Under consideration for publication in Formal Aspects of Computing

Model Checking Learning Agent Systems using Promela with Embedded C code and Abstraction

Ryan Kirwan[1], Alice Miller[1] and Bernd Porr[2]

[1]School of Computing Science, University of Glasgow, Glasgow, UK
[2]School of Electronic Engineering, University of Glasgow, Glasgow, UK

Abstract. As autonomous systems become more prevalent, methods for their verification will become more widely used. Model checking is a formal verification technique that can help ensure the safety of autonomous systems, but in most cases it cannot be applied by novices, or in its straight "off-the-shelf" form. In order to be more widely applicable it is crucial that more sophisticated techniques are used, and have been presented in a way that is reproducible by engineers and verifiers alike. In this paper we demonstrate two non-standard techniques that are used to increase the power of model checking using the model checker SPIN. The first of these is the use of embedded C code within Promela specifications, in order to accurately reflect robot movement. The second is to use abstraction to allow us to verify multiple environments simultaneously. We apply these techniques to a fairly simple system in which a robot moves about a fixed environment and learns to avoid obstacles. The learning algorithm is inspired by the way that insects learn to avoid obstacles in response to pain signals received from their antennae. Crucially, we prove that our abstraction is sound for our example system – a step that is often omitted but is vital if formal verification is to be widely accepted as a useful and meaningful approach.

Keywords: Model Checking, Abstraction, Agent, Simulation, Learning

1. Introduction

Robots are being used more and more in order to automate everyday mundane tasks from housecleaning and parcel delivery to production-line assembly. They are also being used in situations where humans simply can not venture – due to distance (aerospace systems), safety (in accident situations) etc. – and are increasingly required to operate autonomously i.e. they must make their own decisions based on the state of their environment. In order to achieve public acceptance autonomous systems must be shown to behave in a sensible way.

Correspondence and offprint requests to: Alice Miller, School of Computing Science, University of Glasgow, Glasgow, UK. e-mail: alice.miller@glasgow.ac.uk

Formal verification allows us to go some way towards providing a guarantee of safety of such systems. However, in order for it to be applied successfully to complex autonomous systems, the usefulness of formal verification techniques needs to be demonstrated for simple, single agent systems, with clear examples of how more advanced techniques can be applied.

In this paper we focus specifically on a type of system in which robots emulate primitive beetles. These robots use a dual sensor system to navigate environments. They have a pair of antennas, each of which has two sensors: one at a short distance from the robot that generates an inherent pain signal on collision with an object (proximal sensor); and one further from the robot which the robot uses to learn to use over time (distal sensor). A pain stimulus prompts the use of the distal sensors to predict a collision (and change course in order to avoid a further pain signal). Note that the two sensors on each antenna imitate two pairs of antennas (like those of beetle-like insects). The proximal sensors represent a short pair of antennas, and the distal sensors a long pair. In previous work [KKT+10] simulation has been used in order to determine whether, for a given learning algorithm and parameter set, a robot would be able to successfully navigate a variety of environments (without crashing) by learning to respond more (or less) vigorously to signals from their sensors. In this paper, we show in detail how we have used enhanced *model checking* techniques to provide a more formal and rigorous assessment of these systems than is possible using simulation and physical testing alone.

Model checking [CGP99, Mer00, MOSS99] is a useful technique for verifying properties of concurrent systems. It involves constructing a mathematical representation of a system (a *model*) which is then used to define all possible behaviours of the system, under given initial conditions. The model is constructed using a specification language, and checked using an automated model checker for satisfaction of a set of desired (temporal logic) properties. Failure of the *model* to satisfy a property of the system indicates that the model does not accurately reflect the behaviour of the system, that there is an error in the original system, or that the property has been incorrectly stated (i.e. it did not correctly define the desired behaviour). Examination of counterexamples provided by the model checker enable the user to refine the model or property, or, more importantly, to debug the original system. The benefit of model checking over simulation is that *all* behaviours (for given initial conditions) can be checked at once, rather than via a lengthy (and incomplete) series of test runs. The disadvantage is that construction of the model in the first place is a technical process, requiring a high degree of expertise. The modeller needs to decide what level of abstraction to use: or in other words, which information about the system can be ignored? Once a model is created, the generated behaviours (i.e. the *state-space*) may still be prohibitively large, requiring an even greater level of ingenuity to reduce the state-space to a tractable size. This may involve a further layer of abstraction to create a model that is in some way equivalent to the original.

Although model checking allows us to verify all behaviours of a system, with a given initial configuration, a new verification of the model must be run for every initial configuration. This could of course be avoided by introducing a new *dummy* initial state and non-deterministically selecting the *next* state corresponding to the original set of initial configurations. However, this would very quickly result in a blow-up in the size of the state-space. In addition, as we explain later in this paper, we would be restricted to proving properties that hold for *all* initial configurations (or none). In this paper we describe a method of abstraction, in which we replace the representation of an explicit environment for a smaller dynamic one, whereby multiple sets of initial conditions (specifically, different obstacle configurations) are merged into one model. This abstraction reduces the number of variables in the model, and consequently allows us to produce a model whose state-space is small enough to verify. Not only is our model now tractable, many different scenarios are represented by the same model. The system and our models have been described in previous work [KMPD13]. In this paper we focus on two aspects that are not previously published and are of particular relevance to the automated verification community. Specifically, we describe the Promela models, focusing on the use of embedded C code to allow for realistic representation of robot movement, and we describe our abstraction process in detail, providing a full proof that our abstraction is sound. Our proof relies on showing that there is a *simulation relation* [CGP99] between two models. The first model is specified at a level that is close to simulation code, and contains an agent whose behaviour represents a single robot moving within a specific environment. The second model is more abstract and not only represents a robot moving within any of a given class of environment, but merges equivalent views into a single view, so reducing the size of the exploration space. For simplicity we restrict our explanation to include a single robot. We explain how our approach can be extended to a more complex scenario in Section 6, and in more detail in [KMPD13].

Note that although the use of embedded C code in Promela is fairly common, there are no clear, detailed published examples of its use. We provide all of our code, *LTL* property declarations, and instructions for

compilation and verification on our website [KMP]. Our use of abstraction is particular to our example, and our presentation is unusual in that we justify our abstraction via a soundness proof.

2. Background

2.1. Validation of Robot Systems

Even simple robot implementations can exhibit complex behaviours [Bra84] because the actions of the robot influence its sensor inputs which in turn change the behaviour of the robot. Such a behaviour loop is generally non-linear and disturbed by noise. This makes it difficult to benchmark robot behaviour. In order to test if a robot is *successful* (in some way) one would usually conduct numerous simulation runs with different boundary or starting conditions and then perform statistical analysis on the results to calculate the probability at which the robot will fail. However, using this approach will not provide certainty that the robot will not fail in *any* situation. For example, a robot might get stuck in a corner and oscillate there forever. This situation may be rare and not be exposed by classical simulation. In a mission critical context it is essential that even the most unlikely scenario is explored. In our system the primary concern is to establish whether an unstable, simple learning algorithm can always produce a *successful* behaviour, in all circumstances. Our approach aims to demonstrate whether simple, learning algorithms can achieve seemingly complex avoidance behaviour–originally observed in biological systems–, or to identify possible failures of this learning.

Providing predictions about the failure or success of an robot (agent)[1] becomes even more difficult when the robot is able to learn from its experience [PvFW03]. In our case the robot learns to avoid obstacles by utilising its sensors and then turns away before it collides with them. However, this avoidance behaviour might render a perfectly functioning robot into one that is error-prone, for example because of an ill-adjusted learning weight.

In this paper we show how model checking, combined with abstraction, is used to verify properties of a robot system. While classical simulation provides at best a probability of failure or success, model checking can provide definite answers. For example, if a robot can get stuck in a corner, model checking will detect it. Model checking is therefore a powerful tool for analysing mission critical robot scenarios, providing assurance that a system is fit for purpose.

2.2. The System

In this paper we describe two particular aspects concerning the model checking of a system consisting of a robot learning to avoid obstacles within a fixed environment. Details of the physical system are given in [KMPD13]. In this paper we provide only a high level description that is sufficient to justify our modelling decisions. Any assumptions or simplifications that we make are those made in previous work involving simulation [KKT$^+$10].

A learning system consists of: a *robot* (agent), a set of *obstacles* (objects that are impassible by the robot) and an *environment* (an open area that can contain robots and obstacles). Note that for our explicit models the environment is wrapped around on itself, such that when an agent moves off the edge of the environment it will return from the side opposite to its exit trajectory.

A robot senses its surroundings via sensors and responds by turning and driving motors. It has two sensor pairs, one pair of proximal sensors and one of distal sensors. The right antenna contains the right proximal and distal sensors (and similarly for the left antenna). The robot uses learning to attempt to avoid collisions. This involves responding to the signals from its distal sensors by changing its trajectory.

An environment has a defined *density*. In this paper this density refers to the reciprocal of the minimum distance between obstacles. We define this minimum distance to be such that no two obstacles can be in contact with the robot at a given time. This assumption simplifies our model, but could be relaxed if necessary.

The robot learns by using a form of temporal sequence learning ([SB87, PW06]) called *input correlation*

[1] In the context or our software model we refer to an "agent", whereas the term "robot" is used for the physical system.

learning (ICO learning). Learning takes place when the robot collides with an obstacle after receiving a signal from its distal sensors. When the collision occurs the robot receives a *pain* signal from its proximal sensors which is correlated to an earlier signal from the distal sensors. In this case the robot learns to turn more vigorously to distal signals. Specifically, this is achieved by using the difference between the signals received from its left and right pairs of sensors, which can be interpreted as error signals [Bra84]. At any time an error signal x is generated from these signals, and the steering angle adjusted accordingly.

2.3. Robot Movement

We first make a distinction between the actual movement of the physical robot, and the movement as represented in our models (by an *agent*), which is based on the representation used in simulation. The robot's movements are dictated by simply responding to environmental stimuli, the simulator uses equations to emulate these responses. It is important to realise that the fairly complex calculations that are used in the simulation code and Promela models are not performed by the robot. As simulation alone is not sufficient to check all possible behaviour, we build a Promela model based on the simulator code, so that we can use model checking to verify behaviour. Rather than our model exactly following the simulator approach, we have modified things slightly so as to model the system from the robot's perspective (with obstacles moving towards it). This not only allows us to reduce the size of the state space, but is a useful approach when we come to developing a further abstraction in which multiple environments are considered within a single model. This approach has also been (independently) used in the model checking of swarm navigation algorithms [AICE15], where it was shown to reduce the size of the state space considerably.

In this section we describe the observed movement of the robot and its environment. We explain how this movement is simulated using equations and projected onto a fixed, central position for use in our Promela models, in Section 3.

The robot moves in the environment by responding to the presence (or not) of obstacles in its surrounding area. A fixed rate of learning, i.e. the *learning rate* λ, is assumed. At any time there is a *learning weight* ω_d associated with the robot. If an obstacle is detected on its distal sensors the robot will turn an angle that is determined by the current value of ω_d. If an obstacle is detected on its proximal sensors then the robot will turn a predetermined angle ($90°$) to avoid it. If an obstacle is detected on a distal sensor and then subsequently on the paired proximal sensor then ω_d is increased (by an amount proportional to the learning rate). The result is an increased angle of turning after all subsequent distal sensor obstacle detections. If no obstacle is detected, the robot will continue in the current angle of projection.

At any moment, there is a relatively small set of points at which an obstacle can interact with the robot. These points are contained in a conic area in front of the robot. We call this area the *cone of influence*. The size of the cone of influence can be calculated from the specification of the robot and of the environment (including the nature of obstacles, within its environment). We assume the maximum distance between any two points in the cone is less than the minimum distance between any two obstacles in an environment. For our specification there can only be one obstacle in the cone at a given time. The widest point of the cone of influence is calculated by adding the distance between the tips of the robot's antennas to the radius of two obstacles (154 units in our case). In addition, the edge of the cone must be at least one obstacle's width from the robot's antennas. Figure 1 illustrates the cone of influence when the robot is in what we call the default position (i.e. at the pole, travelling due North); it also shows the measurements in relative units –where $1unit$ is the movement rate of the robot.

In Figure 2 we show how the robot responds to an obstacle in its cone of influence. Again, in this example, we assume that the robot is in the default position. On detecting an obstacle on the left antenna the robot first turns to the right (by an angle determined by the current learning weight ω_d) and then moves forward in the new direction. In order to simulate the robot movement, calculation of the new position is fairly straightforward, using standard geometry. This is described in Section 3.

2.4. Spin

Temporal logic model checking [CGP99] involves creating a mathematical representation (*model*) of a system from which all behaviours can be generated. These behaviours take the form of a graph (or *state-space*) in which nodes represent states and edges transitions between states.

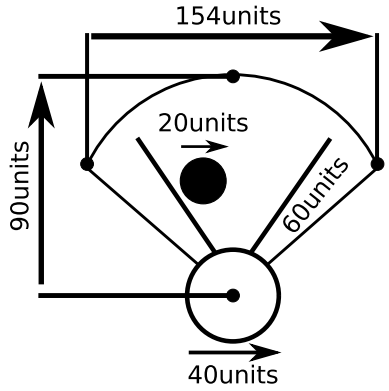

Fig. 1. Cone of Influence.

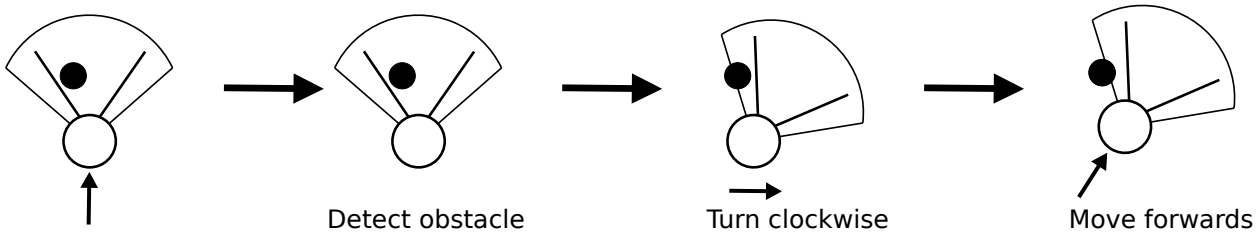

Fig. 2. Response to distal signal.

The SPIN model checker creates models from specifications expressed in the Promela specification language, and has been used to trace logical errors in distributed systems designs, such as operating systems [Cat94, KL02], computer networks [YT01], railway signalling systems [CGM+97], wireless sensor network communication protocols [SLM+09] and industrial robot systems [WBBK11]. SPIN can be used to check for assertion violations, deadlock and satisfaction of Linear Time temporal Logic (*LTL*) formulas (see Section 2.5).

Promela

Promela is an imperative style specification language designed for the description of network protocols. 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 parameterised proctypes. A special process, the `init` process, can also be declared. This process will contain any array initialisations, for example, as well as run statements to initiate process instantiation. If no such initialisations are required, and processes are not parameterised, the `init` process can be omitted and processes declared to be immediately active, via the `active` keyword.

The Promela `do...od` construct provides a way of expressing a loop in which commands are repeatedly selected non-deterministically until a `break` statement is executed. Choices are denoted `::statement`. In addition, Promela allows us to group together statements that should be executed at the same time (i.e., before another component executes a transition) using an `atomic` statement or a `d_step` statement.

Promela specifications in which all statements for a given proctype are contained within a `do...od` loop in which the statement choices have the form: `::atomic{guard->command}` (or `::d_step{guard->command}`) are said to be in *Guarded Command Form* (GCF).

GCF is a common specification form and generally consists of one global loop over a choice of statements of the form `guard->command`. The precise definition of the form depends on the specification language. In some model checking tools (e.g., MurPhi [Dil96] and SMV [McM93]) models are specified directly in this form. GCF for Promela was described in [CM08] as a way of simplifying the connection between a Promela specification and its underlying model. We use GCF in all Promela specifications described in this paper for similar reasons.

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 a proctype definition at each point that it is called. An inline function can not return a value, but may change the value of any variable referred to within it.

Properties are either specified using `assert` statements embedded in the body of a proctype (to check for unexpected reception, for example), an additional *monitor* process (to check global invariance properties), or via *LTL* properties (*LTL* will be described in more detail in Section 2.5). *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.

As an example, consider the property $\langle\rangle([]\texttt{omegaD} == 0)$ (along every path, eventually the variable `omegaD` is always 0). This can be verified by defining proposition `a` to be $(\texttt{omegaD} == 0)$ and including the associated never-claim as follows:

```
#define a (omegaD == 0)
never {  /* !(<> ( []  a)) */
T0_init:
if
:: (! ((a))) -> goto accept_S9
:: (1) -> goto T0_init
fi;
accept_S9:
if
:: (1) -> goto T0_init
fi;
}
```

An error path (in which `a` does not eventually become continuously true) will be detected by Spin as a repeating sequence in which the state labelled with the `accept` prefix is visited. This is known as an *acceptance cycle*.

Alternatively, if `a` is defined within the code then the *LTL* property can be translated into a Promela never claim via the command line option `-f` as follows:

```
spin -f '<> ( []  a)'
```

Since Spin version 6 an even simpler method for declaring *LTL* properties has been available, namely the inclusion of an *LTL* formula inline within the Promela specification. For example, the property above can now simply be declared thus:

```
#define a (omegaD == 0)
ltl property1 { <>[] a }
```

All of our code, including *LTL* property declarations, and instructions for compilation and verification are available from our website [KMP].

Embedded C code

A useful feature of Spin is that it allows for the use of C code embedded within a Promela specification as C code macros. This functionality has been available since Spin version 4, released in 2003. The use of embedded C code is described in the Spin reference manual [Hol04], but illustrations of its use in real-world applications are rare. The primary reason for the use of C code is to provide support for programs already written in C with minimal translation into Promela, not for use in hand-written Promela specifications. However, in our case, the increased accuracy afforded by the use of mathematical functions available using C code outweighs the increased complexity resulting from its use. For the learning agent systems we model, the precision at which we represent the movement of the robot is important – we are interested in whether a robot has collided with an obstacle or avoided it. The distinction between a near miss and contact is important.

Embedded C code allows one to reduce a model's state-space by changing variables from Promela into

```
c_decl {
       typedef struct polarCoord {
               int d,a;
       } polarCoord;
}
```

(a) Example use of `c_decl`

```
c_state ‘‘polarCoord obPos’’    ‘‘Global’’
c_state ‘‘polarCoord agentPos’’ ‘‘Hidden’’
c_state ‘‘polarCoord antenPos’’ ‘‘Local proc1’’
                               ‘‘now.agentPos’’
```

(b) Example uses of `c_state`

```
do
:: c_code {position1.d++; now.agentPos.d++;}
od;
```

(c) Example use of `c_code`

```
do
:: c_expr {position1.d == position2.d}
          c code {now.agentCrash = true;}
od;
```

(d) Example use of `c_expr`

```
c_track ‘‘&moveDist’’ ‘‘sizeof(int)’’
```

(e) Example use of `c_track`

Fig. 3. C code constructs

C code, where the variables no longer create new states (although it is possible to include them as state variables if necessary). For example, variables used for intermediate calculations that are not relevant (i.e., do not affect the truth, or otherwise, of any property being verified) can be stored and manipulated in C code, but not stored as part of the state-vector. This gives a significant advantage in terms of tractability. Note that the use of `c_track` declarations (see below) allow us to control which variables in the C code are visible to the model checker, so that we do not ignore crucial variables.

Note that although using embedded C code allows complex calculations, variables in Promela that are stored in the state vector can not take real values and so must be approximated. This does not result in loss of accuracy during collision detection — we are only concerned as to whether the robot has crashed, representing the exact position of a collision is unnecessary.

Using embedded C code does pose some drawbacks in addition to its increased expressiveness. First, it requires a much greater understanding of the SPIN model checker, particularly the syntax for the inclusion of C code. Second, examining counterexamples from failed verifications is a more tedious process. Counterexamples can be analysed, but this involves additional compilation of one of SPIN's output files and the parsing of the resulting compiled file. However, it is possible to automate this analysis. In this section we describe the semantics of the embedded C code. We illustrate the use of embedded C code in Section 4.

The most common constructs used to embed C code in Promela specifications are:

`c_decl:` A `c_decl` primitive can appear only in the global declarations of a Promela specification. It allows for C code datatypes to be embedded into a model. Figure 3(a) shows how a `c_decl` primitive is inserted into a Promela program.

Here the C code `typedef` is used to create a `struct` that is a polar coordinate (containing a distance and

an angle). This coordinate can be used in calculations within the Promela or in other sections of embedded C code, and it can be referenced in *LTL* formulas when checking properties of the model.

`c_state:` A `c_state` primitive can appear only as a global declaration. It allows for C code variables to be defined in the Promela code that are part of the model's state-space. The `c_state` variables can be defined as *Global*, *Local*, or *Hidden*. Global variables have a common value for all processes; additionally, they are used when generating the state-space of the model. Hidden variables are declared globally in the Promela code although they are not used when generating the model's state-space. They can, however, be used for calculations in the Promela code. Local variables are local to a process, but will still be part of the state-space. Figure 3(b) shows three declarations of `c_state` variables.

There are three possible fields for each declaration, they are: variable type and name, visibility, and initial value. Note that the local variable (`antenPos`) has the name of the process it is local to after the keyword `Local`. Also note, `antenPos` is initiated to the same value as `agentPos`.

`c_code:` A `c_code` primitive can appear anywhere within a Promela specification. Once reached, the C code within the `c_code` primitive is executed unconditionally and automatically. Use of this primitive is illustrated in Figure 3(c).

Here the C code is simply incrementing two integers in `structs`. The "`now.`" prefix indicates that it is referring to the internal state vector for the model. (The internal state vector is used to generate the state-space for the model.)

`c_expr:` A `c_expr` primitive is equivalent to the `c_code` primitive except that it is not executed unconditionally. It is only executed if it returns a non-zero value when its test is evaluated. Figure 3(d) shows an example conditional statement that could be used in a `c_expr`. If the positions are equal (a non-zero value is returned), the `c_code` primitive is executed.

`c_track:` Any C code variables that directly affect the value of Promela variables must be tracked during a verification. The `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. The use of this primitive allows the associated variables to be tracked during the verification of the model, while allowing for the normal verification of properties. It is important to note that even if an embedded C code variable does not directly affect a Promela variable, it may affect it indirectly so will still need to be tracked. A `c_track` declaration is shown in Figure 3(e).

The C code variable `moveDist` is tracked using the `c_track` primitive. Here '`&`' denotes the memory location, while "`sizeof{int}`" indicates that the variable is the size of an integer.

2.5. The Formal Model

In order to be able to reason about our models, we provide formal semantics. We define a Kripke structure [Kri63] 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 [Büc60]). However, to prove that our Abstract model preserves *LTL* properties (in Section 5.2) we will reason about the underlying Kripke structures of our Promela programs.

Definition 2.1. 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, $s_0 \in S$ is the initial state, $R \subseteq S \times S$ is a transition relation and $L : S \to 2^{AP}$ 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, $0 \leq i$, $(s_i, s_{i+1}) \in R$.

Linear Time Temporal Logic

To specify properties of Kripke Structures we employ *LTL*. An *LTL* formula is either a *state* formula or a *path* formula. A state formula is either *true*, an atomic proposition or a boolean combination of state formulas. A path formula is either a state formula or has the form $\neg\phi_1$, $\phi_1 \wedge \phi_2$ or $\phi_1 U \phi_2$, where ϕ_1 and ϕ_2 are path formulas and U is the standard (strong) until operator. We also use the common abbreviations $[]$ and $\langle\rangle$, to denote *for all states* on a path, and *for some state* on a path (i.e. *eventually*) respectively.

For model $\mathcal{M}$=(S, s_0, R), if state formula ϕ holds at a state $s \in S$ then we write $s \models \phi$. For a path π with initial state s_0 and path formula ϕ, $\pi \models \phi$ implies that ϕ is true at s_0. We write $\mathcal{M} \models \phi$ when ϕ holds for every path. For example, if p is a proposition, $\mathcal{M} \models []p$ if p holds for all states on every path, and $\mathcal{M} \models \langle\rangle p$ if p is true for at least one state on every path.

In Section 1 we considered the necessity of verifying a new model per initial configuration. This could theoretically be avoided by introducing a new *dummy* initial state and non-deterministically selecting the *next* state corresponding to the original set of initial configurations. Apart from the potential blow-up in the size of the state-space, since *LTL* only allows us to check properties that hold for *every* path, we would be restricted to proving properties that hold for every initial configuration. Our abstraction approach (see Section 5) involves clustering initial configurations (according to original obstacle placement). This reduces the number of verification runs required but in a more controlled way.

ϕ-Simulation relation

A simulation relation (H) [Mil71] is a relation between the states of two Kripke structures, $\mathcal{M}$ and $\mathcal{M}'$ that preserves *LTL* properties. If we have a model $\mathcal{M}$ that is too large for us to verify properties for, if it is possible to create a model $\mathcal{M}'$ that simulates $\mathcal{M}$, we can verify properties for $\mathcal{M}'$, and thus *infer* that they hold for $\mathcal{M}$.

For brevity we use a simplified form of simulation relation that is property specific. Our relation is called a ϕ-simulation relation.

Let AP_ϕ denote the set of atomic propositions in property ϕ, and for any state s, let $L_\phi(s)$ denote the label of s with respect to AP_ϕ (i.e. the set of propositions from AP_ϕ true at s). The following definition is adapted from [CGP99]:

Definition 2.2. Given two structures $\mathcal{M}$= (S, s_0, R, L) and $\mathcal{M}'$= (S', s'_0, R', L') whose sets of propositions contain AP_ϕ, a relation $H_\phi \subseteq (S \times S')$ is a ϕ-simulation relation between $\mathcal{M}$ and $\mathcal{M}'$ if and only if for all $(s, s') \in H_\phi$

1. $L_\phi(s) = L_\phi(s')$
2. For every state $s_1 \in S$ such that $R(s, s_1)$ there is a state $s'_1 \in S'$ with the property that $R'(s', s'_1)$, and $H_\phi(s_1, s'_1)$.

We say that $\mathcal{M}'$ *ϕ-simulates* $\mathcal{M}$ (denoted by $\mathcal{M} \preceq_\phi \mathcal{M}'$) if there exists a ϕ-simulation relation H_ϕ such that $H_\phi(s_0, s'_0)$.

The following theorem is also adapted from one in [CGP99]:

Theorem 2.1. Suppose $\mathcal{M} \preceq_\phi \mathcal{M}'$. Then $\mathcal{M}' \models \phi$ implies $\mathcal{M} \models \phi$.

We illustrate the concept of ϕ-simulation via Figure 4. Suppose that ϕ is the property $[]p$, and that the labels of all of the states contain p. Note that we are not interested in the other propositions true at each state when verifying ϕ, and we can assume that the states matched in our simulation below have different labels in general, although they all contain p. The simulation relation here is:

$$H_\phi = \{(s_0, s'_0), (s_1, s'_1), (s_2, s'_1), (s_3, s'_1), (s_4, s'_1), (s_5, s'_1),$$
$$(s_6, s'_2), (s_7, s'_3), (s_8, s'_3), (s_9, s'_4), (s_{10}, s'_5)\}$$

In the following, we refer to the "paths in P_i" ("paths in P'_i") as the set of paths consisting of s_0 (s'_0) and the set of subpaths contained in the box labelled Pi (Pi') in Figure 4. Note that the paths in P_1 and P_2 in $\mathcal{M}$ are mapped to the paths in P'_1 in $\mathcal{M}'$, and the paths in P_3 are mapped to the paths in P'_2, as indicated by the arrows in the diagram.

3. Simulated Robot Movement

In this section we describe how the observed movement of the robot is simulated using mathematical equations, which will be incorporated into our Promela models in Sections 4 and 5. We use a similar approach to that used in existing simulation code, although the translation of the new position of the robot back to the default position is our own modification. Note that the original simulation has been described in previous work [KKT+10] and we make similar assumptions to those contained therein. Our models use an *agent* to

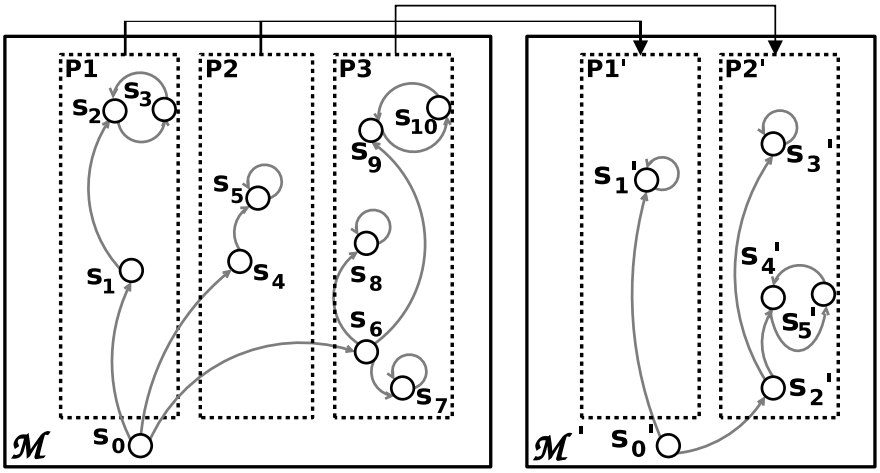

Fig. 4. ϕ Simulation Relation. All states satisfy p and $\phi = []p$.

simulate the movement of the robot. Whereas the robot simply responds to to environmental stimuli, equations are used to reposition the agent in a way that emulates these responses. Although our agent is more of an abstract concept (i.e. doesn't physically exist, move in an environment, or have antennas, for example), in this section we sometimes refer to it as if it were a physical object. For example, when we say that a signal is received on the agent's sensor, we simply mean that the allocated position of an obstacle coincides with the theoretical position of the sensor (given the current position and direction of movement of the agent).

We represent the environment as a polar grid; where the centre of the environment is the pole, and angles are measured clockwise from a ray projected North from the pole (this represents the polar axis). The agent, environment, and each obstacle have their centre points stored as polar coordinates. This representation allows for precise angles to be stored, which is important because the robot turns an exact angle to avoid obstacles. We define an agent whose transitions reflect the movement of the robot (and are determined mathematically, according to the current position, angle, position of obstacles, and current learning rate). In Section 4 we show how we use C code embedded in our Promela specification to determine the new position of the agent at every step. Note that precise floating point values are rounded to determine whether a collision has occurred (for example) but their floating point values are retained for future calculations.

We assume that there is one agent in an open, circular environment and that the radius of the environment is 200 units. Like the robot, the agent learns to use its distal sensors to avoid colliding into obstacles. This is done by comparing the sensor signals from the two antennas. If an obstacle is in a position where it would touch a sensor, then a signal is said to have been received from that antenna with a value between 1 and 6, as illustrated in Figure 5. The overall signal received (`sig`) is the difference between the signal received from the left and right antennas, and so $-6 \leq \texttt{sig} \leq 6$. Since an obstacle can not touch both antennas simultaneously, $\texttt{sig} \in \{-6, 6\}$ indicates a proximal signal, $0 < |\text{sig}| < 6$ a distal signal and $\texttt{sig} = 0$ no signal.

The circular environment makes our polar representation less complex for calculating the edges of the environment. However, the use of a square, or another shape, for the environment is within the scope of our approach.

If the boundary has been reached the agent is relocated to a position at the perimeter on the opposite side from its orientation. The mathematical function required to do this relocation is discussed in Section 4.1.

Figure 6 shows an example of an agent in an explicit environment. Given an environmental density, it is possible to place the agent and obstacles in a range of positions and angles. Each such placement is called a *configuration*, and defines a set of initial conditions.

In our models, we are concerned with the position of the obstacles *relative* to the agent. The relative positions can be found by assuming that, rather than the agent moving, obstacles move towards (or away) from the agent. So the situation illustrated in Figure 2 becomes that of Figure 7.

Note that antennas are assumed to behave as perfect springs. This explains why, in Figure 7, the antenna appears to pass through the obstacle. In fact the antenna springs back until the agent has turned and the

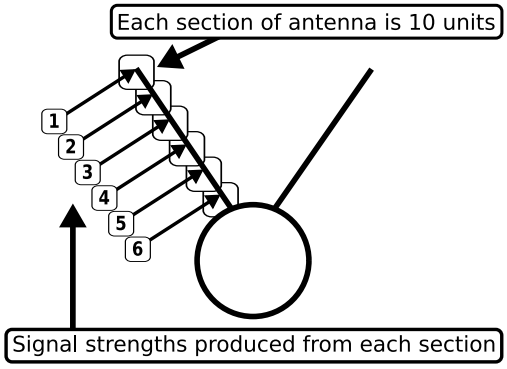

Fig. 5. Antenna signal strengths.

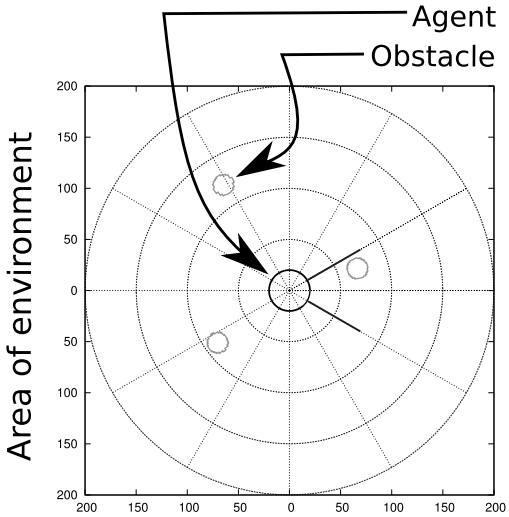

Fig. 6. Example of an agent in an environment in the Explicit model.

antenna is released from the obstacle. This emulates insect antennae retracting upon contact. The antennas can only sense one obstacle on a pair of proximal and distal sensors at a time. If there is more than one obstacle along the line of an antenna, then the closest obstacle is used to produce the signal to the agent. In fact, we assume that the distance between obstacles is such that this situation does not occur. An obstacle is anything in the environment other than free space. The agent starts to move in the centre of an environment, and there are no obstacles to inhibit it from doing so.

When an agent is not in the default position its new position is determined by mapping it to one that is. The relative position of the agent to any obstacle is then calculated and the agent's response is determined. The cone of influence is then mapped back to the original position and the agent moved accordingly. This translation not only simplifies our calculations, but aids our proof of abstraction (see Section 5.2).

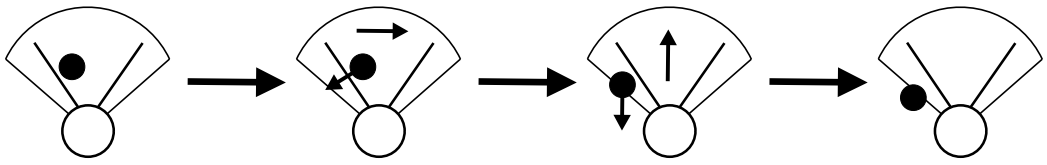

Fig. 7. Translated response to distal signal.

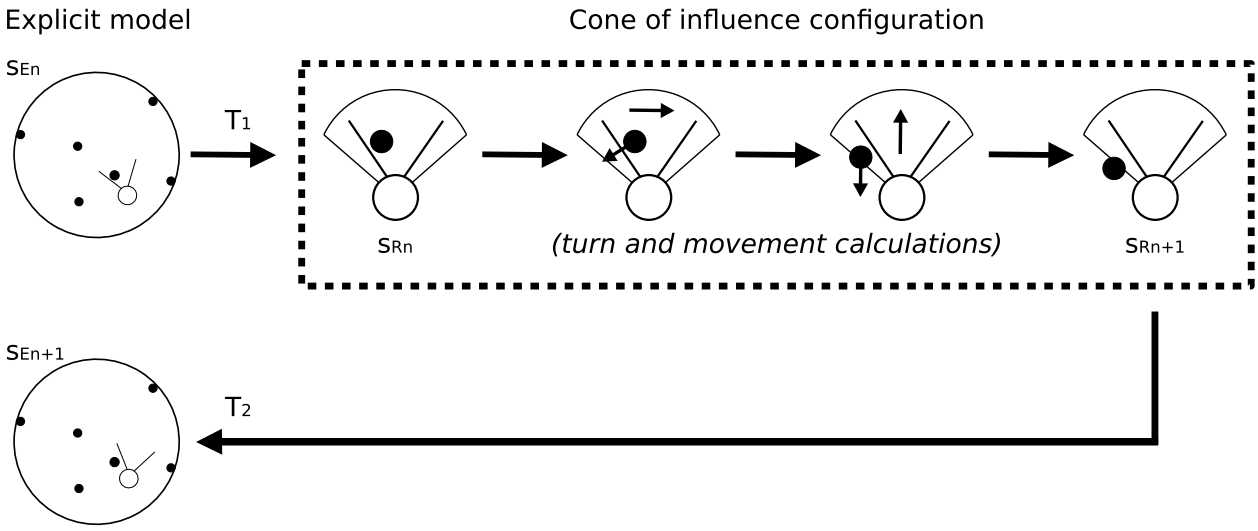

Fig. 8. Translation of agent to and from the default position.

In Figure 8 we illustrate how this translation takes place. Two functions, T_1 and T_2 are used to map the agent in an original position into the default position, and then back again once the movement of the agent has been calculated.

Note that function T_1 involves isolating a cone of influence around the agent and then mapping the agent and that cone to the default position. Any obstacle is then moved towards the agent and then function T_2 used to return the agent to its new position in the grid, relative to the fixed position of the obstacle. In all cases, in order to return the agent to the correct place, a record must be kept of the original coordinates of the agent relative to the centre of the environment. This record is called the *key*, and is denoted K_n.

Functions T_1 and T_2, and calculation of agent movement are described fully in [Kir14]. We give a flavour of these calculations by describing function T_1 in Section 3.1.

3.1. Translation function T_1

The translation function T_1 maps a state in our Explicit model s_{En} to an equivalent state s_{Rn} in the cone of influence representation. To calculate this translation, we take the original coordinates of the agent and isolate the cone of influence around it. Once in this representation the translation is complete. Note that all of the variable names introduced in this section are given, with explanations, in Table 1. Let the coordinates of the agent in s_{En} be (eA, eD), and the coordinates in s_{Rn} be $(0, 0)$. If there is no obstacle within the cone of influence then there is no more to be done. In the rest of this section we assume that there is an obstacle in the cone of influence, with coordinates (oA, oD).

We must now calculate the relative position ($relA, relD$) of the obstacle with respect to the agent. To do this we need to calculate (i) the distances of the agent from the Euclidean axes (*lOrg* and *hOrg*), (ii) the distances of the obstacle from the Euclidean axes (*lNew* and *hNew*), (iii) the horizontal and vertical distances between the agent and the obstacle, and the angle between them (*lFin*, *hFin* and *fZ*), (iv) the relative position of the agent to the obstacle (*fR* and *fU*), and finally (v) *relA* and *relD*. These calculations are done with the help of the diagram shown in Figure 9, containing triangles $\triangle_1$, $\triangle_2$ and $\triangle_3$. Note that rA denotes the direction of movement of the agent relative to North.

Figure 9 shows the triangles which represent the Euclidean distances of the agent and the obstacle relative to the centre of the environment. The coordinates of the agent and the obstacle are used to generate them.

Triangles $\triangle_1$ and $\triangle_2$ are made between the centre of the environment and the position of the agent or obstacle, respectively. Triangle $\triangle_3$ is calculated from triangles $\triangle_1$ and $\triangle_2$. It is used to calculate the agent's cone of influence, and the position of the obstacle inside it –relative to the centre of the agent. The Euclidean axes run in line with the polar axis (due North), and perpendicular to this, passing horizontally through the pole, respectively.

Variable Name (s)	Description
s_{En}	state in explicit model
s_{Rn}	equivalent state in cone of influence representation
(eA, eD)	coordinates of the agent in explicit model
(oA, oD)	coordinates of obstacle nearest to agent (in explicit model)
($relA, relD$)	coordinates of obstacle nearest to agent, with respect to agent
$lOrg$ and $hOrg$	distances of the agent from the Euclidean axes
$lNew$ and $hNew$	distances of the obstacle nearest to agent, from the Euclidean axes
$lFin$, $hFin$	horizontal and vertical distances between the agent and nearest obstacle
fZ	angle between agent and nearest obstacle
fR and fU	relative position of the agent to the nearest obstacle
rA	direction of movement of the agent relative to North
$\mathcal{O} \subseteq \mathcal{D} \times \mathcal{A}$	set of coordinates of all the obstacles in the environment
$\mathcal{Z} \subseteq \mathcal{D} \times \mathcal{A}$	set of coordinates that make up the agent's cone of influence

Table 1. Variable names used in translation function

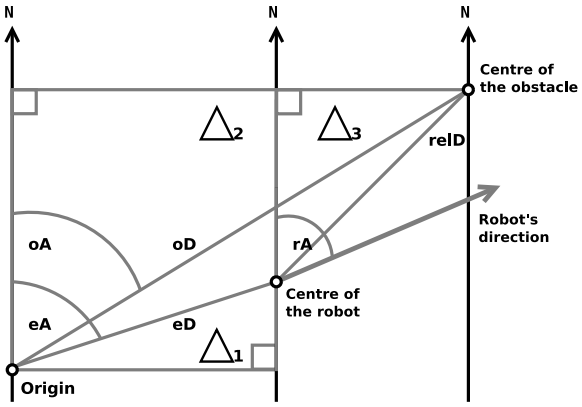

Fig. 9. Triangles representing the calculations to convert from the original position to the default position.

In order to simplify our calculations in the remainder of this section we introduce some notation to denote quarters of the set $\Theta = \{\theta^\circ : 0 \leq \theta \leq 360\}$. The quarters of Θ are $[Q_0]$, $[Q_1]$, $[Q_2]$ and $[Q_3]$, where $[Q_i] = \{\theta^\circ : 90i \leq \theta \leq 90(i+1)\}$. For $0 \leq i \leq 3$, $(Q_i]$, $[Q_i)$ and (Q_i), denote the corresponding sets which are open at one or both ends. For example $(Q_0] = \{\theta^\circ : 0 < \theta \leq 90\}$.

(i) Distances of the agent from the Euclidean axes

The distances from the centre of the agent to the vertical and horizontal Euclidean axis are represented in triangle $\triangle_1$ by lines of length $lOrg$ and $hOrg$ respectively. The line of length eD represents the distance from the origin to the centre of the agent. Values $lOrg$ and $hOrg$ are calculated thus:

$$oZ = eA\%90$$

$$(lOrg, hOrg) = \begin{cases} (eD, 0), & \text{if } eA = 90^\circ \text{ or } eA = 270^\circ \\ (0, eD), & \text{if } eA = 0^\circ \text{ or } eA = 180^\circ \\ ((\sin(oZ) * eD), \\ (\cos(oZ) * eD)), & \text{if } eA \in (Q_0) \cup (Q_2) \\ ((\cos(oZ) * eD), \\ (\sin(oZ) * eD)), & \text{otherwise} \end{cases}$$

(ii) Distances of the obstacle from the Euclidean axes

Distances ($lNew$ and $hNew$) are calculated in a similar way, using triangle $\triangle_2$.

$$nZ = oA\%90$$

$$(lNew, hNew) = \begin{cases} (oD, 0), & if\, oA = 90^\circ \text{ or } oA = 270^\circ \\ (0, oD), & if\, oA = 0^\circ \text{ or } oA = 180^\circ \\ ((\sin(nZ) * oD), \\ (\cos(nZ) * oD)), & if\ oA \in (Q_0) \cup (Q_2) \\ ((\cos(nZ) * oD), \\ (\sin(nZ) * oD)), & otherwise \end{cases}$$

(iii) Distances and angle between the agent and obstacle,

Using triangles $\triangle_1$ and $\triangle_2$, we construct triangle $\triangle_3$ whose sides have lengths *lFin* and *hFin*, the horizontal and vertical distances between the agent and the nearest obstacle. The angle between the agent and the obstacle is the angle between these two lines and is denoted fZ. Values of *lFin*, *hFin* and fZ are calculated thus:

$$lFin = \begin{cases} lOrg + lNew, & if\ eA \in [Q_0] \cup [Q_1) \text{ and } oA \in (Q_2] \cup [Q_3) \text{ or} \\ & \quad eA \in (Q_2] \cup [Q_3) \text{ and } oA \in [Q_0] \cup [Q_1) \\ |lOrg - lNew|, & otherwise \end{cases}$$

$$hFin = \begin{cases} hOrg + hNew, & if\ eA \in [Q_0) \cup (Q_3) \text{ and } oA \in (Q_1] \cup [Q_2) \text{ or} \\ & \quad eA \in (Q_1] \cup [Q_2) \text{ and } oA \in [Q_0) \cup (Q_3) \\ |hOrg - hNew|, & otherwise \end{cases}$$

$$fZ = \arctan(hFin/lFin)$$

(iv) Relative position of the agent to the obstacle

Next we calculate the relative position of the agent to the obstacle, horizontally (fR) and vertically (fU). Calculating whether the agent is further up and/or further right of the obstacle is necessary for calculating the angle of the obstacle relative to the agent ($relA$). The values of fU and fR are calculated thus:

$$fR = \begin{cases} 1, & if\ eA \in [Q_0] \cup [Q_1] \text{ and } oA \in [Q_0] \cup [Q_1] \text{ and } lOrg > lNew \text{ or} \\ & \quad eA \in [Q_2] \cup [Q_3) \text{ and } oA \in [Q_2] \cup [Q_3) \text{ and } lOrg < lNew \text{ or} \\ & \quad eA \in [Q_0] \cup [Q_1) \text{ and } oA \in (Q_2] \cup [Q_3) \\ 0, & otherwise \end{cases}$$

$$fU = \begin{cases} 1, & if\ eA \in [Q_0] \cup [Q_3) \text{ and } oA \in [Q_0] \cup [Q_3) \text{ and } hOrg > hNew \text{ or} \\ & \quad eA \in [Q_1] \cup [Q_2] \text{ and } oA \in [Q_1] \cup [Q_2] \text{ and } hOrg < hNew \text{ or} \\ & \quad eA \in [Q_0] \cup [Q_3) \text{ and } oA \in [Q_1] \cup [Q_2] \\ 0, & otherwise \end{cases}$$

(v) Relative position of the obstacle with respect to the agent

We can now calculate the relative angle and the relative distance from the agent to the obstacle ($relA$ and $relD$). The angle $relA$ is measured from the farthest anticlockwise point in the agent's cone of influence (40° anticlockwise from the direction that the agent is facing). The distance $relD$ is the distance from the centre of the agent to the centre of the obstacle. The values of $relA$ and $relD$ are calculated thus:

$$\begin{aligned} relA &= ((90 + (180 * fR) + (2 * fZ * |fR - fU|) - fZ) - rA + 400)\%360 \\ relD &= \sqrt{(hFin^2 + lFin^2)} \end{aligned}$$

4. The Explicit Representation

In this section we describe our Promela specification and explain how we calculate the movement of the agent. Note that there are two models described in this paper. The first (in this section) describes a more explicit representation, i.e. close to simulation code. We refer to this model as the Explicit model. Our other model (see Section 5) is an abstraction of the Explicit model and we refer to it as the Abstract model. Note the use of capital letters to infer a particular model (as opposed to a general explicit or abstract model). So as to avoid confusion, whenever we refer to the current learning weight ω_d or fixed learning rate λ of the agent, we do so via the variables/constants used to represent them (`omegaD` and `LAMBDA`) in our Promela models. We will continue to do this in the remainder of the paper.

4.1. The Promela Specification of the Explicit model

Our initial Promela specification (i.e. the Explicit specification) is based on the original code used for simulation [KKT+10]. All model checking requires some degree of abstraction, but due to closeness of the Explicit specification to the original simulation code, we assume this specification (and its derivable model) to be *concrete*, i.e. we do not question that the specification leads to a realistic model of the underlying system. A different model is required per learning rate and environment.

An example Promela specification for the Explicit model with a given environment is shown in Figure 10. Note that all of our code (and instructions for its use) is available from our website [KMP].

We employ embedded C code to implement the movement of the agent, i.e. to calculate the precise new location of the agent given its current position and direction of movement, and the position of any obstacle touching its antennas. For example, embedded C code is used to implement the calculations required for translation function T_1, as described in Section 3.1. Global variables representing the learning rate and the number of obstacles are declared, and an array representing the environment is initialised within the `init` process. A file containing a number of C-like macros and *inline functions* is included, namely `newExMo.h`, and a text file containing further inline statements, namely `newExMoInLines.txt` is also included. One of the constants declared in the `newExMo.h` file is `LAMBDA`, which denotes the learning rate. The use of embedded C code. requires the setup of some `c_track` (see Section 2.4) variables, which are used to monitor the relevant values in the calculations –including them as part of the state-space. Note that `c_track` variables do not need to be considered as part of the state-space as they can be declared with the parameter "`Unchecked`" which stops them contributing to the state-space. However, they will still be stored on the search stack during verification.

Each Explicit model contains a set of obstacles stored in an array (this is named `arrObs[OBMAX]` in Figure 10). The number of obstacles in the array varies depending on the size and density of the model. The constant `MAX_OMEGAD` is used for verification purposes, and is explained in Section 4.2.

The main body of the code is in the process `agent()`, which defines the behaviour of the agent. This behaviour is determined via a loop (a `do...od` loop, see Section 2.4). Every movement of the agent corresponds to an execution of the loop. If the boundary has not been reached, a sequence of inline functions are used to calculate the new position of the agent and respond to any sensor signal. If the boundary has been reached the new position of the agent is calculated via the `WRAP` function. This causes the agent to be relocated from its current position to a position at the perimeter on the opposite side from its orientation, and is discussed in detail in Section 4.1.

The inline function `SCAN_APPROACHING_OBS` returns a value indicating the position of any obstacle touching the sensors and `RESPOND` updates the direction of movement of the agent and implements learning if appropriate. There is a special inline function to deal with the situation when an agent collides with an obstacle head on, namely the `HEAD_ON` function. This type of collision is unique because it means that the agent has crashed into an obstacle without contacting any of its antennas. In this case the agent continues to try to drive forward, and this results in it slipping to one side of the obstacle, which causes contact with

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

/*C library included*/
c_decl { #include <math.h> }

/*Promela includes/definitions*/
#include "newExMo.h"
#include "newExMoInLines.txt"
#define OBMAX 2
#define MAX_OMEGAD 2

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

/*Setting C_Track vars (C vars tracked in verification)*/
c_track "&x"     "sizeof(double)"
c_track "&prevX" "sizeof(double)"
/*...etc.(more go here) */

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

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

proctype agent() {
      do
      ::    (doWrap==0) ->

      d_step {
      /*Call these functions in 1 transition of the model*/
            SCAN_APPROACHING_OBS();   /*Is there an obstacle?*/
            RESPOND();    /*Agent responds to antenna signals*/
            MOVE_AGENT();       /*Drive (maybe after turning)*/
            HEAD_ON();       /*Check for un-sensed collision */
      };
      :: (doWrap==1) ->d_step{ WRAP() };
      od;
};
init {
      d_step {
      /* Setup polar-coords of obstacles-fixed for model */
            arrObs[0].d=45; arrObs[0].a = 350;
            arrObs[1].d=154; arrObs[1].a = 83;
            atomic{ run agent()
      };
};
/*Define Variables for Property*/
#define p ((sig<=-6)||(sig>=6))
#define d ((prevSig!=0) && (prevSig>-6) && (prevSig<6) )
/*Properties Verified:
 *ltl phi1_prime { <>[](p->!d)}
 *ltl phi2 { [] (omegaD<=MAX_OMEGAD}*/
```

Fig. 10. Example Promela specification for the Explicit model

one of its proximal sensors. Because it is largely random as to which side the agent slips to, we select which agent's antenna is contacted non-deterministically. Note that there is a small additional assumption here: that the agent slides to one side of the obstacle. This is a result of the surfaces of the obstacle and the agent being rounded and the fluctuations in the wheels' driving forces (this behaviour can be observed in the physical system). A description of the inline functions (including those called from other functions) is given in Table 2.

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 sensor signals. We test for this overlap using variables `sig` and `prevSig`. The test works by checking if a proximal signal ($|\texttt{sig}| = \pm 6$) corresponds to a previous

Name	Purpose
SCAN_APPROACHING_OBS	scans the area in front of the agent for obstacles. This area is restricted to distances and angles at which an obstacle may interact with the agent. Uses function GET_OB_REL_TO_AGENT.
GET_OB_REL_TO_AGENT	calculates the centre of an obstacle relative to the centre of the agent.
RESPOND	updates the signal from the agent's antennas then calls the RESPOND_TO_OB_BY_TURNING function.
RESPOND_TO_OB_BY_TURNING	turns the agent in response to the signals from its antennas. If the signals indicate a proximal reaction then the LEARN function is called. If the obstacle is touching the agent then the CRASH function is called.
LEARN	causes the agent to learn; i.e., increments omegaD using the defined value of LAMBDA.
CRASH	evaluates the movement of the agent after it has collided with an obstacle. If collision is head-on then the HEAD_ON function is called. Otherwise a proximal turning response occurs.
HEAD_ON	evaluates the movement of the agent after it has collided head-on with an obstacle. Eventually results in a proximal turning response.
MOVE_AGENT	moves the agent forward in the direction of its current orientation. Calls the MOVE_FORWARD function.
MOVE_FORWARD	calculates the new position of the agent after moving forward. If the agent has reached the perimeter of the environment, sets a variable (doWrap) to 1.
WRAP	wraps the position of the agent to the other side of the environment, using the point at which the agent approaches the perimeter of the environment and the orientation of the agent as it approaches.

Table 2. Inline functions

distal signal ($0 < |\texttt{prevSig}| < 6$). If so, the agent's learning weight (omegaD) is incremented by its learning rate (LAMBDA) – which is 1 in this model. The new value of omegaD is then factored into the agent's future movement calculations.

The Promela specification is initiated within the init process. In the example specification shown in Figure 10 an environment is defined in the init process. Usually, we call an environment from a set of stored environments defined as inline functions within the file exMoInLines.txt.

The functions in the Promela specification are declared in the underlying C code where real variables are used in their calculations. It should be noted that these real variables are maintained and stored throughout verification of the model. In order for some of these values to generate different states in the model their values are rounded to whole numbers and stored separately as part of the state vector.

In order for models to be generated faster, we automate the calculations for a set of legal obstacles (i.e. to set the values in the arrObs array). This allows us to generate an environment for every model. This auto-generation code requires as input: an environmental density, a size of obstacle, and a size of environment. From these values, the auto-generation code produces an array of legal obstacles. Note, for these models we stipulate that there are no obstacles in the centre of the environment. The agent will always start at the centre of the environment. These restrictions are not essential, but they are sensible assumptions to make.

Two *LTL* properties are shown at the bottom of the example specification. We simply uncomment the property that we wish to verify (full instructions can be found on our website [KMP]).

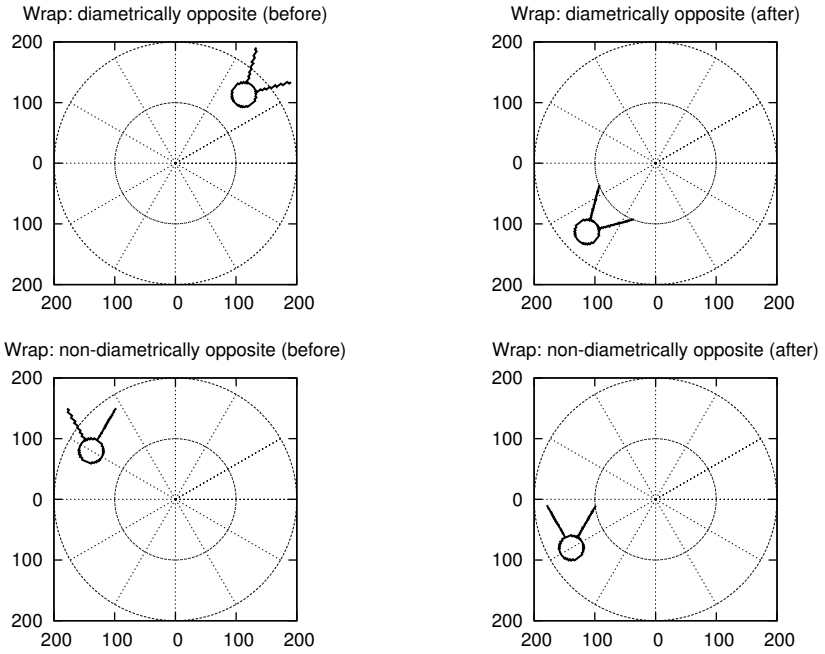

Fig. 11. Examples of the WRAP function

Note that our specification is in GCF, see Section 2.4.

The WRAP function

Note that our simulations and models are different to those described in [KKT+10] as there are no boundaries imposed on our environments. The removal of boundaries is a simplification that allows us to present a proof of concept analysis with a simpler system. Removing the boundaries also helps us in our abstraction (see Section 5). To accommodate the removal of the boundaries we allow the agent to wrap around the environment once it reaches the edge; such that it appears at the opposite side of the environment, in the direction it was facing. We include details of the WRAP function here, as it provides us with a small example with which to illustrate the use of embedded C code.

The WRAP function uses the angle at which the agent exits the environment to calculate the farthest point on the environment, opposite the exit point, at which to reposition the agent.

In Figure 11 we illustrate the wrap function in two scenarios. In the first, since the agent is able to drive continuously along the same trajectory, it wraps to the point at the edge of the environment diametrically opposite to the point at which it left. In the second, the agent emerges at the point vertically below its point of exit. Hence, the agent is never reorientated unless it interacts with an obstacle.

The Promela implementation of the WRAP function, incorporating embedded C code is given in Figure 12. The code simply tries to place the agent as far as possible from where it is on the other side of the environment. It does this using a `for` loop, checking each generated position until it finds one that is within the boundary of the environment.

4.2. Verification of the Explicit Model

We are concerned with whether the agents manage to learn to avoid obstacles. To assess this we define two properties using *LTL* formulas. The properties are as follows.

- ϕ_1: The signal produced from the proximal sensors will eventually stay zero, indicating that the agent is now only using its distal sensors.

```
#define WRAP() c_code {

/*Declare Locals*/
    int l = 0;
    long double orgEnviA = 0;
    long double orgEnviD = 0;
    int oppAng = 0;

    roboA = (double)now.roboAng;/*Save roboAng*/
    moveDist = ENVIDIAM;/*Set to Enviromnt diameter*/
    oppAng = ((int)(roboA+180))%360;
    roboA = oppAng; /*set for MOVE_FORWARD function*/
    orgEnviA = enviA;
    orgEnviD = enviD;

    /*Call MOVE_FORWARD function.*/
    MOVE_FORWARD();

    /*Reposition agent in nearest legal position.*/
    for (l=399; l--; l>=0) {

        if (enviD > 200) {
        /*Reset the environment variables.*/
            enviA = orgEnviA;
            enviD = orgEnviD;
            moveDist = l;

            MOVE_FORWARD();
        } else {break;} \
    }

    /*Reset the roboAng and doWrap flag*/
    roboA = (double)now.roboAng;
    now.doWrap = 0;
};
```

Fig. 12. WRAP function

- ϕ_2: The learning value eventually stabilises, indicating that the agent has finished learning –stopped crashing.

These properties are defined in terms of the variables in our Promela speficiation thus:

$\phi_1 : \langle\rangle[](!p)$
$\phi_2 : [](\texttt{omegaD} <= \texttt{MAX\_OMEGAD})$

In property ϕ_1, p is a proposition defining a proximal reaction. Specifically p is defined as ((sig>= 6)||(sig<= −6)) where sig is a variable denoting the signal difference between the agent's sensors and takes integer values in the range $[-6, 6]$. The extreme values indicate a proximal reaction.

In property ϕ_2, the constant MAX_OMEGAD represents the maximum level of learning possible for a particular model. The property allows us to show that eventually learning will cease (and the agent will avoid obstacles without the need to further increase its turning angle). Although any (finite) value of MAX_OMEGAD for which ϕ_2 were true would allow us to show this, we find the smallest appropriate value. We do this via a series of simpler verifications, in which an initially over-approximated value is decreased to the point at which the property is no longer true. We can not determine this value using a single verification, as SPIN only allows us to prove that a property holds for all paths, or fails for at least one.

It would be possible to include an additional state variable in the model that would remove the need for calculating MAX_OMEGAD. That is, C code variable pLearn could be tracked as a state variable. This variable is set to 1 whenever learning occurs, and to 0 after learning. We could define an alternate property $\langle\rangle[](\texttt{pLearn} == 0)$ stating that eventually pLearn remains equal to 0 (i.e., learning stabilises).

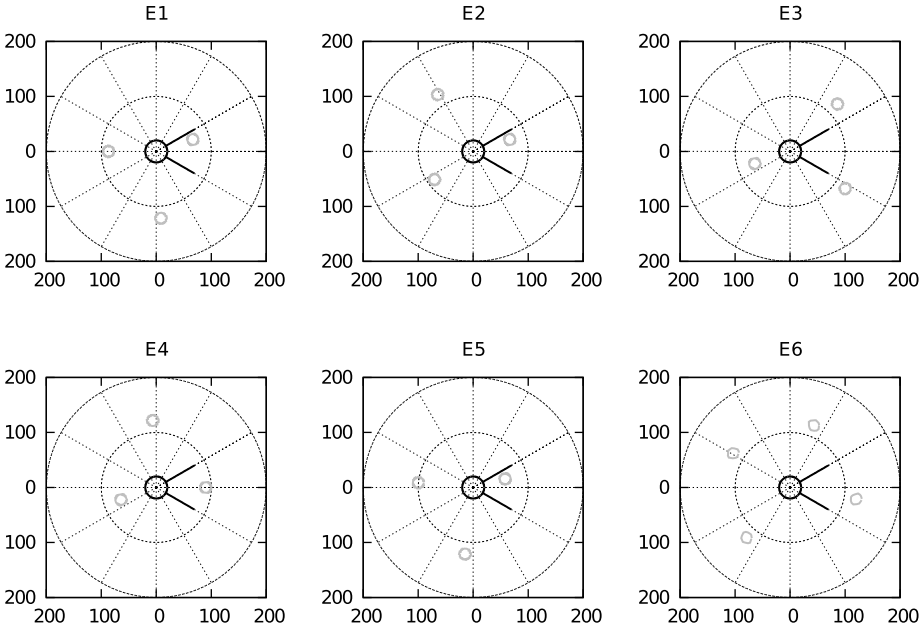

Fig. 13. Environments E1 - E6.

The drawback of tracking `pLearn` as a state variable is that it will increase the state-space of the model. For this reason we decided to not use this and to use property ϕ_2 instead.

Property ϕ_1 was checked and shown to be false for an example environment. Initially, it was suggested that this indicated that the agent was unable to learn to avoid colliding with obstacles. However, examination of a counter-example revealed an error due to the fact that there had been a head on collision, with the obstacle passing undetected through the antennae. This prompted us to define new property ϕ_1' that relates proximal reactions to corresponding distal reactions. The counter-example for property ϕ_1 also led to a discussion with the robot designers about the effectiveness of the parameter values in the physical system.

Property ϕ_1' is defined thus:

$$\phi_1' : \langle\rangle[](p- >!d)$$

Here p is the proposition defining a proximal reaction as before and d a proposition defining a corresponding distal reaction. Specifically d is defined as $((\texttt{prevSig} > -6)\&\&(\texttt{prevSig} < 6)\&\&(\texttt{prevSig!} = 0))$, where `prevSig` is a variable recording the previous value of the signal difference between the agent's sensors. This property is shown to be true for our set of example environments.

Properties ϕ_1' and ϕ_2 were verified on six automatically generated environments. Note that the properties are included in the Promela specification (see Figure 10) by uncommenting the appropriate line. Each model has the same environmental density (in that in all cases no two obstacles can be in contact with the agent at any time), but a different distribution of obstacles. The six environments used for the verifications are shown in Figure 13. These indicate the starting locations of the models. The agent is in the centre facing East, and the obstacles are shown as hollow, grey circles.

Verification results for the models associated with the environments shown in Figure 13 are presented in Table 3, where the column labelled *Env* denotes the environment used. The value of `MAX_OMEGAD` (required for property ϕ_2) is calculated separately for each environment. Note that in environments `E3` and `E4` the maximum learning weight achieved is zero. This is not because the agents don't interact with the obstacles, but because those interactions don't involve contact with their distal sensors. The prior contact with the distal sensors is the key component of ICO learning; therefore, without it no learning takes place.

In `E3` and `E4`, the agents have continuous collisions with an obstacle; they turn 90° on each impact, and then repeat the collision from a different angle. Hence, these models get locked into repetitive sequences of collisions and therefore exhibit no variance in behaviour (no learning). In `E6`, although there is an initial

Env	MAX_OMEGAD	Stored states	Max search depth	Time (sec)
E1	1	268974	537947	0.25
E2	1	149900	299799	0.14
E3	0	670	1339	0.00
E4	0	78602	157203	0.07
E5	1	216692	433383	0.21
E6	1	61100	122199	0.06

Table 3. Verification results for the Explicit model

collision (hence learning does occur in this case) the agent quickly settles into a cyclic path. As a consequence, the state-spaces for these models are lower than the others. This is an interesting observation for an individual environment; though, as the number of obstacles is increased then the likelihood of this decreases. In fact, for this type of environment, this scenario is not useful if learning is to be assessed.

Table 3 also shows the total states stored when verifying the properties. As discussed above, In E3 and E4, the number of *stored states* is low relative to the other models because of the lack of learning and distal reactions in these models and in E6 the number of stored states is again low because the agent very quickly adopts a regular, cyclic pattern of movement after learning. The *Max search depth* is the length of the longest path explored when verifying the property. On the right of the table the time taken to run each verification is displayed in seconds. We verified our properties using SPIN version 6.4.3. Properties ϕ_1' and ϕ_2 were successfully verified for each model. In all cases, the number of stored states, the maximum search depth and the elapsed time were the same for both properties (this was not true when using older versions of SPIN).

5. The Abstract model

In this section we describe the concept behind our Abstract model and how it is derived from the Explicit model.

Model checking allows us to check all behaviours of a system *with a given set of initial conditions*. In our example this means that for every individual environment (i.e. placement of the obstacles), and every learning rate, we need a different Promela specification and verification. In this section we describe how we create an abstract model which allows us to represent a class of environments in a single model (the *class* being all environments for which the minimum distance between obstacles is large enough to ensure that no two objects appear within the cone of influence at any time). A new model is still required per learning rate, as the response of the agent to an obstacle depends on this rate.

Our abstraction relies on the fact that, at any state, atomic propositions occurring in properties of interest (in our case, properties ϕ_1' and ϕ_2) refer only to the agent's current position, direction of movement, and the position of any obstacle (if indeed there is one) within its cone of influence. Hence, calculating the agent's next position is independent of the coordinates and orientation of the agent relative to any explicit environment (i.e. it is the *relative* position of any obstacle that matters), or the number of obstacles elsewhere in that environment. For this reason, any state of the Abstract model can be represented by a state where the agent is in the default position, with any obstacle in the same relative position. This concept is illustrated in Figure 14.

In our Abstract model the agent is in the default position at every state. The dimensions of the cone of influence are shown in Figure 1 and only the cone of influence in the default position is modelled, not an entire environment with given radius and fixed obstacles, as in the Explicit model. In this case the movement of the agent is represented by the movement of any obstacle (i.e. relative to the centre of the agent). This is illustrated within the dashed box on the right hand side of Figure 8. Note that, as explained in Section 2.3, in the Explicit model all configurations are mapped to the default position, obstacles moved relative to the agent, and then a reverse mapping used to calculate the new position of the agent. The difference here is that there is no translation, all calculations are done in the default configuration, and – most importantly – non-determinism is used to allow us to model all possible environments (up to the given density) within

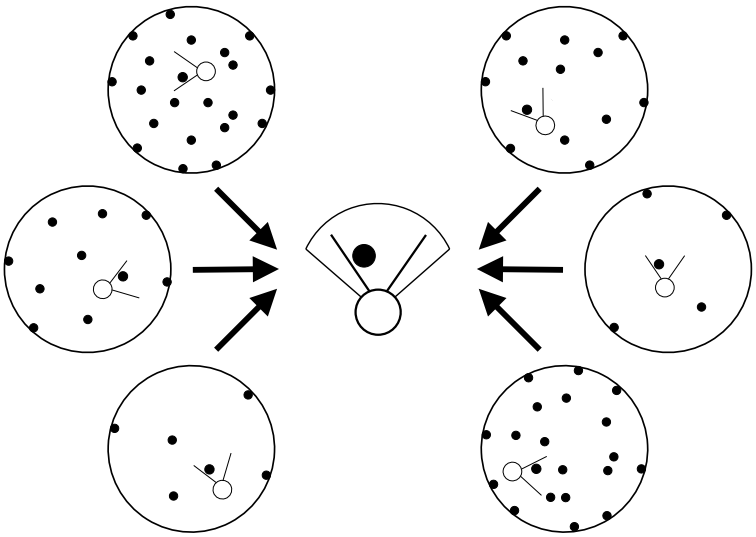

Fig. 14. Equivalent positions of the agent in various environments. Associated states are merged in the abstraction.

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

Fig. 15. Fragment of Promela specification for the Abstract model

a single model. There is a clear relationship between the Explicit and Abstract models. The difference is, however, that a range of scenarios for the Explicit model in a fixed environment are merged to a single situation in the Abstract model. Figure 14 illustrates a range of scenarios where the agent's situation in an Explicit model is equivalent to one scenario when using the default configuration in the Abstract model (centre of figure).

Our Abstract model incorporates a further level of abstraction by way of non-determinism. If there is no obstacle within the cone of influence at a given state s, the next state is chosen non-deterministically as either s (representing any other position at which there is no obstacle within the cone of influence), or any state s at which there is an obstacle at the boundary of the cone of influence. This further level of abstraction means that a single Abstract model not only condenses several views of a single environment into a single view, but abstracts all environments with a given density (i.e. all associated sets of initial conditions) into a single model.

5.1. Promela Specification of the Abstract model

A fragment of our Promela specification for the Abstract model containing the proctype declaration `moving`, representing the movement of an agent in the cone of influence, is shown in Figure 15. Note that all of our code is available from our website [KMP].

At each execution of the loop, the behaviour of the agent is determined by whether there is an obstacle within the cone of influence (`freeSpace==0`) and whether a collision has just occurred (`obDist<=30`). If there is no obstacle within the cone of influence then the `GENERATE_NEW_OB` inline causes the agent to either continue in free space, or for a new obstacle to appear at the boundary of the cone of influence in front of the agent. If a collision has just occurred, then the agent is assumed to have turned 90° to avoid it. In

our Abstract model this is represented by the agent remaining stationary and the obstacle moving out of the cone of influence. Since we assume that there can only be one obstacle in the cone at a given time (see Section 2.3), the agent either moves into free space, or a new obstacle appears somewhere on the boundary of the cone of influence in front of the agent. This is again achieved via the `GENERATE_NEW_OB` inline.

If the agent is not in free space, but has not collided with an obstacle, then it checks for signals on its sensors and turns if required. These actions are controlled by the `AB_RESPOND_TO_OB_BY_TURNING` inline. It then moves forward via the `AB_MOVE_FORWARD` inline. The impact of these actions will be a new position of the obstacle either inside or outside of the cone of influence. Finally, if the new sensor signals indicate that learning is appropriate, the learning weight (`omegaD`) is incremented by the learning rate (`LAMBDA`), via the `AB_LEARN` inline.

5.2. Proof that our abstraction is sound

Our Abstract model was constructed using intuition: in the Explicit model we translate the cone of influence associated with an agent in a given position to one in the default position to do our calculations, so modelling only the cone of influence in the default position seemed legitimate. Similarly, by allowing obstacles to appear in the cone of influence non-deterministically seemed to be a sensible way to cover all possible environments rather than modelling them individually. However, in the context of a formal verification technique it is important to *prove* formally that our abstraction is sound. To do this, we show that our Abstract model *simulates* our Explicit model, with respect to our properties ϕ_1' and ϕ_2.

We need to prove that satisfaction of an *LTL* property for the Abstract model implies its satisfaction for any Explicit model with an environment in the specified class. In this section we use the term *model* to denote the underlying Kripke structure (see Definition 2.1) associated with a Promela specification.

In order to prove that satisfaction of an *LTL* formula ϕ for the Abstract model $\mathcal{M}_R$ implies its satisfaction for any Explicit model $\mathcal{M}_E$ with an environment within the defined class, we must demonstrate that there is a ϕ-simulation relation, H_ϕ, between our models such that $\mathcal{M}_E \preceq_\phi \mathcal{M}_R$. Therefore, we need to determine that the relation H_ϕ (i.e. the set of pairs of states (s_{En}, s_{Rn})) satisfies the conditions of Definition 2.2.

Before we define our ϕ-simulation, we justify our approach with reference to Figure 16. In the diagram, functions T_1 and T_2 are the translation functions that map to and from the default position representation. These functions were first introduced in Section 2.3. In the following, $\mathcal{F}_E$ and $\mathcal{F}_R$ denote the transition relations associated with $\mathcal{M}_E$ and $\mathcal{M}_R$.

Consider transition $(s_{En},\ s_{En+1})$ in the left model ($\mathcal{M}_E$). State s_{En} is mapped to a state s_{Rn} in the right model ($\mathcal{M}_R$) using function T_1, and then there is a transition from s_{Rn} to s_{Rn+1} via $\mathcal{F}_R$. State s_{Rn+1} is determined from the position of an obstacle relative to the centre of the agent and from the agent's turn, movement, and learning. The relative position of the obstacle is used to calculate the new sensor difference signal, denoted in the code as `sig`. The learning weight is denoted `omegaD`. Once s_{Rn+1} is calculated, it is mapped back to s_{En+1} via T_2 using the key K_n. The key is used to identify a unique successor state in $\mathcal{M}_E$, as a state in $\mathcal{M}_R$ can be mapped to many states. The original coordinates of the agent are identified from K_n. Given this information and the result of the agent's turn, movement, and learning from $\mathcal{F}_R$, we can now calculate s_{En+1} (the unique successor state of s_{En}).

Note that s_{En} and s_{Rn}, and s_{En+1} and s_{Rn+1} can be matched because their cone of influence representations hold the same information. Specifically, the relative position between an agent and an obstacle is the same in both the Explicit and Relative models' representations. Therefore, the antenna signals (`sig` and `prevSig`) and the learning weight `omegaD` are also the same –as they are only affected by the relative position of an obstacle.

To formally prove the existence of a ϕ-simulation relation we must fulfil the conditions of Definition 2.2. We define AP_ϕ to be the set of all atomic propositions from the variables (represented in the Promela specification as) `sig`, `prevSig`, and `omegaD`. Our properties ϕ_1' and ϕ_2 contain only propositions from AP_ϕ.

We define $H_\phi \subset (S_E \times S_R)$ as follows: $H_\phi = \{(s_{En}, s_{Rn}) : s_{En} \in S_E \text{ and } s_R = T_1(s_E)\}$. For any state s_E, the agent's learning weight, represented as `omegaD`, and the relative position of any obstacle to it are the same in s_E and $T_1(s_E)$; so `sig`, `prevSig`, and `omegaD` are the same for both of states. Hence the atomic propositions in the formulas ϕ_1' and ϕ_2 are the same in s_E and $T_1(s_E)$. So for any $(s_E, s_R) \in H_\phi$, $L_\phi(s_E) = L_\phi(s_R)$.

We must now show that, for a transition (s_{En}, s_{En+1}) in $\mathcal{M}_E$, if $H_\phi(s_{En}, s_{Rn})$ then there exists an s_{Rn+1} such that (s_{Rn}, s_{Rn+1}) is a transition in $\mathcal{M}_R$ and $H_\phi(s_{En+1}, s_{Rn+1})$. We consider separately the cases where,

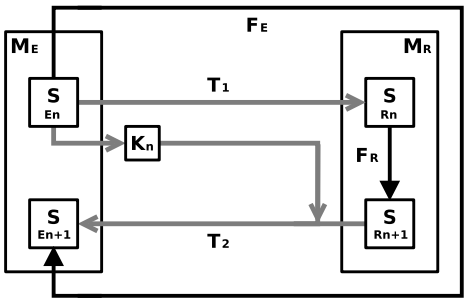

Fig. 16. Deterministic function mapping. Grey arrows indicate translations between models, and black arrows indicate transitions within models.

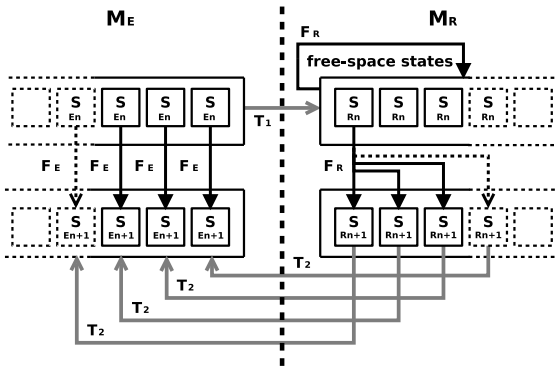

Fig. 17. Non-deterministic function mapping. Grey arrows indicate translations between models, and black arrows indicate transitions within models.

at s_{En}: there is an obstacle in the cone of influence representation, but no collision has occurred; there is no obstacle in the cone of influence representation; a collision has occurred. In each case it is sufficient to demonstrate a state s_{Rn+1} such that (s_{Rn}, s_{Rn+1}) is a transition in $\mathcal{M}_R$ and $T_2(s_{Rn+1}) = s_{En+1}$. In all cases we refer to the Abstract Promela model described in Section 5.1.

If there is an obstacle in the cone of influence, but no collision has occurred, the transition (s_{Rn}, s_{Rn+1}) in $\mathcal{M}_R$ is purely deterministic (depending on the precise position of the obstacle, the agent may turn, and then move forward, resulting in a change in the position of the obstacle in the cone of influence representation). Since T_2 is a direct translation back to the original cone of influence, clearly $T_2(s_{Rn+1}) = s_{En+1}$.

Suppose that there is no obstacle in the cone of influence. Then s_{En} is mapped via T_1 to a *free-space* state in $\mathcal{M}_R$. That is, state s_{Rn} in Figure 17 such that $H_\phi(s_{En}, s_{Rn})$. The set of free-space states in $\mathcal{M}_R$ have no obstacles in their cone of influences and the only difference between them is the value of the learning weight `omegaD`. From a free-space state s_{Rn} there is a non-deterministic choice to make a transition back to itself (no obstacle in the cone of influence and the same `omegaD` value), or to any state in which there is an obstacle at the boundary of the cone of influence (see the right hand side of Figure 17, $\mathcal{M}_R$). One of these states s_{Rn+1} will satisfy the condition $T_2(s_{Rn+1}) = s_{En+1}$.

If a collision has occurred at state s_{En} then a collision will have occurred in state s_{Rn}. It follows that s_{Rn} is again mapped to either the same free-space state (i.e. with learning weight unchanged) or to a state in which there is an obstacle at the boundary of the cone of influence, and the same argument applies as for the previous case.

5.3. Verification of the Abstract Model

We check properties ϕ_1' and ϕ_2, as defined in Section 4.2. Verifying only properties ϕ_1' and ϕ_2 is sufficient to justify our abstraction technique, however verifying other properties is possible (providing we can demonstrate a suitable ϕ-simulation). Property ϕ_1' states that the agent will eventually avoid all obstacles that

it contacts with its distal sensors and, hence, has learnt successful distal avoidance behaviour. Property ϕ_2 states that the agent will eventually be equal to, or less than its maximum level of learning (that learning stabilises). Note that additional verifications are still required to calculate the maximum level of learning –here the maximum level is 6 (`MAX_OMEGAD` = 6).

As with the Explicit models, verification was successful. In this case, there is only one verification required for each property –as opposed to running verifications on a set of explicit environments. The learning rate is again assumed to be 1. The minimum value of `MAX_OMEGAD` (for property to hold) is 6 and for both properties the number of stored states is 11813, the maximum search depth is 28617, and the elapsed time is 0.07 seconds.

From property ϕ_1' we can infer that in all variations of this type of environment, the agent eventually learns to stop all avoidable collisions. Hence, we assert that eventually it is always the case that if an agent detects an obstacle with a distal sensor it will avoid it.

Property ϕ_2 is verified with a maximum learning level of 6. We infer from this that responding to obstacles to this extent (with this level of learning) is sufficient to avoid all collisions for this type of environment –collisions that are detectable with distal sensors.

The representation of an Explicit model provides an exact cut off point as to when an agent will finish learning for a particular environment. This precise evaluation is important when using a specific environment. However, the Abstract model provides guarantees that have a much broader scope; i.e., wherever this system is deployed, if the environment fits the given specification, then we know that it will work correctly.

6. Extensions to the approach

The presentation in this paper is deliberately limited to a very simple case, namely one agent in a fixed size environment, with a variable number of obstacles. The number of obstacles and environment size can be changed easily by altering the value of constants.

The Promela models are deliberately structured so that they can be easily modified to verify different scenarios (more agents, different sized obstacles etc.). More details of the different possible scenarios have been discussed in [KMPD13].

All of our models assume that the system is well-behaved, i.e. that there is no uncertainty in the system. This is a little unrealistic: for example the turn angle is highly dependent on the current, often highly unpredictable, state of the battery. We could extend our approach to include this type of uncertainty by using probabilistic models which can be analysed using a probabilistic model checker, such as Prism[HKNP06].

Currently models are created by hand, but a simple next step would be to allow different models to be generated automatically by updating a fixed set of parameters.

7. Related Work

There are several avenues of research which have some overlap with our analysis of agent-based learning systems. We discuss some of these in this section, highlighting the differences with our approach.

The subject of motion planning for robots with similar goals to ours (such as obstacle avoidance) is covered in [FGKGP09]. As with our work, the focus is on having formal guarantees that the continuous motion of a robot satisfies a specific temporal logic formula. However the approach to calculating the motion of the robot is discretized such that the robot only has macro movements between cellular divisions of an environment. In addition, the position of the robot within an environment and the specific sequence of movements is of greater importance than with our models (e.g., did the robot move around the environment in the correct order, from area `r1` to `r2`, to `r3`, etc?).

Verification of an agent-based system is also considered in the domain of human-robot interaction in [SDDF12]. Here the scenario of robot helpers is discussed, where these systems are represented in the multi-agent modelling, simulation, and development environment, Brahms [Sie01]. Like our approach, Promela and Spin are used to formally verify system properties. However, in order to verify with Spin an automatic converter from Brahms to Promela is used.

There has been considerable work done analysing multi-agent systems by using Kripke modelling techniques. In [JTZW06] the behaviour of Unmanned Ariel Vehicles (UAVs) is formally analysed in the design phase through Kripke modelling and then model checking. Specifically, a group of UAVs are formalised with

a Kripke model, then properties of communication between the UAVs are expressed in temporal logic and verified using model checking tools. In [Hum13] a detailed description of Promela and the use of Spin to verify scenarios involving a team of UAVs is presented. This work demonstrates the usefulness of a step-by-step description of the use of Spin. They do not however use embedded embedded C or use (and justify) abstraction like we do.

In [KB06] and [KB07], multiple agents act as a swarm system where they are analysed in hierarchical layers of abstraction in order to use temporal logic to formally capture behavioural information (for a specific layer of abstraction). In the case study presented in [KB06], swarm robots navigate an environment via specific locations while avoiding: colliding with other swarm members, crashing into large polygonal obstacles, and moving outside a given area of a swarm cluster. These papers present a fully automated framework in which swarm-robot control-laws can be constructed, where controlling the essential features of a swarm is dealt with as a model checking problem.

Similar work on swarm systems uses model checking to verify whether given temporal logic properties are satisfied by all possible behaviours of a swarm [DWFZ12]. Here the focus is on a particular swarm control algorithm which has been used and tested on real systems. The algorithm is refined using temporal analysis via model checking. This process of refinement involves iterating from highly abstract models to much more detailed models which are, ideally, to a level of detail equivalent to that of the real systems. In [AICE15] an approach similar to ours of viewing the system from the robot's perspective in order to reduce the size of the state-space was used. Our research shares the concept of applying temporal analysis via model checking to an already existing system (using it as a reference point to assess results and refine models). A recent survey [FDW13] addresses the more general question of how to verify autonomous systems behaviours.

In [KDF12] probabilistic model checking is applied to swarm systems. Model checking is proposed as an alternative to the common analysis of simulation –as we also advocate. In this case, the probabilistic model checker Prism is used to verify formulas relating to the behaviours of the swarms systems. In particular, the energy usage of a foraging swarm colony is analysed.

Although much of this related work applies similar analysis and techniques that we use, it does not, however, completely overlap with our research. Much of the work agrees with our assertion of the common approach of simulation being insufficient to formally verify given properties of a system, and each also proposes model checking as a potential solution. In our work we uniquely focus on the modelling of an unsupervised learning algorithm (ICO learning) as part of the agent in our model. We also utilise Promela's embedded C code to provided highly detailed models of our systems without producing intractable state-spaces. This removes the need to apply the methodology of using hierarchical levels of abstraction, or alternate specialised languages, in order to formally analyse properties of our system.

In [KMPD13] we introduced the idea of using model checking to analyse a robot system with learning. Whereas in that paper we were concerned with introducing the concept of model checking to a non-formal audience as a way to compliment the traditional simulation approach, here we focus more on the details. We describe in detail the use of embedded C code within Promela and use it to allow for an accurate representation of robot movement. We also include a full proof of our abstraction technique that allows us to capture the behaviour of many environments within a single model.

8. Conclusion and Future Work

Model checking is an established technique for verifying finite state systems. Spin is a popular open source model checker that has been freely available for over 20 years, and whose development has been influenced by its wide community of users. Although Spin is one of the more straightforward formal verification tools (it is often used in the teaching of undergraduate formal methods courses, for example), it is not always possible to use the basic tool alone in order to verify the behaviour of complex systems. In this paper we describe two more advanced techniques that enhance the power of model checking with Spin, namely the use of embedded C code within a Promela specification (to allow us to accurately reflect robot movement), and abstraction with proof of soundness. Our abstraction allows us to model a whole class of environments (initial conditions) using a single model. Without it, robot behaviour in any two different environments must be modelled using a separate Promela specification.

We illustrate these techniques using a real case study, namely the verification of a robot system with learning. We have made several simplifying assumptions about our system. For example, we assume that only one obstacle at a time can appear in the cone of influence of a robot, there is only one robot within

the environment, and the obstacles are all of uniform size and shape. In order to illustrate our technique these assumptions were necessary, but their relaxation would require very little adjustment to our models. In future work we aim to automate the generation of models, reusing as much of the C code as possible, to allow more complex environments and robot configurations to be considered.

References

References

[AICE15] L. Antuna, D. Ilan, S. CAmpos, and K. Eder. symmetry reduction enables model checking of more complex emergent behaviours of swarm navigation algorithms. In *Towards Autonomous Robotic Systems*, volume 9287 of *Lecture Notes in Computing Science*, pages 26–37. Springer, 2015.

[Bra84] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Colorado, 1984.

[Büc60] J. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Science*, Stanford University Press, pages 1–12, 1960.

[Cat94] T. Cattel. Modeling and verification of a multiprocessor realtime OS kernel. In D. Hogrefe and S. Leue, editors, *Proceedings of the 7th WG6.1 International Conference on Formal Description Techniques (FORTE '94)*, volume 6 of *International Federation For Information Processing*, pages 55–70, Berne, Switzerland, October 1994. Chapman and Hall.

[CGM+97] Alessandro Cimatti, Fausto Giunchiglia, Giorgio Mingardi, Dario Romano, Fernando Torielli, and Paolo Traverso. Model checking safety critical software with SPIN: an application to a railway interlocking system. In R. Langerak, editor, *Proceedings of the 3rd SPIN Workshop (SPIN'97)*, pages 5–17, Twente University, The Netherlands, April 1997.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[CM08] M. Calder and A. Miller. An automatic abstraction technique for verifying featured, parameterised systems. *Theoretical Computer Science*, 404(3):235–255, September 2008.

[Dil96] D. Dill. The Murϕ verification system. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, USA, July/August 1996. Springer-Verlag.

[DWFZ12] C. Dixon, A. Winfield, M. Fisher, and C. Zeng. Towards temporal verification of swarm robotic systems. *Robotic and Automomous Systems*, 60:1429–1441, 2012.

[FDW13] M. Fisher, L. Dennis, and M. Webster. Verification of autonomous systems. *Communications of the ACM*, 56(9):84–93, 2013.

[FGKGP09] G. Fainekos, A. Girard, H. Kress-Gazit, and G. Pappas. Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343 – 352, 2009.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. *LNCS*, 3920/2006:441–444, 2006.

[Hol04] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Pearson Education, 2004.

[Hum13] L. Humphrey. Model checking for verification in uav cooperative control applications. volume 444 of *Lecture Notes in Computer Science*, pages 69–117. Springer, 2013.

[JTZW06] S. Jeyaraman, A. Tsourdos, R. Zbikowski, and B. White. Kripke modelling approaches of a multiple robots system with minimalist communication: a formal approach of choice. *International journal of system science*, 37(6):339–349, 2006.

[KB06] M. Kloetzer and C. Belta. Hierarchical abstractions for robotic swarms. In *proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2006)*, pages 952–957, 2006.

[KB07] M. Kloetzer and C. Belta. Temporal logic planning and control robotic swarms by heirarchical abstractions. *IEEE Transactions on robotics*, 213(2):320–330, 2007.

[KDF12] S. Konur, C. Dixon, and M. Fisher. Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems*, 60(2):199 – 213, 2012.

[Kir14] Ryan F. Kirwan. *Applying Model Checking to Agent-based learning systems*. PhD thesis, University of Glasgow, February 2014.

[KKT+10] T. Kulvicius, C. Kolodziejski, T. Tamosiunaite, B. Porr, and F. Worgotter. Behavioral analysis of differential Hebbian learning in closed-loop systems. *Biological cybernetics*, 103(4):255–271, 2010.

[KL02] S. Kumar and K. Li. Using model checking to debug device firmware. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI 2002)*, Boston, MA, December 2002. USENIX.

[KMP] R. Kirwan, A. Miller, and B. Porr. Online appendix: `http://www.dcs.gla.ac.uk/people/personal/alice/appendices/publications/FAC2016_Appendix.pdf` .

[KMPD13] R. Kirwan, A. Miller, B. Porr, and P. Di Prodi. Formal modelling of robot behaviour with learning. *Neural Computation*, 25(11):2976–3019, November 2013.

[Kri63] S. Kripke. Semantical considerations on modal logics. *Acta Philosophica Fennica*, 16:83–94, 1963.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Boston, 1993.

[Mer00] S. Merz. Model checking: A tutorial overview. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modeling and*

Verification of Parallel Processes, 4th Summer School, MOVEP 2000, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38, Nantes, France, June 2000. Springer-Verlag.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.

[MOSS99] M. Müller-Olm, D. Schmidt, and B Steffen. Model-checking: A tutorial introduction. In A. Cortesi and G. File, editors, *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science (LNCS)*, pages 330–354, Venice, Italy, September 1999. Springer-Verlag.

[PvFW03] Bernd Porr, Christian von Ferber, and Florentin Wörgötter. ISO-learning approximates a solution to the inverse-controller problem in an unsupervised behavioural paradigm. *Neural Computation*, 15(4):865–884, 2003.

[PW06] B. Porr and F. Wörgötter. Strongly improved stability and faster convergence of temporal sequence learning by utilising input correlations only. *Neural Computation*, 18(6):1380–1412, 2006.

[SB87] R. Sutton and A. Barto. A temporal-difference model of classical conditioning. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, pages 355–378, Seattle, Washington, July 1987 1987.

[SDDF12] R. Stocker, L. Dennis, C. Dixon, and M. Fisher. Verifiying Brahms human-robot teamwork models. In *proceedings of the 13th European conference on logics in Artificial Intelligence (JELIA-2012)*, pages 385–397, 2012.

[Sie01] M. Sierhuis. *Modeling and simulating work practice. BRAHMS: A multi-agent modeling and simulation language for work system analysis and design*. PhD thesis, University of Amsterdam, 2001.

[SLM+09] O. Sharma, J. Lewis, A. Miller, A. Dearle, D. Balasubramaniam, R. Morrison, and J. Sventek. Towards verifying correctness of wireless sensor network applications using insense and spin. In C. Păsrăeanu, editor, *Proceedings of the 16th International SPIN Workshop (SPIN 2009)*, Lecture Notes in Computer Science, pages 223–240, Grenoble, France, June 2009. Springer-Verlag.

[WBBK11] M. Weissman, S. Bedenk, C. Buckl, and A. Knoll. Model checking industrial robot systems. In A. Groce and M. Musuvathi, editors, *Proceedings of the 18th International SPIN Workshop (SPIN 2009)*, volume 6823 of *Lecture Notes in Computer Science*, pages 161–176, Snowbird, UT, USA, July 2011. Springer-Verlag.

[YT01] C. Yuen and Tjioe. Modeling and verifying a price model for congestion control in computer networks using Promela/Spin. In M.B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 272–287, Toronto, Canada, May 2001. Springer-Verlag.