi$. Generally to prove LTL properties, a depth-first search is used. As the search progresses, all states visited are stored (in a reduced form) in a hash array (or heap) and states along the current path are pushed on to a stack. If an error path is found, a counterexample can be produced from the contents of the stack.

In Figure 6 we give Büchi automata for LTL properties $[]p$ (p is true at every state) and $\langle\rangle q$ (q is true eventually). Note that any path of an automaton $\mathcal{A}$ has associated paths in the Büchi automaton. For example, consider the Büchi automaton of Figure 6A. If π is a path in $\mathcal{A}$ for which p becomes false at some state, s, say, it would be possible to loop around the state labeled `T0:init` until s is reached, then make a transition to the (acceptance) state labeled `accept_all`. The infinite continuation of π would result in infinite looping around the acceptance state in the Büchi automaton. Thus, π would

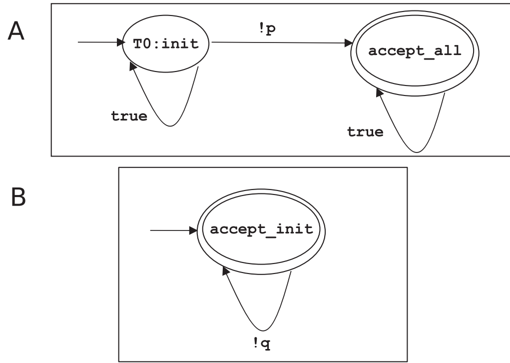

Figure 6: Example Büchi automata. (A) $[]p$. (B) $\langle\rangle q$.

be accepted. Similarly, a path π' in $\mathcal{A}$ for which q is never true would be accepted by the Büchi automaton of Figure 6B. Note that the names of the states are not significant (although an acceptance state is generally prefixed with the term `accept`). The Büchi automaton in this example was generated using Spin.

When we use model checking to prove properties of a system, the underlying automata are constructed by the model checker itself. The user must supply a specification of the system that is recognizable as a true representation of the system and the translation to automata is unseen. In section 4.2, we introduce Promela, the specification language for the model checker Spin.

4.2 Promela and Spin. The model checker Spin (Holzmann, 2004) allows one to reason about specifications written in the model specification language Promela. Spin has been used to trace logical errors in distributed systems designs, such as operating systems (Cattel, 1994; Kumar & Li, 2002), computer networks (Yuen & Tjioe, 2001), railway signaling systems (Cimatti et al., 1997), wireless sensor network communication protocols (Sharma et al., 2009), and industrial robot systems (Weissman, Bedenk, Buckl, & Knoll, 2011).

Promela is an imperative-style specification language designed for the description of network protocols. A user of Spin does not see the Büchi automata associated with their Promela specification. The Promela specification is a clear and understandable representation of the system to be modeled. Indeed, since Promela syntax is close to C-code, a Promela specification is often very close to the implementation of the system to be modeled. Underlying semantics allow a Promela specification and an LTL property to be converted into their respective automata and combined as described in section 4.1. The specification can be relatively short, whereas the associated

Büchi automata can contain thousands (or indeed millions) of states. It is not therefore feasible to construct the Büchi automata by hand.

In general, a Promela specification consists of a series of global variables, channel declarations, and `proctype` (process template) declarations. Individual processes can be defined as instances of parameterized proctypes. A special process—the `init` process—can also be declared. This process will contain any array initializations, for example, as well as run statements to initiate process instantiation. If no such initializations are required and processes are not parameterized, the `init` process can be omitted and processes declared to be immediately active via the `active` keyword. Properties are specified using `assert` statements embedded in the body of a proctype (e.g., to check for unexpected reception), an additional monitor process (to check global invariance properties), or by LTL properties. We do not give details of Promela syntax here but illustrate the structure of a Promela program and some common constructs by way of our example system in appendix B. LTL properties that are to be checked for the system are defined in terms of Promela within a construct known as a `never claim`. A never claim can be thought of as a Promela encoding of a Büchi automaton representing the negation of the property to be checked.

Spin creates a finite state automaton for each process defined in a Promela specification. It then constructs the asynchronous product, A, of these automata and a Büchi automaton $\neg B$ corresponding to any never claim defined. As described in section 4.1, in practice A and $\neg B$ are executed in alternate steps—the propositions in $\neg B$ being evaluated with respect to the current values of the variables in A. Automaton A can be thought of as a graph in which the nodes are states of the system and in which there is an edge between nodes s_1 and s_2 if at state s_1, some process can execute a statement (make a transition) that results in an update from state s_1 to state s_2.

The system that we are modeling here is not concurrent: there is just a single robot moving in an environment. However, our model does involve nondeterministic choice. When a head-on collision occurs, the robot moves to the left or right of the obstacle. When the collision occurs at the farthest point on the shell from the center of the robot (i.e., at a point equidistant from the antennae), the direction of movement is chosen nondeterministically. Spin allows us to check every path through a model for counterexamples (i.e., paths that violate a given property) without having to manually construct a set of test cases. This includes infinite (looping) behavior, which cannot possibly be checked using simulation alone. Future work (see section 7) will involve us adding robots. This will be a simple case of adding further instantiations of the robot process template.

In our model, it is important that the movement of the robot in its environment is represented as accurately as possible. Due to the limitations of the Promela language, this precision is not possible with Promela alone. However, it is possible to embed C code within a Promela specification.

Note that the calculations performed in the C code are visible to the Promela specification and the values of variables contained in it used to determine transitions in the automata. However, we can choose that some variables used for intermediate calculations that are not relevant (i.e., do not influence transitions) are not visible and do not form part of each state. In appendix A, we describe how embedded C code is used in our Promela specification.

5 The Promela Specification: Explicit Model

In this letter we describe two Promela specifications, the Explicit specification, which describes a low-level representation of the system for a particular environment, and the Abstract specification, which allows us to capture all paths of a robot in any environment (with some restrictions). The associated models are the Explicit model and the Abstract model. Figure 5 illustrates the relationship between our models and the relationship between simulation and model checking.

The Explicit specification is so low level that it closely resembles simulation code. This is an advantage: it is easy to convince system designers that our specification (and hence the resulting model) is correct. Verification of the underlying model is more powerful than simulation alone but is restricted to proving properties for a single environment. In addition, the state-space associated with such an unabstracted model can be prohibitively large. (This is not true in our case but applies to systems with a high level of concurrency and nondeterminism.) The Abstract specification is much further removed from the simulation code and requires expert knowledge to construct. The benefit of the Abstract model is that we can verify properties for any environment (under the given assumptions), and memory and time requirements are much smaller (than the combined requirements for all environments). However mathematical proof is required to show that the Abstract model does indeed capture the behavior of a single robot in any suitable environment.

In this section we describe the Explicit specification.

5.1 Assumptions. We assume that the environment, the robot, and the obstacles are all circular (see Figure 7). The robot setup is illustrated in Figure 7. The length of an antenna is 60 units, and the angle between the antennae is 60 degrees. The diameter of the robot is 40 units (see Figure 7C), and the diameter of an obstacle is twenty units (see Figure 7B). The environment has diameter at least 100 units—the radius of the robot plus the length of an antenna plus the diameter of an obstacle (see Figure 7A).

We assume that the complexity of the environment is such that at most, one obstacle can touch any part of the robot at any time. We denote the minimum allowable distance between obstacles as δ_0.

An environment is a circular region, represented by a set of polar coordinates $C = \{(r, \theta) : 0 \leq r \leq \rho, 0 \leq \theta < 360\}$, where ρ is the radius of

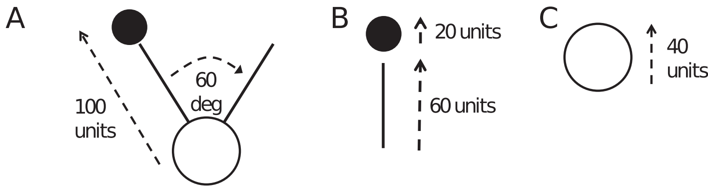

Figure 7: Robot setup. (A) Distance from the center of robot to the far edge of the obstacle. (B) Lengths of the antenna and obstacle. (C) Diameter of the robot.

the environment and angles are measured clockwise from north. We use polar coordinates to represent the environment and the current position of the robot and the obstacles. This allows us to store the angular information of the system (without having to recalculate this from Cartesian coordinates at every point) and so determine the robot's new position when moving at an angle that is not parallel to either Cartesian axis. The use of polar coordinates also allows us to represent the turning angles of the agent to an accuracy of 1 degree, a level of accuracy that we deemed acceptable.

The robot is initially placed in the center of the environment, facing a given direction. Since the robot is the only moving obstacle in the environment, the state of the system reflects the position of the robot, the direction in which it is moving, and (by implication) at what point (if any) an obstacle touches either of the sensors. We do not include the robot's motors or external wheels in the model.

The precise location of the robot as it moves around an environment is calculated using *C*-code embedded within our Promela specification. At each time step, the new direction of the robot is calculated from the signals received from the sensors. We simply calculate the position of any obstacle touching the sensors to infer this information. As well as deciding the new direction of the robot, the angular response to a sensor impact will be incremented by a fixed amount (the learning rate) if a collision occurs at the proximal sensor after a collision has occurred at a distal sensor. If a head-on collision occurs, the robot moves to the left or right of the obstacle. When the collision occurs at the farthest point on the shell from the center of the robot (at a point equidistant from the antennae), the direction of movement is chosen nondeterministically.

As in the simulated environment, we ignore the presence of any boundary wall. When a robot reaches the perimeter of the environment, it is simply relocated to the opposite edge of the perimeter. This is achieved via a `WRAP` function that reflects the position of the robot about a ray, $r2$, that runs through the pole (center of the environment) and is perpendicular to the ray r_1 that runs through the position p of the robot at the angle that the robot is facing. The new position of the robot is p', where p and p' are at the same distance from the two points of intersection of r_1 with the perimeter of the

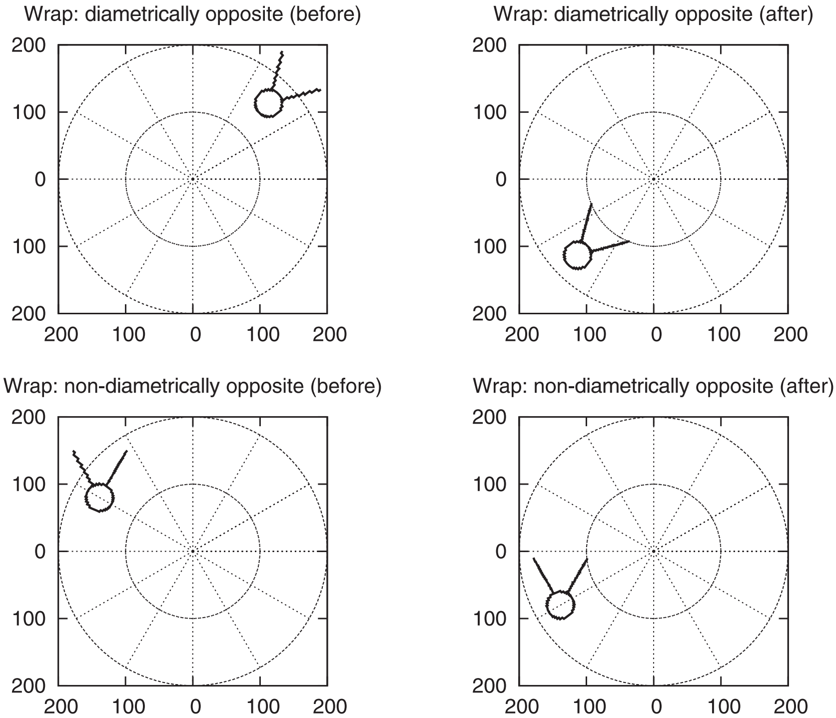

Figure 8: Examples of the `WRAP` function.

environment. Note that the robot continues to face in the same direction. Figure 8 illustrates the effect of the `WRAP` function when the new position of the robot is diametrically opposite to the old position and when it is not. Our particular implementation of the movement of the robot at the boundary reflects the implementation in the closed-loop simulation that in both cases was chosen to simplify our description and the introduction of our approach. Other implementations are possible, as discussed in section 7.

We include sample Promela code in appendix B.

5.2 Verification Results for the Explicit Model. Our initial experiments with Spin involved attempting to reproduce the results of the simulations of Figure 4. In particular, we were interested in the results of the simulation where the agent initially faces 58 degrees from north (see Figure 4). As observed in section 3.1, in this case, the learning weight (ω_d) continues to rise, apparently indefinitely. This continual rise is counterintuitive. We examined this case using model checking. Our specification was for a robot initially facing due north, and the environment adapted appropriately. Using Spin, we verified that the system should always stabilize. This

is in fact property 2, which is given (along with details of verification) below. Analyzing this inconsistency in more detail, we suggest the simulation of Figure 4A should simply be run for longer. The results of this extended simulation are shown in Figure 4G. Running the extended simulation shows that the proximal value does eventually stabilize, indicating that learning has ceased and successful avoidance behavior has been achieved. Note that this illustrates an advantage of model checking over simulation. Whereas in simulation the user decides when to stop waiting for stabilization to occur, the model checker automatically checks all possible outcomes. The model checking process does not terminate until all possible paths have been explored, however unlikely they may be. This is illustrated in Figure 5: a simulation run maps to a prefix of a path in the state-space.

We now give details of how we verified both of the properties identified in section 2:

1. The sensor input x_p of the proximal sensor will eventually stay zero, indicating that the agent is using only its distal sensors.
2. The weight ω_d will eventually become constant, indicating that the agent has finished learning.

Our models are defined separately for each learning rate λ and environment (i.e., location of obstacles). Using this model, we cannot verify properties for any environment; we must construct a different model (using a different Promela specification) for every environment. We fix our learning rate to 1 and verify our properties for an example set of environments.

Environments E1 to E6 are shown in Figure 9. These environments have obstacles placed at random, at a minimum distance of δ_0 from each other (see section 5.1). Note that the positioning of the obstacles is further restricted by the `WRAP` function (see Table 4) in two ways. First, when an obstacle is randomly placed, its minimum distance from other obstacles must take into account the wrapping of the environment. Second, obstacles cannot be placed so close to the perimeter that the `WRAP` function could cause the robot to wrap directly into it.

The experiments were conducted on a 2.5 GHz dual core Pentium E5200n processor with 3.2 Gb of available memory, running UBUNTU (9.04) and SPIN 6.0.1.

To prove property 1, we used the LTL formula $\langle\rangle[]p$ (in all paths p is eventually always true) where p is defined to be the proposition, $((sig \neq 6)\&\&(sig \neq -6))$. Note that *sig* and *PrevSig* are the variables we use in our Promela specification to denote the current signal from the sensors and the previous signal from the sensors respectively. The latter has default value 0 and is reset to this value when a proximal signal is received.

Attempts to verify this property using Spin proved the property to be false. Examination of a counterexample trace showed that it was indeed always possible to have an impact on a proximal sensor, even when learning

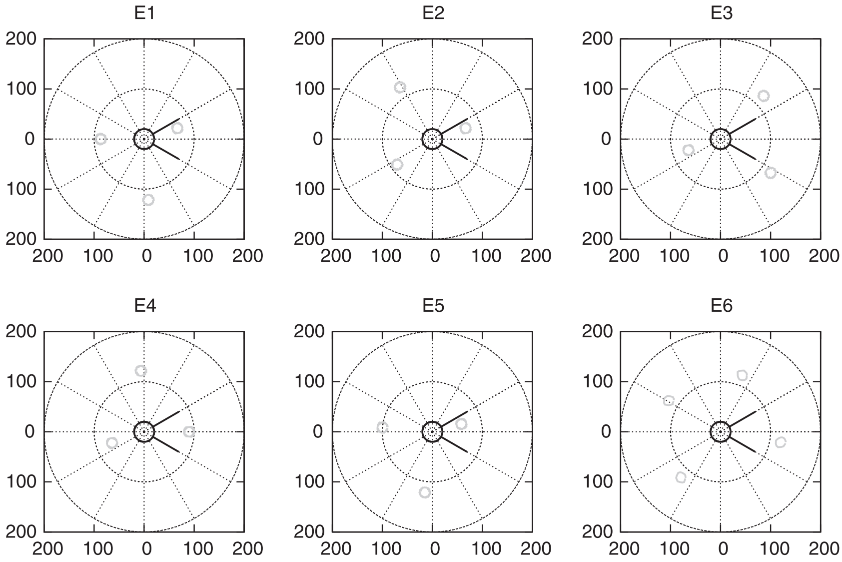

Figure 9: Environments E1 to E6.

had ceased. This happens when the robot approaches an obstacle head-on, and the obstacle has an impact without the distal sensor touching the obstacle. This situation was missed during simulation. Of course, on another day, a different simulation may have exposed this possibility. The benefit of model checking here is that it never misses an error path, although, of course, it is the definition of the property being checked that determines what an error path is.

Since learning occurs only when an impact on the proximal sensor follows an impact on the distal sensor, we can rephrase the property to eliminate this rare behavior. The new property (property 1A) is $\langle\rangle[](!p \rightarrow !q)$ (eventually p is always true unless q is false), where p is defined as above and q is defined to be $((prevSig > -6)\&\&(prevSig < 6))$. This property is shown to be true for our set of example environments (see Table 2 for verification results). Note that the Stored States column gives an indication of the size of the underlying state graph. As the graph is explored (during any verification), states are generated on the fly from the transitions indicated in the Promela specification. When a new state is encountered, it is stored (in the state-space). When a previously visited state is encountered, the search backtracks. Maximum search depth is the length of the longest path that is explored during search, and time denotes the time (in seconds) taken for verification.

To prove property 2, in each case we performed initial experiments to find the maximum value of ω_d (call this value *Max*). These experiments

Table 2: Verification Results for the Explicit Model.

Environment	Property	Max ω_d	Stored States	Maximum Search Depth	Time (sec)
E1	1A	1	273,664	547,323	2.19
	2A		273,734	547,323	2.26
E2	1A	1	150,562	300,979	1.03
	2A		150,492	300,979	1.04
E3	1A	0	670	1339	0.00
	2A		670	1339	0.00
E4	1A	0	77,034	154,067	0.16
	2A		77,034	154,067	0.28
E5	1A	1	218,326	436,647	1.36
	2A		218,372	436,647	1.73
E6	1A	1	61,256	122,507	0.28
	2A		61,434	122,507	0.52

involved choosing an initial (high) value of *Max* and checking a further LTL property $[]\omega_d < Max$. When the property is proved true, *Max* is decremented and the process repeated, until the property is shown to be false, in which case *Max* is fixed to the last value for which the property was shown to be true. We checked a slightly modified version of property 2, namely, property 2A, to verify that *MAX* is eventually reached, but never exceeded. This is verified using the following LTL property: $(\langle\rangle(omegaD == MAX))\&\&([]omegaD <= MAX)$. This property was shown to be true for our set of example environments (see Table 2).

6 The Abstract Specification

The Promela specification described in section 5 models the physical world explicitly; it represents an explicit environment and an explicit robot. In this section we describe an Abstract specification in which the environment is abstracted to a much smaller area, known as the cone of influence surrounding the robot. Rather than move the robot around an Explicit environment, at any state we consider only the position of any obstacle within this cone of influence and its position relative to the robot. Depending on any impact made to the sensors of the robot at a given state, the next state is calculated. In this case rather than the robot move, the robot stays fixed in its original position (at the origin) and any obstacle moves relative to the robot. The advantage of this specification over the Explicit specification is that the model represents a robot moving in any environment with distance between obstacles at least δ_0 (as defined in section 5.1). The relationship between our models and between simulation and our models is illustrated in Figure 5.

As well as the usual benefits of model checking, the Abstract model allows further benefits over simulation in that it allows us to analyze a set of

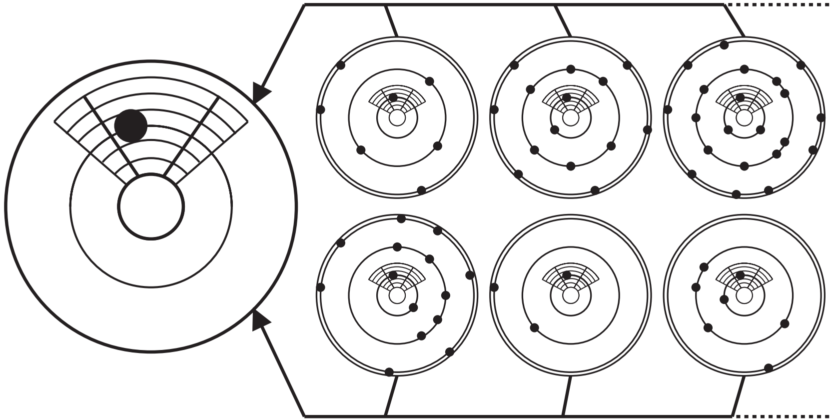

Figure 10: Abstraction of environments to a single representation. From a given position of the robot, several environments look identical. Shown are states in six differerent environments that correspond to the same state in the Abstract model.

environments in a single verification. Of course, we need to define assumptions on our environment (in the same way that we would need to define restrictions on a simulation environment). The environments represented by a single abstract model must all be equivalent in some way. In our case this equivalence is determined by the minimal distance between obstacles; it could, of course, be defined differently. In section 7 we describe how the Abstract model could be extended to more complex scenarios (e.g., a range of distances between obstacles).

To see how the Abstract model represents multiple environments, consider Figure 10. For a given position of the robot, several environments look identical from the robot's perspective (i.e., within its cone of influence). Our abstraction merges these symmetrically equivalent cases. In our case, only one obstacle can appear within the cone of influence, and equivalent cases are determined by the distance and angle of any obstacle from the antennae. If there were more obstacles, symmetry would still exist between different scenarios (see section 7). Note that our abstraction also merges situations in which the robot is in different positions, but its view within its cone of influence is the same. This is not illustrated in Figure 10.

From a state in which there is an obstacle in the cone of influence, the next state is calculated in the same way as it is for the Explicit specification. However, if there is no obstacle within the cone of influence (i.e., the robot is in free space), nondeterministic choice determines the position (if any) of an obstacle appearing at the front of the cone of influence in the next state.

We refer to the model (i.e., a Kripke structure) associated with the Abstract specification as the Abstract model. In section 6.1 we give an outline

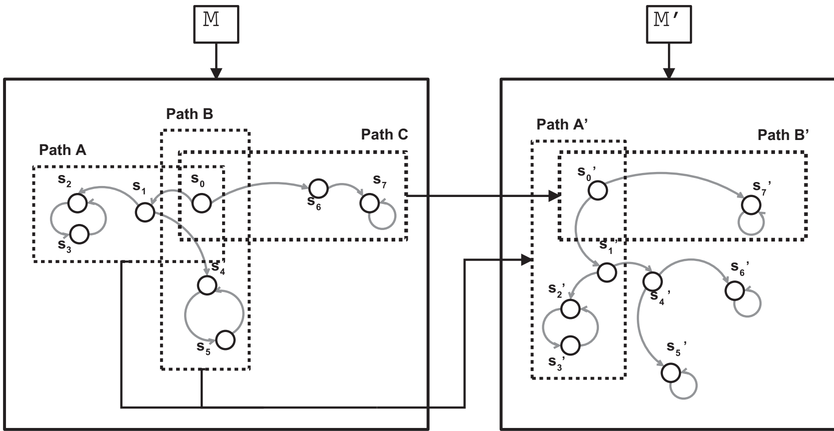

Figure 11: Simulation relation between models.

proof to show that for any suitable environment E, any LTL property satisfied by the Abstract model is satisfied by an Explicit model with environment E.

We give outline code for the Abstract specification in appendix C.

6.1 Justification for the Abstract Model. We need to show that for a given learning rate λ, by verifying an LTL property for the Abstract model with learning rate λ we can infer that ϕ holds for all Explicit models with learning rate λ. In this section we use the term *model* to denote the underlying Kripke structure (see definition 1) associated with a Promela specification.

Our justification is based on the concept of simulation between two Kripke structures $\mathcal{M}$ and $\mathcal{M}'$. A simulation relation R between the sets of states of $\mathcal{M}$ and $\mathcal{M}'$ is a set of pairs of states, (s, s') where s and s' are states of $\mathcal{M}$ and $\mathcal{M}'$, respectively, and any transition in $\mathcal{M}$ is matched to a transition in $\mathcal{M}'$. Formally, for any transition (s, s_1) in $\mathcal{M}$, if $(s, s') \in R$, then there is a transition (s', s'_1) in $\mathcal{M}'$ where $(s_1, s'_1) \in R$.

We say that $\mathcal{M}'$ simulates $\mathcal{M}$ if there is a simulation R between the sets of states and the initial states of $\mathcal{M}$ and $\mathcal{M}'$ are in the relation. For example, in Figure 11, a simulation relation is given by

$$R = \{(s_0, s'_0), (s_1, s'_1), (s_2, s'_2), (s_3, s'_3), (s_4, s'_2), (s_5, s'_3), (s_6, s'_7), (s_7, s'_7)\}.$$

For every path in $\mathcal{M}$, there is a corresponding path in $\mathcal{M}'$; in this case, we say that a path π in $\mathcal{M}$ is matched to a corresponding path π' in $\mathcal{M}'$. Note that in Figure 11, paths A and B in $\mathcal{M}$ are both matched to path A' in $\mathcal{M}'$ and path C in $\mathcal{M}$ is matched to path B' in $\mathcal{M}'$. Crucially, several paths in $\mathcal{M}$ can be matched to a single path in $\mathcal{M}'$, and not all paths in $\mathcal{M}'$ need

to be matched to paths in $\mathcal{M}$. Any LTL property that holds for (all paths in) $\mathcal{M}'$ holds for (all paths in) $\mathcal{M}$.

Let $\mathcal{M}$ and $\mathcal{M}'$ denote an Explicit model and the Abstract model, for a given learning rate λ. It is possible to prove that there is a simulation relation between the Explicit model (i.e., for an example environment) and the Abstract model. Similarly, every path in the Abstract model is mapped to a path in some Explicit model. It follows that if an LTL property ϕ holds for the Abstract model, for a given learning rate λ, it will hold for every path in every Explicit model with learning rate λ. Thus, ϕ holds for every Explicit model with learning rate λ.

It is beyond the scope of this letter to give a full proof that there is a simulation relation for every explicit model. However, we describe the general technique indicating how paths are matched. We have used this approach in previous work (Miller, Calder, & Donaldson, 2007).

The Explicit and Abstract Promela specifications are written in a way that can easily be translated into a form known as Guarded Command Form. The robot process in each case is defined using a single repeating loop in which every statement consists of an atomic step containing a guard followed by an update (or command). Thus, every statement in the Promela specification corresponds to a transition in the associated model. This allows us to easily match states and transitions in an Explicit model to corresponding states and transitions in the Abstract model (and vice versa).

For example, in each model, in the initial state the robot is at the origin and facing due north, so the initial states can clearly be matched. For any environment, at any state, the response of the robot is determined by the position of the nearest obstacle with respect to its cone of influence. If, in the Explicit model, the robot is in a state at which no obstacle is in this cone, then this state is matched to one in the Abstract model in which the robot is in free space. If an obstacle touches a sensor in the Explicit model, then this state can be matched to one in the Abstract model in which an obstacle touches the same area of the antenna.

Any transition in the Explicit model involves a change in the position of the nearest obstacle relative to the robot. This transition can be matched to a transition in the Abstract model in which the position of an obstacle moves likewise, relative to the robot.

6.2 Verification Results for the Abstract Model. In this case, only one verification is required for each property. The learning rate is again assumed to be 1. Results are given in Table 3.

7 Model Enhancements

The environments and robot behavior that were represented in both our simulation setup and Promela specifications (with their associated models)

Table 3: Verification Results for the Abstract Model.

Property	Max ω_d	Stored States ($\times 10^5$)	Maximum Search Depth	Time (sec)
1A	6	121,413	28,881	0.32
2A		118,129	28,881	0.26

were deliberately chosen to be simple. Whether creating a computer-based closed-loop simulation or a Promela specification, it is necessary to make assumptions about the system that we are modeling. In either case, we cannot have limitless possibilities about the number of obstacles or their shape. In addition, we have to decide a priori whether to consider a fixed boundary and, if so, the nature of the boundary. The purpose of this letter is to demonstrate the effectiveness of model checking (as a complementary approach to simulation), not to consider all possible environments or robot behavior. In this section we discuss how we could adapt our models to consider more complex scenarios. Note that in all cases, modifications are made to the Promela specification (and Spin will produce the underlying models in each case).

In each of the cases below, we assume that only the specified modification is to be implemented. Clearly we could combine the modifications in any way we like, but we consider only one at a time here to make our explanation simpler. For each modification, we first consider how the Explicit Promela specification would be adapted. The Explicit specification would be modified in much the same way as the simulation code would be modified. The corresponding Abstract specification in all cases would require more detailed consideration.

When considering a new model, we always start by using an Explicit model that is close to implementation level and abstract from there (removing unnecessary variables, for example). Creating, what we refer to as an Abstract model requires experience of the Explicit model so as to gauge what the equivalence classes are. For example, in the Abstract model considered in this letter, the equivalence classes correspond to the possible positions of a single obstacle in the cone of influence.

In each of the modifications in the following list, we indicate the corresponding equivalence class. Note that proof of soundness would involve proving that every state in a corresponding Explicit model would map to an equivalence class representative (and so to a state in the Abstract model). We do not include all possible extensions here, just indicate a few that could be implemented easily.

- **Inclusion of environment boundaries.** Boundaries can easily be included in our Explicit model. In this case, the boundary would be incorporated as a set of unreachable coordinates. The robot would

respond to a signal from its sensors resulting from a collision with a boundary in the same way as it would a collision with an obstacle. Depending on the shape of the boundary (and assuming a single obstacle), the equivalence classes in the Abstract model would correspond to the possible positions of a single obstacle and a segment of boundary in the cone of influence.

- **Arbitrary or dynamic boundaries.** Any Explicit model would assume that a boundary was fixed. However, there is plenty of scope for allowing arbitrary boundary shapes or dynamic boundaries in our Abstract model, provided, of course, that we assume (as we would do for simulation) that the possible types of boundary belong to a finite set. The equivalence classes in this case would be as for the previous example, but the number of possible different types of segment of visible boundary in the cone of influence would increase.
- **Increased complexity.** This would mean allowing there to be more than one obstacle within the cone of influence at any time. The Explicit specification could be modified to accommodate this very easily (the array containing the positions of the obstacles would simply have to be altered). Assuming that there are at most N obstacles within the cone of influence at any time, the equivalence classes (and hence the states in the Abstract model) correspond to the possible positions of up to N obstacles within the cone of influence.
- **Additional robots.** Our explicit Promela model involves a process definition of a robot and a single instantiation of that process. Adding robots would simply involve instantiating multiple robot processes (with learning, or not). Our Abstract model concerns the view of a single robot. Any additional robots would be viewed as dynamic obstacles. The behavior of other robots (whether learning or not), would be relevant only within the cone of influence (e.g., all possible movements of the other robot after a collision need to be considered).
- **Alternative learning algorithm.** Both of our Promela specifications can be adapted easily to accommodate an alternative learning algorithm. This would involve altering our C-code functions determining the progress of the robot from any state after a collision. We could use our models to compare the consequences of different algorithms.
- **Dynamic obstacles or different obstacles.** By defining obstacles as processes, they could be defined as dynamic, following either a prescribed path or a nondeterministic path. Similarly, obstacles could be defined to have a variety of shapes and sizes, provided they can be defined and constrained before modeling. In the Abstract model, it would make no difference to make obstacles dynamic; the assumption of a maximum number of obstacles within the cone of influence would be sufficient. Different shapes and sizes of obstacles in the Abstract model would require a minimal revision of the code (again requiring the different possibilities to be defined a priori).

- **Measuring explicit time.** It is not possible to represent explicit time (e.g., to measure the amount of time between events) using Spin alone, although the temporal ordering of events is clearly representable. When there is only one robot, there is a correlation between the number of global transitions between events and the time between the events. It would therefore be able to give a (discrete) representation of time using Spin in this case. However, concurrent events are executed sequentially by Spin, so when there is more than one component (i.e., robot), there is no such correlation. In order to prove quantitative properties such as time between events or the probability of an event, a more specialized model checker, such as the timed model checker Uppaal (Larsen, Patterson, & Yi, 1997) or the probabilistic model checker Prism (Hinton, Kwiatkowska, Norman, & Parker, 2006), would be required.

8 Comparison of Classical Closed-Loop Simulation and Model Checking

New strategies had to be developed to translate the behavior-based approach into a form suitable for model checking. For simulation, we used an existing framework to easily calculate the position of obstacles on the sensors, the new direction of the robot, and so on. In comparison, the Promela model was rather cumbersome, in that we had to construct a number of C functions, in addition to just using pure Promela. However, we were able to adapt the code for (the simulated) robot behavior. In order to simplify the Promela model, we kept C functions used for calculation hidden from the user (in included files). These functions can be reused in future models.

The advantage of the model-checking approach was that we could simply specify LTL properties to define behavior that was expected for all paths for our model. We did not have to run an exhaustive set of simulations to verify behavior; the model checker would find any error path if it existed. In addition, our Abstract model allowed us to check certain properties for all possible environments: if there were any distributions of obstacles for which one of our properties did not hold, the model checker would find it. Having the capacity to examine error trails allowed us to not only debug our models but to identify the pathological case in which one of the initial properties did not hold (i.e., the situation in which the robot hit an obstacle head-on, without first making contact with a distal sensor). This allowed us to strengthen the property to ignore this unusual case.

In addition, model checking allows us to identify deficiencies before, during, and after learning. That the robot cannot see obstacles that are hitting it head-on is clearly a deficiency of its sensor distribution. While simple to spot in our example, more complex sensor motor setups will make it much more difficult to identify deficiencies that might occur only rarely. However unlikely, if these cases could cause damage to the robot or a deterioration

of its performance (say), then they need to be tackled appropriately. Model checking can help here (alongside classical simulation) to identify these problems in the design phase of a robot and will lead ultimately to a more reliable system.

The main drawback of the model checking approach is that it requires expert knowledge to construct a Promela specification with just the right level of abstraction and develop LTL properties to capture identified error behavior. While the level of mathematical expertise required for our Explicit model is high, an even greater degree of theoretical knowledge is essential for the Abstract model.

9 Related Work

When model checking is used in the context of autonomous agents it has been traditionally used to verify successful communications between agents in multiagent systems (Dekhtyar, Dikovsky, & Valiev, 2003; Konur, Dixon, & Fisher, 2012). A number of methods for formally specifying multiagent systems, with a view to prototyping or verification have been proposed (Hilaire et al., 2000, 2004; Da Silva & De Lucena, 2004; Wooldridge et al., 2004; D'Inverno et al., 2004). This is a natural use of model checking that was designed primarily for communication protocol analysis. Formal aspects of multi-agent systems are the subject of an annual workshop (Formal Aspects of Multi-Agent Systems; see Dunin-Kplicz & Verbrugge, 2004, and 2009). Approaches tend to focus on protocol verification, the formalization of goals, and plans and knowledge-based agents.

Model checking has also been used to test the success of single agents, for example, whether a dynamical system can generate a trajectory navigating from a starting position to a specific target (Fainekos, Girard, Kress-Gazit, & Pappas, 2009) or if agents always perform a given task without errors (Webster, Fisher, Cameron, & Jump, 2011; Molnar & Veres, 2009; Ingrand & Py, 2002; Lerda, Kapinski, Maka, Clarke, & Krogh, 2008). However, none of these approaches involve agents with learning. Indeed, learning is considered only in the context of model checking when it is used as a way to enhance model checking algorithms (Leucker, 2007; Mao et al., 2011).

Fisher (2005) uses a temporal logic framework to specify the behavior of individual agents as well as systems of agents. Refinement is used to reason about behavior, as well as verification by logical deduction. The framework is extended to include the concepts of knowledge and belief, but learning is not considered. Model checking has been used in Bordini, Fisher, Visser, and Wooldridge (2006) in which agents are specified in the logic-based agent-oriented programming language AgentSpeak, and the specification of the system is automatically converted into Promela or Java for verification with Spin or the Java Pathfinder tool (Visser, Havelund, Brat, & Park, 2000). This

approach does not consider agent learning or model collision avoidance or use Abstraction, as we do.

To our knowledge, we are the first to introduce biologically inspired agent learning into the model checking paradigm. This has been achieved by directly using Promela and SPIN. The implemented ICO learning determines the robot's reflex (in the specification), and model checking allows us to check if the generated model is successful under all possible conditions. This substantially extends the application domain of model checking to systems that can inform the development of future models or optimize agent learning algorithms. We have presented a preliminary abstract describing the model checking aspects of our work in Kirwan and Miller (2011).

10 Conclusions and Future Work

Model checking is a powerful tool that allows us to check temporal properties of a (model of a) system. In this letter, we have shown how the Spin model checker can be used to verify properties of a system that has previously been analyzed using simulation. The system, consisting of a robot navigating around an environment using learning to avoid obstacles, serves as an instructive example for the technique of model checking and its use within this context. We have described our Promela model and how we verified some example LTL properties for it. The original properties that were assumed to hold for the system were found to be insufficient. We therefore strengthened our properties so that any error reported would accurately reflect the kind of behavior we were interested in. Our abstracted model is a powerful one: it allows us to prove properties for any environment for which no two obstacles can interfere with the sensors at any time. This model removes the need to run multiple verifications to check a property (i.e., one per environment).

The learning algorithm implemented in both the simulation environment and the Promela specification is a simplified version of temporal sequence learning. It would be straightforward to adapt both of these to implement alternative learning algorithms and provide a platform for comparison purposes. Our Promela specifications could easily be converted to Prism (the specification language of the probabilistic model checker Prism) (Hinton et al., 2006), for example, using the Prism2Promela tool (Power & Miller, 2008). Prism would be an ideal tool for analyzing probabilistic learning algorithms.

Our work has demonstrated the feasibility and value of using model checking in the context of a robot navigating around a set of obstacles in an environment. The setup cost, in terms of the transfer of knowledge between engineers and computer scientists, creation of the formal specification, and development of the temporal properties, has been high. However, all of the C functions, template specifications, and temporal properties can be reused

(or adapted) for more complex systems involving more robots or alternative learning algorithms.

Future work involves the development of a software system to automatically create a Promela specification for a system of robots in an environment, given the number of robots, the location (or number) of obstacles, and the learning algorithms used.

Appendix A: The Use of Embedded C-Code in Our Promela Specification

In our model, it is important that the movement of the robot in its environment is represented as accurately as possible. Due to the limitations of the Promela language, this precision is not possible with Promela alone. However, it is possible to embed C code within a Promela specification. The primary reason for this is to provide support for programs already written in C with minimal translation into Promela (Holzmann, 2004), not for use in handwritten Promela specifications. However, in our case, the increased accuracy afforded by the use of mathematical functions available using C outweighed the increased complexity resulting from its use.

Another advantage of using embedded C code is that variables that are declared solely in the C code do not need to be considered as part of the state-space (although it is possible to include them as state variables if necessary) when generating the model. This is a significant advantage in terms of state-space tractability. For example, variables used for intermediate calculations that are not relevant (i.e., do not influence transitions) can be ignored.

We embed our C code using functions that can be called in the main Promela specification. Our C macro functions are declared in a separate file included from within the Promela specification. Each function can be stored as an individual file to be tested and debugged separately from the main model. Several of these functions are used in both of our models.

The use of embedded C code does have some drawbacks. Simulations are more cumbersome, and the generation of meaningful counterexamples is a more complicated process. Also, any C code variables that affect the value of Promela variables must be tracked during a verification. The Promela `c_track` primitive allows us to do this. Each `c_track` declaration refers to the memory location and size of a C variable to be tracked, as seen in Figure 12. The use of this primitive allows the associated variables to be tracked during verification, allowing the normal verification of properties. It is important to note that even if an embedded variable does not directly affect a Promela variable, it may affect it indirectly so will still need to be tracked.

To illustrate, we include one of our embedded C code functions, the (`MOVE_FORWARD`) function, which determines the new position of the robot from a given state:

```
/*Explicit Model: Using Functions (Macros)*/

c_decl { #include <math.h> }
#include "exMoInLines.txt"
#define OBMAX 2

/*Define a polar coordinate to be a distance and angle from origin (pole)*/
/*(Pole: centre of the environment. Polar axis: vertical line directed north.)*/
typedef polarCoord {int d; int a};

/*Setting the C_Track variables*/
/*C_track statements keep track of our C globals.*/
c_track"&x" "sizeof(double)"
/*...etc*/

/*Array of obstacles in the fixed environment*/
polarCoord arrObs[OBMAX];

/*Robot is initialized in the centre of the environment*/
int roboAng, enviDist, enviAng, omegaD, sig, prevSig =0;
byte doWrap, headOn = 0;

proctype robot() {
     do
     :: (doWrap==0) -> d_step{ SCAN_APPROACHING_OBS();
                                 RESPOND();
                                 MOVE_ROBOT();
                                 HEAD_ON();
                                };
     :: (doWrap==1) -> d_step{ WRAP() };
    od;
};

init {
     d_step{
    /* Set up the polar coordinates of the obstacles - fixed for model */
      arrObs[0].d = 45;    arrObs[0].a = 350;
      arrObs[1].d = 154;    arrObs[1].a = 83;
      };
     atomic{ run robot()};
};
```

Figure 12: Promela code for the Explicit model.

```
#define MOVE_FORWARD() {

/*Declare Locals*/

if (now.relDist > 30) {

long double oZ, nZ, fZ, lOrg, hOrg, lNew, hNew, lFin, hFin = 0;

int oFR, oFU, nFR, nFU = 0;



oZ = fmodl(enviA, (long double)90);

if ((enviA == 90) || (enviA ==270))

{lOrg = enviD; hOrg = 0; }
```

```
else if ((enviA == 0) || (enviA == 180))
{lOrg = 0; hOrg = enviD; }
else {
if ((enviA <=90) || ((enviA >=180)&&(enviA<=270))) {
lOrg = (sin(oZ*DEG))*enviD;
hOrg = (cos(oZ*DEG))*enviD;
} else {
hOrg = (sin(oZ*DEG))*enviD;
lOrg = (cos(oZ*DEG))*enviD;
}
}

nZ = fmod(roboA, 90.00);
if ((roboA == 90) || (roboA ==270))
{lNew = moveDist; hNew = 0; }
else if ((roboA == 0) || (roboA == 180))
{lNew = 0; hNew = moveDist; }
else {
if ((roboA<90) || ((roboA>180)&&(roboA<270))) {
lNew = (sin(nZ*DEG))*moveDist;
hNew = (cos(nZ*DEG))*moveDist;
} else {
hNew = (sin(nZ*DEG))*moveDist;
lNew = (cos(nZ*DEG))*moveDist;
}
}
```

```
if ((enviA<180)&&(roboA>180) || (enviA>180)&&(roboA<180))
                { lFin = fabs(lOrg - lNew);}
  else {  lFin = lOrg + lNew;}
if ( (((enviA<90)||(enviA>270))&&((roboA>90)&&(roboA<270))) ||
(((enviA>90)&&(enviA<270))&&((roboA<90)||(roboA>270))) ) {
hFin = fabs(hOrg - hNew);
} else { hFin = hOrg + hNew; }
if ((hFin!=0)&&(lFin!=0))
{ fZ = (atan(hFin/lFin)*(180/PI)); }
else { fZ = 0;}
enviD = sqrt((lFin*lFin)+(hFin*hFin));

if ((enviA >=0) && (enviA <90))  { oFR = 1; oFU = 1;}
else if ((enviA >= 90) && (enviA <180))  { oFR = 1; oFU = 0;}
else if ((enviA >= 180) && (enviA <270)) { oFR = 0; oFU = 0;}
else if ((enviA >= 270) && (enviA <360)) { oFR = 0; oFU = 1;}
else { oFR = 0; oFU = 0;}
nFR = oFR;
nFU = oFU;
if ((oFR==1) && (oFU==1)) {
if ((roboA>=180) && (roboA<360) && (lNew > lOrg))
{ nFR = 0;}
if ((roboA>=90) && (roboA<270) && (hNew > hOrg))
{ nFU = 0;}
}
else if ((oFR==1) && (oFU==0)) {
if ((roboA>=180) && (roboA<360) && (lNew > lOrg))
{ nFR = 0;}
if ( (((roboA>=270)&&(roboA<360)) ||
```

```
((roboA>=0)&&(roboA<90))) && (hNew>hOrg))
                        { nFU=1;}
}
else if ((oFR==0) && (oFU==0)) {
if ((roboA>=0) && (roboA<180) && (lNew > lOrg))
{ nFR = 1;}
if ( (((roboA>=270)&&(roboA<360)) ||
((roboA>=0)&&(roboA<90)))&&
(hNew>hOrg))
                           { nFU=1;}
}
else if ((oFR==0) && (oFU==1)) {
if ((roboA>=0) && (roboA<180) && (lNew > lOrg))
{ nFR = 1;}
if ((roboA>=90) && (roboA<270) && (hNew > hOrg))
{ nFU = 0;}
}

/*Many catches for when movement is along/opposing axis line
or when both facing and polar angles are the same.*/
if (roboA==enviA) { enviA = roboA;}
else if ((roboA==180)&&(enviA==0)&&(hNew>hOrg))
{enviA = 180;}
else if ((roboA==180)&&(enviA==0)&&(hNew < hOrg))
{enviA = 0;}
else if ((roboA==180)&&(enviA==0)&&(hNew == hOrg))
{enviA = 0;}
else if ((roboA==0)&&(enviA==180)&&(hNew>hOrg))
{enviA = 0;}
```

```
else if ((roboA==0)&&(enviA==180)&&(hNew < hOrg))
{enviA = 180;}
else if ((roboA==0)&&(enviA==180)&&(hNew == hOrg))
{enviA = 0;}
else if ((roboA==90)&&(enviA==270)&&(lNew>lOrg))
{enviA = 90;}
else if ((roboA==90)&&(enviA==270)&&(lNew < lOrg))
{enviA = 270;}
else if ((roboA==90)&&(enviA==270)&&(lNew == lOrg))
{enviA = 0;}
else if ((roboA==270)&&(enviA==90)&&(lNew>lOrg))
{enviA = 270;}
else if ((roboA==270)&&(enviA==90)&&(lNew < lOrg))
{enviA = 90;}
else if ((roboA==270)&&(enviA==90)&&(lNew == lOrg))
{enviA = 0;}
else if ((nFR==1)&&(nFU==1)) { enviA = 90 - fZ;}
else if ((nFR==1)&&(nFU==0)) { enviA = 90 + fZ;}
else if ((nFR==0)&&(nFU==0)) { enviA = 270 - fZ;}
else if ((nFR==0)&&(nFU==1)) { enviA = 270 + fZ;}
if (enviA>=360) {now.enviAng = 0;}
else {now.enviAng = ((int)(2*enviA)) - ((int)enviA);}

enviD = ((int)(2*enviD)) - ((int)enviD);

if ((enviD>=200) && (now.doWrap==0))
 { enviD = 200; now.doWrap=1; }
now.enviDist = (int)enviD;
}
};
```

Table 4: Inline Functions.

Name	Purpose
SCAN_APPROACHING_OBS	Scans the area in front of the robot for obstacles. This area is restricted to distance and angle at which an obstacle may interact with the robot. Uses function GET_OB_REL_TO_ROBOT.
GET_OB_REL_TO_ROBOT	Calculates the center of an obstacle relative to the center of the robot.
RESPOND	Updates the signal from the robot's antennae, then calls the RESPOND_TO_OB_BY_TURNING function.
RESPOND_TO_OB_BY_TURNING	Turns the robot in response to the signals from its antennae. If the signal indicates proximal reaction, the LEARN function is called. If the obstacle is touching the robot, the CRASH function is called.
LEARN	Causes the robot to learn (i.e., increments ω_d.)
CRASH	Evaluates the movement of the robot after it has collided with an obstacle. If collision is head-on, the HEAD_ON function is called. Otherwise, a proximal turning response occurs.
HEAD_ON	Evaluates the movement of the robot after it has collided head-on with an obstacle. Eventually results in a proximal turning response.
MOVE_ROBOT	Moves the robot forward in the direction of its current orientation. Calls the MOVE_FORWARD function.
MOVE_FORWARD	Calculates the new position of the robot after moving forward. If the robot has reached the perimeter of the environment, sets a variable (doWrap) to 1.
WRAP	Wraps the position of the robot to the other side of the environment, using the point at which the robot approaches the perimeter of the environment and the orientation of the robot as it approaches.

Appendix B: Example Promela Code

B.1 Sample Code. Figure 12 shows a Promela specification in which the learning rate is 1 and there are two obstacles.

Note that `exMoInLines.txt` is an included file that contains a number of C-like macros and inline functions. An inline function in Promela is similar to a macro and is simply a segment of replacement text for a symbolic name (which may have parameters). The body of the inline function is pasted into the body of a proctype definition at each point that it is called. An inline function cannot return a value but may change the value of any variable referred to within the inline function. The purpose of each of the functions contained in `exMoInLines.txt` is described in Table 4.

Note that we do not include details of all of these inline functions here, although all of our code is available from the authors. Some of the functions

(e.g., MOVE_FORWARD and RESPOND_TO_OB_BY_TURNING) require mathematical calculations that are beyond the scope of Promela. Therefore, to perform these calculations, we use C code embedded within the Promela specification. To illustrate how this is achieved, we provide the definition of the MOVE_FORWARD below and the associated embedded C code in appendix A.

We return to the outline Promela specification given in Figure 12. After the inclusion of the inline function file, a constant OBMAX, indicating the number of obstacles, is declared. There follows a typedef definition, by which a type, namely, PolarCoord, consisting of two integers d (denoting distance from the origin) and a (denoting angular distance, clockwise from north). Some c_track (see appendix A) and global variables are then defined.

The robot and init proctypes are then declared. The robot proctype declaration contains a main do...od loop. The do...od loop contains two choices that are repeated indefinitely. At each invocation, variable doWrap is evaluated. This variable indicates whether the robot is close to the perimeter of the environment. If the variable has value 1, the robot will be relocated from its current position to a position at an equal distance from the perimeter on the other side of the environment. The new position is determined by the WRAP function (see Table 4) and is described in more detail in the text of the letter. Otherwise the SCAN_APPROACHING_OBS function checks to see if an obstacle has hit any of the sensors and updates the value of variable sig. If sig has a value of 0, then no antenna sensor has been hit. A negative signal indicates that the left antenna has been hit, and a positive signal indicates that the right antenna has been hit. If the value is -6 or 6, a proximal sensor has been hit. If neither of the antennae has been hit (and there has been no direct hit), the robot will simply move forward in its current direction. If an antenna has been hit or there has been a head-on collision, the robot will respond accordingly before moving forward in the new direction. In all cases, if the robot has reached the perimeter of the environment, the doWrap variable is set to 1. The do . . . od loop is then repeated.

The init process contains the initialization of the arrObs array of obstacles, and the initiation of the robot process.

Learning occurs during the RESPOND_TO_OB_BY_TURNING function. For learning to occur, there needs to be a temporal overlap between the proximal and distal antenna signals. We test for this overlap using the prevSig variable. The test works by checking if whenever there is a proximal signal (sig= ± 6), there was previously a distal signal ($0 < |\text{prevSig}| < 6$). If so, the learning weight ω_d is incremented by the learning rate λ, which is 1 in this model.

The WRAP function is defined in the main text, and illustrated in Figure 8.

B.2 The Move_Forward Function. Full code for the MOVE_FORWARD function is given in appendix A. The MOVE_FORWARD function calculates the new position of the robot after one time step, given the current direction of movement of the robot (relative to a ray pointing due North from the center

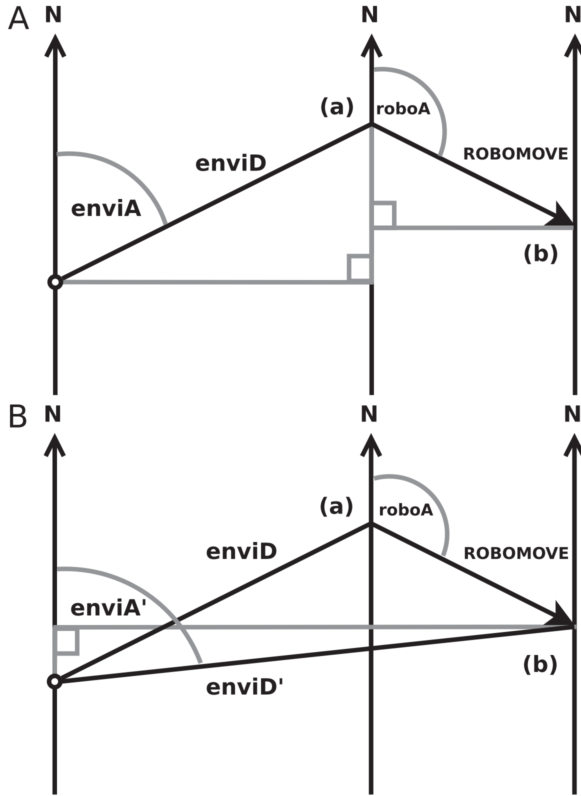

Figure 13: (A) Movement of the robot after one time step. (B) New position relative to origin.

of the robot), `roboA`, and the robot's current position *a* (coordinates `enviD` and `enviA`). The new position of the robot is *b*. The coordinates of *b* relative to *a* are `ROBOMOVE` and `roboA`, where `ROBOMOVE` is a constant, set to 1 in this example. Figure 13A illustrates this situation.

The coordinates of *b* relative to the origin are then calculated. These are represented by `enviD`′ and `enviA`′ in Figure 13B. The coordinates of the robot are updated to these values.

Appendix C: Abstract Specification in Promela

We give outline code for the abstract specification in Figure 14. As before, we include a file containing inline functions and C macros.

```
/*Abstract  Model: Using Functions (Macros)*/

c_decl { #include <math.h> }
#include "absInlines.txt"

int obDist = 90;
int obAng = 11;
int omegaD = 0;
byte freeSpace = 1;
byte pLearn = 0;

active proctype moving()
{
    do
    :: ((obDist > 30) && (freeSpace == 0)) ->
                              d_step{RESPOND_TO_OB_BY_TURNING();};
                              d_step{MOVE_FORWARD();};
                              d_step{LEARN();};
    :: ((obDist < 30) || (freeSpace == 1)) ->
                              atomic{GENERATE_NEW_OB();};
    od;
}
```

Figure 14: Promela Code for the Abstract Model.

References

Bordini, R., Fisher, M., Visser, W., & Wooldridge, M. (2006). Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, *12*(2), 239–256.

Braitenberg, V. (1984). *Vehicles: Experiments in synthetic psychology*. Cambridge, MA: Bradford.

Büchi, J. (1960). On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Science* (pp. 1–12). Stanford, CA: Stanford University Press.

Cattel, T. (1994). Modeling and verification of a multiprocessor realtime OS kernel. In *Proceedings of the 7th WG6.1 international conference on formal description techniques (FORTE '94)* (pp. 55–70). London: Chapman and Hall.

Chesi, G. (2009). Performance limitation analysis in visual servo systems: Bounding the location error introduced by image points matching. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2009 (pp. 695–700). Piscataway, NJ: IEEE.

Cimatti, A., Giunchiglia, F., Mingardi, G., Romano, D., Torielli, F., & Traverso, P. (1997). Model checking safety critical software with SPIN: An application to a railway interlocking system. In *Proceedings of the 3rd SPIN workshop* (pp. 5–17). Twente University, Enschede, Netherlands.

Clarke, E., & Emerson, E. (1981). Synthesis of synchronization skeletons for branching time temporal logic. In *Proc. of the 1st Workshop in Logic of Programs* (pp. 52–71). New York: Springer.

Clarke, E., Emerson, E., & Sistla, A. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, *8*(2), 244–263.

Clarke, E., Grumberg, O., & Peled, D. (1999). *Model checking*. Cambridge, MA: MIT Press.

Da Silva, V., & De Lucena, C. (2004). From a conceptual framework for agents and objects to a multi-agent system modeling language. *Autonomous Agents and Multi-Agent Systems*, *9*(1–2), 145–189.

Dekhtyar, M., Dikovsky, A., & Valiev, M. (2003). On feasible cases of checking multi-agent systems behavior. *Theoretical Computer Science*, *303*(1), 63–81.

D'Inverno, M., Luck, M., Georgeff, M., Kinny, D., & Wooldridge, M. (2004). The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, *9*(1–2), 5–53.

Dunin-Kęplicz, B., & Verbrugge, R. (Eds). (2004). Fundamenta informaticae (special issue on formal aspects of multi-agent systems). *Fundamenta Informaticae*, *63*(2–3).

Dunin-Kęplicz, B., & Verbrugge, R. (Eds.). (2009). Autonomous agents and multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, *19*(1).

Dwyer, M., Avrunin, G., & Corbett, J. (1998). Property specification patterns for finite-state verification. In *Proceedings of the Second International Workshop on Formal Methods in Software Practice* (pp. 7–15). New York: ACM Press.

Fainekos, G. E., Girard, A., Kress-Gazit, H., & Pappas, G. J. (2009). Temporal logic motion planning for dynamic robots. *Automatica*, *45*(2), 343–352.

Fisher, M. (2005). Temporal development methods for agent-based systems. *Autonomous Agents and Multi-Agent Systems*, *10*(1), 41–66.

Grana, C. Q. (2007). Selecting the optimal resolution and conversion frequency for A/D and D/A. In *Proceedings of the Instrumentation and Measurement Technology Conference* (pp. 1–6). Piscataway, NJ: IEEE.

Hilaire, V., Koukam, A., Gruer, P., & Müller, J.-P. (2000). Formal specification and prototyping of multi-agent systems. In *Proceedings of the 1st International Workshop on Engineering Societies in the Agent World* (pp. 114–127). New York: Springer.

Hilaire, V., Simonin, O., Koukam, A., & Ferber, J. (2004). A formal approach to design and reuse of agent and multiagent models. In *Proceedings of the 5th International Workshop on Agent-Oriented Software Engineering* (pp. 142–157). New York: Springer.

Hinton, A., Kwiatkowska, M., Norman, G., & Parker, D. (2006). Prism: A tool for automatic verification of probabilistic systems. 3920/2006 (pp. 441–444). New York: Springer.

Holzmann, G. (2004). *The Spin model checker: Primer and reference manual*. Upper Saddle River, NJ: Addison-Wesley Pearson Education.

Ingrand, F., & Py, F. (2002). An execution control system for autonomous robots. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Vol. 2, pp. 1333–1338). Piscataway, NJ: IEEE.

Kirwan, R., & Miller, A. (2011). Abstraction for model checking robot behaviour. In *Proceedings of the 18th Workshop on Automated Reasoning* (pp. 1–2). Glasgow, UK.

Konur, S., Dixon, C., & Fisher, M. (2012). Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems*, *60*(2), 199–213.

Kripke, S. (1963). Semantical considerations on modal logics. *Acta Philosophica Fennica*, *16*, 83–94.

Kulvicius, T., Kolodziejski, C., Tamosiunaite, T., Porr, B., & Wörgötter, F. (2010). Behavioral analysis of differential Hebbian learning in closed-loop systems. *Biological Cybernetics*, *103*(4), 255–271.

Kumar, S., & Li, K. (2002). Using model checking to debug device firmware. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*. San Mateo, CA: IEEE Computer Society.

Larsen, K. G., Patterson, P., & Yi, W. (1997). Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, *1*(1–2), 134–152.

Lerda, F., Kapinski, J., Maka, H., Clarke, E. M., & Krogh, B. H. (2008). Model checking in-the-loop: Finding counterexamples by systematic simulation. In *Proceedings of the American Control Conference*, *2008* (pp. 2734–2740).

Leucker, M. (2007). Learning meets verification. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* (pp. 127–151). New York: Springer.

Lichtenstein, O., & Pnueli, A. (1985). Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages* (pp. 97–107). New York: ACM Press.

Mao, H., Chen, Y., Jaeger, M., Nielsen, T., Larsen, K., & Nielsen, B. (2011). Learning probabilistic automata for model checking. In *Proceedings of the 8th International Conference on Quantitative and Qualitative Evaluation of Systems* (pp. 111–120). San Mateo, CA: IEEE Computer Society.

Matlab (2010). *Version 7.10.0 (r2010a)*. Natick, MA: MathWorks.

Miller, A., Calder, M., & Donaldson, A. F. (2007). A template-based approach for the generation of abstractable and reducible models of featured networks. *Computer Networks*, *51*(2), 439–455.

Miller, K. D. (1996). Synaptic economics: Competition and cooperation in correlation-based synaptic plasticity. *Neuron*, *17*, 371–374.

Molnar, L., & Veres, S. M. (2009). System verification of autonomous underwater vehicles by model checking. In *Proceedings of Oceans 2009—Europe* (pp. 1–10). Piscataway, NJ: IEEE.

Oja, E. (1982). A simplified neuron model as a principal component analyzer. *J. Math. Biol.*, *15*(3), 267–273.

Pnueli, A. (1981). The temporal semantics of concurrent programs. *Theor. Comp. Sci.*, *13*, 45–60.

Porr, B., & Wörgötter, F. (2006). Strongly improved stability and faster convergence of temporal sequence learning by utilising input correlations only. *Neural Computation*, *18*(6), 1380–1412.

Power, C., & Miller, A. (2008). Prism2promela. In *Proceedings of the 5th International IEEE Conference on Qualitative Evaluation of Systems* (pp. 79–80). Piscataway, NJ: IEEE.

Sharma, O., Lewis, J., Miller, A., Dearle, A., Balasubramaniam, D., Morrison, R., et al. (2009). Towards verifying correctness of wireless sensor network applications using insense and spin. In *Proceedings of the 16th International SPIN Workshop* (pp. 223–240). New York: Springer.

Sutton, R., & Barto, A. (1987). A temporal-difference model of classical conditioning. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (pp. 355–378). Hillsdale, NJ: Erlbaum.

Tronosco, J., Sanches, J. R. A., & Lopez, F. P. (2007). Discretization of ISO-learning and ICO-learning to be included into reactive neural networks for a robotics simulator. In *Nature Inspired Problem-Solving Methods in Knowledge Engineering* (pp. 367–378). New York: Springer.

Vardi, M., & Wolper, P. (1986). An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science* (pp. 332–344). San Mateo, CA: IEEE Computer Society Press.

Vardi, M., & Wolper, P. (1994). Reasoning about infinite computations. *Information and Computation*, *115*, 1–37.

Verschure, P., & Pfeifer, R. (1992). Categorization, representations, and the dynamics of system-environment interaction: A case study in autonomous systems. In *Proceedings of the Second International Conference on Simulation of Adaptive Behaviour* (pp. 210–217). Cambridge, MA: MIT Press.

Verschure, P., & Voegtlin, T. (1998). A bottom-up approach towards the acquisition, retention, and expression of sequential representations: Distributed adaptive control III. *Neural Networks*, *11*, 1531–1549.

Visser, W., Havelund, K., Brat, G., & Park, S. (2000). Model checking programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering* (pp. 3–12). San Mateo, CA: IEEE Computer Society Press.

Walter, W. (1953). *The living brain*. London: G. Duckworth.

Webster, M., Fisher, M., Cameron, N., & Jump, M. (2011). Formal methods and the certification of autonomous unmanned aircraft systems. In *Proc. 30th International Conference on Computer Safety, Reliability and Security* (pp. 228–242). New York: Springer.

Weissman, M., Bedenk, S., Buckl, C., & Knoll, A. (2011). Model checking industrial robot systems. In *Proceedings of the 18th International SPIN Workshop* (pp. 161–176). New York: Springer.

Wolper, P., Vardi, M., & Sistla, A. (1983). Reasoning about infinite computation paths. In *Proceedings of the 4th IEEE Symposium on Foundations of Computer Science* (pp. 185–194). San Mateo, CA: IEEE Computer Society.

Wooldridge, M., Jennings, N., & Kinny, D. (2004). The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, *3*(3), 285–312.

Yuen, C., & Tjioe, W. (2001). Modeling and verifying a price model for congestion control in computer networks using Promela/Spin. In *Proceedings of the 8th International SPIN Workshop* (pp. 272–287). New York: Springer.

Received September 4, 2012; accepted April 6, 2013.