Model checking For Improved Adaptive Behaviour

Alice Miller∗, Ryan Kirwan†, Bernd Porr‡, Paolo di Prodi†

∗Department of Computer Science, University of Glasgow, Glasgow, UK, alice.miller@glasgow.ac.uk,
†Department of Computer Science, University of Glasgow, Glasgow, UK, kirwanr@dcs.gla.ac.uk,
‡School of Electrical Engineering, University of Glasgow, Glasgow, UK, bernd.porr@glasgow.ac.uk,
†School of Electrical Engineering, University of Glasgow, Glasgow, UK, robomotic@gmail.com

Keywords: Model checking, robotics, adaptive behaviour, PROMELA.

Abstract

Closed loop systems are traditionally analysed using simulation. We show how a formal approach, namely model checking, can be used to enhance and inform this analysis. Specifically, model checking can be used to verify properties for any execution of a system, not just a single experimental path. We describe how model checking has been used to investigate a system consisting of a robot navigating around an environment, avoiding obstacles using sequence learning. We illustrate the power of this approach by showing how a previous assumption about the system, gained though vigorous simulation, was demonstrated to be incorrect, using the formal approach.

1 Introduction

Simulation is commonly used to investigate behaviour of adaptive agents [2, 20, 23] to benchmark their performance [11]. While simulation is a relatively inexpensive method for determining the behaviour of a system it cannot guarantee that the agent fails in a rare situation. It cannot determine properties of the form in all cases property P holds, or it is never true that property Q holds. In order to overcome this problem we introduce a formal method called model checking which guarantees that certain properties hold for any run of the system (i.e. that should hold for any experiment).

Formal methods have been used to analyse agent-based systems [6, 8, 9, 25]. However, unlike our approach, previous work does not relate simulations of a real system to the application of an appropriate automatic formal technique to verify the same system. In this way our model checker adds value to the real system by pointing out weaknesses of the real system, for example, a non optimal sensor setup.

In this paper, we describe a formal model of an adaptive closed-loop system in which an agent’s behaviour changes from being reactive to proactive [17, 19]. In particular we use a simple obstacle avoidance scheme where an agent has to learn to use a distal sensor to steer away from objects.

A system comprised of learning agents has an inherent risk of becoming unstable [17], especially at high learning rates. Model checking is particularly valuable for identifying this sort of behaviour. Our model is specified in the model specification language Promela, and verified using the model checker SPIN. We will focus on a simple obstacle avoidance scenario with learning as described in [11]. We show that model checking is able to diagnose deficiencies in the agent’s design which have not been detected using classical closed-loop simulation. Thus, we demonstrate that model checking provides important insights into the dynamics of the system and can be used to improve the design of the agent.
2 The simulation Environment

In our simulation environment the robot moves in a grid of pixels. The ability of a robot to successfully navigate its environment without colliding with obstacles is used to assess the agent’s learning algorithm and sensor configuration. This can also be interpreted as a control system which turns a reflexive behaviour into a forward model [16] which is shown in Fig. 1.

We start with a fixed reflex via the proximal sensor $x_p$ which causes the robot to do a sharp turn as soon as it has touched an obstacle (Fig. 1B). The purpose of learning is to generate an anticipatory action using a distal sensor ($x_d$, Fig. 1A) so that the agent safely steers away from the obstacle without bumping into it. The sensor signals $x_d$ and $x_p$ are generated by probing the pixel values from the left and right corners of the robot and calculating their differences (see Fig 2). The resulting differences for both the proximal and distal sensors are then fed into first order low pass filters ($f_c = 0.1$) for smoothing and then summed to generate the new steering angle:

$$v = x_p + \omega_d x_d$$  \hspace{1cm} (1)

At every time-step the robot moves forward one pixel at angle $\phi$ (from North). The steering angle $v$ is added to $\phi$ every time-step. I.e. $\phi(t + 1) = \phi + v$.

Learning is achieved by increasing the weight $\omega_d$ by $\lambda$ every time the robot bumps into an obstacle ($x_p \neq 0$) and at the same time the distal signal has been active ($x_d \neq 0$) before. This implements a simple version of temporal sequence learning. Such a learning rule causes the weight $\omega_d$ to increase until the agent no longer uses its reflex reaction via $x_p$. For a detailed treatment of behaviour based robotics and learning of anticipations we refer to [16, 17, 21].

3 Properties

Systems described above are deemed to be _successful_ if eventually the proximal reflex is no longer triggered ($x_p = 0$). In order to test stability one usually checks that the forward model converges which, in the simplest case, is if the weight $\omega_d$ stabilises. This can be now formulated as properties:

1. The sensor input $x_p$ of the proximal sensor will eventually stay zero, indicating that the agent is using only its distal sensors.

2. The weight $\omega_d$ will eventually become constant, indicating that the agent has finished learning.

3. The time $\tau$ between distal and proximal events will continue to rise until there are no longer any impacts on the proximal sensors ($\tau \to \infty$).

Our goal is to verify the properties above using model checking. In section 5 we describe how the system is specified in Promela.
4 Model checking

Errors in system design are often not detected until the final testing stage when they are expensive to correct. Model checking [4, 5] is a popular method that helps to find errors quickly by building small logical models of a system which can be automatically checked.

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

A software tool called a model checker then exhaustively searches the finite state model, checking whether it satisfies φ. In Linear Time Temporal Logic (LTL) model checking, this involves checking that φ holds for all paths of the model. If φ does not hold for some path, an error trace or counter-example is reported. Manual examination of this counter-example by the system designer can reveal that the specification does not adequately represent the behaviour of the system, that φ does not accurately describe the given requirement, or that there is an error in the design. In this case, either P, φ, or the system design (and thus also P and possibly φ) must be modified, and re-checked.

4.1 Promela and Spin

The model checker SPIN [10] allows one to reason about specifications written in the model specification language Promela. SPIN has been used to trace logical errors in distributed systems designs, such as operating systems [3, 12], computer networks [26], wireless sensor network communication protocols [18] and industrial robot systems [24].

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 parametrised 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 parametrised, the init process can be omitted and processes declared to be immediately active, via the active keyword. 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. We do not include details of LTL in this paper. Instead we state our properties in words only.

The system that we are modelling here is not concurrent: there is just a single robot moving in an environment. However SPIN has the benefit of allowing us to check every path through a model for counter-examples (i.e. paths that violate a given property), without having to manually construct a set of test cases. Future work will involve us adding additional robots. This will be a simple case of adding further instantiations of the robot process template.

5 The Promela Specification

We assume that the environment, the robot and the obstacles are all circular. The diameter of the robot is 40 units, and the diameter of an obstacle is 20 units. The reach of the proximal and distal sensors is 10 and 60 units respectively and the angle between the left and right sensors is 60°. The environment has a diameter of at least 100 units.

We assume that the complexity of the environment is such that at most one obstacle can touch any part of the robot at any time. We denote the maximum allowable complexity as δ0.

An environment is a circular region, represented by a set of polar co-ordinates C = {(r, θ : 0 ≤ r ≤ ρ, 0 ≤ θ < 360)}, where ρ is the radius of the environment and angles are measured clockwise from North. We use polar coordinates so that the turning angles of the agent can be represented to an accuracy of one degree. The current position of the robot and the obstacles are stored as polar coordinates (with origin the centre of the environment).

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

The precise location of the robot as it moves around an environment is calculated using C-code embedded within our Promela specification. At each time step the new direction of the robot is calculated from the signals received from the sensors. We simply calculate the position of any obstacle touching the sensors to infer this information. As well as deciding the new direction of the robot, the angular response to a sensor impact will be incremented by a fixed amount (the learning rate) if a collision occurs at the proximal sensor.

As in the simulated environment, we ignore the presence of any boundary wall. When a robot reaches the perimeter of the environment it is simply relocated to the opposite edge of the perimeter. (I.e. as if the environment were to wrap round, like a torus.)

For space reasons we do not include our Promela specification here. All models and LTL files can be obtained from the authors.

5.1 Verification results for the Promela specification

Our models are defined separately for each learning rate $\lambda$ and environment (i.e. location of obstacles). Using this model we can not verify properties for any environment, but can run separate verifications for different environments. We fix our learning rate to 1, and verify our properties for an example set of environments.

To prove property 1, we used an LTL formula stating that: along any path of the model, we will eventually reach a point at which the proximal signal remains zero.

Attempts to verify this property using SPIN proved the property to be false. Examination of a counter-example trace showed that it was indeed always possible to have an impact on the proximal sensor, even when learning had ceased. This happens when the robot approaches an obstacle straight on, and the obstacle impacts without the distal sensor touching the obstacle. Since learning only occurs when an impact on the proximal sensor follows an impact on the distal sensor, we can rephrase the property to eliminate this rare behaviour. The new property (Property 1A) is: along any path of the model, we will eventually reach a point from which it is always true that the proximal signal is either zero, or a proximal signal is not zero but is not correlated to a previous distal signal. This property is shown to be true for our sample set of example environments.

To prove property 2, in each case we performed initial experiments to find the maximum value of $\omega_d$, call this value $Max$. Then, for the associated value we checked a slightly modified version of Property 2, namely Property 2A, to verify that $MAX$ is eventually reached, but never exceeded. This property was shown to be true for our set of example environments.

It is not possible to measure time using SPIN alone. To do this a timed model checker such as Uppaal [13] should be used. However, since we have only one active process (the robot), time between events is proportional to the number of transitions occurring between them. It would therefore be possible to prove a similar, but alternative property to property 3. However, were more robots added to the system (which is our eventual goal) this reasoning would no longer apply. Concurrent events are executed sequentially by SPIN, and so there is no direct correlation between the number of global transitions and the time between events related to a given process (robot). For this reason, we do not pursue this approach here.

6 Discussion

The advantage of the model-checking approach is that we can simply specify LTL properties to define behaviour that is expected to hold for all paths for our model. I.e. we do not have to run an exhaustive set of simulations to verify behaviour – the model checker will find any error path if it exists. Having the capacity to examine error trails allows us to not only debug our models, but to identify the pathological case in which one of the initial properties does not hold (i.e. the situation in which the robot hits an obstacle dead on, without it first making contact with a distal sensor). This allows us to strengthen the property to ignore this unusual case.

In terms of the actual robot design model checking allows us to identify deficiencies in the agent during the early design stages. That the robot cannot see obstacles which are hitting it dead on is obviously a deficiency of its sensor system. While simple to spot in our example, more
complex sensor motor setups will make it much more difficult to identify deficiencies which might happen only in rare cases. If these cases cause damage to the robot or a deterioration its performance then they need to be tackled appropriately. Model checking can help to identify these problems in the early design phase of a robot and lead to a more reliable system.

Learning agents introduce a potential risk when implemented in mission critical scenarios. A rare case might cause weights to grow exponentially and thus cause a deterioration of the behaviour. As far as we are aware, we are the first to verify agent learning using model checking. While formal aspects of multi-agent systems are the subject of an annual workshop [7], approaches tend to focus on verifying communication protocols, formalising goals and plans and knowledge-based agents. A number of methods for formally specifying multi-agent systems, with a view to prototyping and/or verification have been proposed [6, 9, 25]. One particular approach [8] uses a temporal logic framework to specify behaviour of individual agents as well as systems of agents. Refinement is used to reason about behaviour, as well as verification via logical deduction. The framework is extended to include the concepts of knowledge and belief, but learning is not considered. Model checking has been used in [1]. Agents are specified in the logic-based agent-oriented programming language AgentSpeak and the specification of the system is automatically converted into Promela or Java, for verification with SPIN or the Java Pathfinder tool [22]. This approach does not consider agent learning, nor does it model collision avoidance or use abstraction as we do. Learning is only considered in the context of model checking when it is used as a way to enhance model checking algorithms [14, 15]. In summary, we have been the first to verify agent learning using model checking. This approach is suitable for analysing learning in mission critical situations. Learning has hitherto been avoided by industry in this context because of the potential risks involved. Model checking would be an invaluable tool in this kind of scenario, allowing for increased confidence that a learning system would stay stable in previously unexplored environments.

7 Conclusions

Model checking is a powerful tool that allows us to check temporal properties of a (model of a) system. In this paper we have shown how the SPIN model checker can be used to verify properties of a system that has previously been analysed using simulation. The system, consisting of a robot navigating around an environment using learning to avoid obstacles, serves as an instructional example for the technique of model checking and its use within this context. We have described our Promela model and how we verified some example LTL properties for it.

References


