# Mining Invariants from Reachable State Space Observations*

Ben Lloyd-Roberts[1], Michael Edwards[1], and Phillip James[1]

Swansea University, United Kingdom
{ben.lloyd-roberts, michael.edwards, p.d.james}@swansea.ac.uk

**Abstract.** The application of formal methods to verify that railway signalling systems operate correctly is well established within academia and is beginning to see real applications. However, there is yet to be a substantial impact within industry due to current approaches often producing false positive results that require lengthy manual analysis. It is accepted that invariants, properties which hold for all states under which a signalling system operates, can help reduce occurrences of false positives. However automated deduction of these invariant remains a challenge. In this work we report on using reinforcement learning to explore state spaces of signalling systems and generate a dataset of state spaces from which we envisage invariants could be mined. Our results suggest the viability of reinforcement learning in both maximising state space coverage and estimating the longest loop free path for state spaces. Additionally, we use experiences observed by our agents to compute the phi coefficient between sets of variables across unique states. This allows us to both infer invariant properties across states and inform agent exploration by converging coefficients over time.

**Keywords:** First keyword · Second keyword · Another keyword.

## 1 Introduction

Review / reframe for invariant approach

Model checking is a formal verification technique stemming from the need to systematically check whether certain properties hold for different configurations (states) of a given system. Given a transition system $T$ and a formula (or property) $F$, model checking attempts to verify through refutation that $s \vdash F$ for every system state $s \in T$, resulting in $T \vdash F$.

The application of model checking to verify railway interlockings has a long history within academia and has seen some real applications in industry. As early as 1995, Groote et al. [14] applied formal methods to verify an interlocking for controlling the Hoorn-Kersenbooger railway station. Newer approaches to interlocking verification have also been proposed in recent years [12, **?**,**?**]. This includes work by Linh et al. [34] which explores the verification of interlockings

* Supported by EPSRC and Siemens Mobility UK

written in a similar language to Ladder Logic using SAT-based model checking. In spite of this, such approaches still lack widespread use within the railway industry.

In particular, one of the limitations of such model checking solutions is that verification can fail due to over approximation, typically when using techniques such as inductive verification [28]. Inductive verification fails to consider whether system states that violate a given property are indeed reachable by the system from a defined initial configuration. These false positive often require manual inspection, alternatively one solution is to introduce so-called invariants to suppress false positives [3]. Invariants are properties that hold for sub-regions of the state space. The aim is to introduce invariants that help bound the region of reachable states when model checking. However, generating sufficiently strong invariants automatically is complex [7].

In this work, we take first steps towards using machine learning to generate invariants by providing a first formal mapping of interlocking based state spaces to a reinforcement learning (RL) environment. We then explore how such state spaces can be generated in a controlled manner to test the scalability of our approach on environments where the number of reachable states is known. Second, we provide an analysis of how various reinforcement learning algorithms can be used to effectively explore state spaces in terms of their coverage. We see this as a first step towards mining invariants from such state spaces as this approach would indeed require reasonable coverage. Finally, we reflect upon future work in directing our approach to improve exploration and learn invariants from a dataset of state sequences generated by our RL agents.

## 2 Preliminaries

We briefly discuss model checking of railway interlockings and reinforcement learning. For further details we refer the reader to [20, **?**] and [25, 30, 24] respectively.

### 2.1 Ladder Logic & Interlockings

Interlockings serve as a filter or 'safety layer' between inputs from railway operators, such as route setting requests, ensuring proposed changes to the current railway state avoid safety conflicts. As a vital part of any railway signalling system, interlockings are critical systems regarded with the highest safety integrity level (SIL4) according to the CENELEC 50128 standard.

Ladder logic is a graphical language widely used to program Programmable Logic Controllers[33] and in particular the Siemens interlocking systems we consider in this work. From an abstract perspective, ladder logic diagrams can be represented as propositional formulae. Here we follow the definition of James et al. [17]. A ladder logic rung consists of the following entities. *Coils* represent boolean values that are stored for later use as output variables from the program. A coil is always the right most entity of the rung and its value is computed by

executing the rung from left to right. *Contacts* are the boolean inputs of a rung, with *open* and *closed* contacts representing the values of un-negated and negated variables respectively. The value of a coil is calculated when a rung fires, making use of the current set of inputs – input variables, previous output variables, and output variables already computed for this cycle – following the given connections. A horizontal connection between contacts represents logical conjunction and a vertical connection represents logical disjunction. An interlocking executes such a program from top-to-bottom over and over, indefinitely.

More formally, following [17] a ladder logic program is constructed in terms of disjoint finite sets $I$ and $C$ of input and output/state variables. We define $C' = \{c' \mid c \in C\}$ to be a set of new variables in order to denote the output variables computed by the interlocking in the current cycle.

*Defn 1. Ladder Logic Formulae:* A ladder logic formula $\psi$ is a propositional formula of the form

$$\psi \equiv (c'_1 \leftrightarrow \psi_1) \wedge (c'_2 \leftrightarrow \psi_2) \wedge \ldots \wedge (c'_n \leftrightarrow \psi_n)$$

Where each conjunct represents a rung of the ladder, such that the following holds for all $i, j \in \{1, \ldots, n\}$:

- $c'_i \in C'$ (i.e. $c'$ is a coil)
- $i \neq j \rightarrow c'_i \neq c'_j$ (i.e. coils are unique)
- $vars(\psi_i) \subseteq I \cup \{c'_1, \ldots, c'_{i-1}\} \cup \{c_i, \ldots, c_n\}$ (i.e. the output variable $c'_i$ of each rung $\psi_i$, may depend on $\{c_i, \ldots, c_n\}$ from the previous cycle, but not on $c_j$ with $j < i$, due to the nature of the ladder logic implementation, those values are overridden.)

Figure 1, shows a ladder logic formulae for a simple ladder logic program for controlling a pelican crossing. For full details of the program we refer the read to [17]. However for understanding here, consider line one and three. In line 1, the value of the coil CROSSING' is calculated based upon the inputs REQ and CROSSING. Whereas in line 3, the value of the coil TL_1_G (representing a traffic light being set to green), is calculated based upon the already compared coils CROSSING' and REQ' along with the input PRESSED. This illustrates the imperative nature of ladder logic in referencing variables.

### 2.2 Transition Systems and Model Checking for Ladder Logic

For this work, we have concentrated on trying to produce invariants for the approaches taken by Kanso et al. [20] and James et al. [17]. Here we include their model of ladder logic based railway interlocking programs as we use this as a basis for defining a learning environment.

Building upon the propositional representation of a ladder logic program given in Section 2.1, we can define, following [17], the semantics of a ladder logic program in terms of labelled transition systems.

$((\text{CROSSING}' \leftrightarrow (\text{REQ} \wedge \neg\ \text{CROSSING})) \wedge$
$(\text{REQ}' \leftrightarrow (\text{PRESSED} \wedge \neg\ \text{REQ})), \wedge$
$(\text{TL\_1\_G}' \leftrightarrow ((\neg\ \text{CROSSING}') \wedge (\neg\ \text{PRESSED} \vee \text{REQ}'))) \wedge$
$(\text{TL\_2\_G}' \leftrightarrow ((\neg\ \text{CROSSING}') \wedge (\neg\ \text{PRESSED} \vee \text{REQ}'))) \wedge$
$(\text{TL\_1\_R}' \leftrightarrow \text{CROSSING}') \wedge$
$(\text{TL\_2\_R}' \leftrightarrow \text{CROSSING}') \wedge$
$(\text{PL\_1\_G}' \leftrightarrow \text{CROSSING}') \wedge$
$(\text{PL\_2\_G}' \leftrightarrow \text{CROSSING}') \wedge$
$(\text{PL\_1\_R}' \leftrightarrow \neg\ \text{CROSSING}') \wedge$
$(\text{PL\_2\_R}' \leftrightarrow \neg\ \text{CROSSING}') \wedge$
$(\text{AUDIO}' \leftrightarrow \text{CROSSING}'))$

Fig. 1: Ladder logic program for pelican crossing.

Let $\{0,1\}$ represent the set of boolean values and let

$$Val_I = \{\mu_I \,|\, \mu_I : I \to \{0,1\}\} = \{0,1\}^I$$
$$Val_C = \{\mu_C \,|\, \mu_C : C \to \{0,1\}\} = \{0,1\}^C$$

be the sets of valuations for input and output variables.

The semantics of a ladder logic formula $\psi$ is a function that takes the two current valuations and returns a new valuation for output variables.

$$[\psi] : Val_I \times Val_C \to Val_C$$
$$[\psi](\mu_I, \mu_C) = \mu'_C$$

where $\mu'_C$ is computed as follows: the value of each variable $c_i$ is computed using the $i$th rung of the ladder, $\psi_i$, using the valuations $\mu_C$ and $\mu_I$ from the last cycle and the current valuations restricted to those evaluated be fore the current variable. We refer the reader to [17] for full details.

*Defn 2. Ladder Logic Labelled Transition System:* We define the labelled transition system $LTS(\psi)$ for a ladder logic formula $\psi$ as the tuple $(Val_C, Val_I, \to, Val_0)$ where

- $Val_C = \{\mu_C | \mu_C : C \to \{0,1\}\}$ is a set of states.
- $Val_I = \{\mu_I | \mu_I : I \to \{0,1\}\}$ is a set of transition labels.
- $\to \subseteq Val_C \times Val_I \times Val_c$ is a labelled transition relation, where $\mu_C \xrightarrow{\mu_I} \mu'_C$ iff $[\psi](\mu_I, \mu_C) = \mu'_C$.
- $Val_0 \subseteq Val_C$ is the set of initial states.

We write $s \xrightarrow{t} s'$ for $(s,t,s') \in R$. A state $s$ is called *reachable* if $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$, for some states $s_0, \ldots, s_n \in Val_C$, and labels $t_0, \ldots, t_{n-1} \in Val_I$ such that $s_0 \in Val_0$.

Consider Figure 3, which illustrates multiple models of a simple ladder logic program for controlling a pelican crossing as considered by James et al. [17]. One model highlighted is, as defined, a ladder logic LTS. We can see that states

contain Boolean valuations for the ladder logic variables (for the LTS model we note the input variables below the dashed line at the bottom of each state are not included in the state variables). For example, state S1 shows that the variable CROSSING is 0 (i.e. false) in that state. Transitions are labelled with (for our purposes here the blue labels) inputs and their Boolean valuations. For instance the arrow from S1 to S2 is labelled with the input PRESSED = 1. Finally we can also see one initial state, state S0 (the state with dotted edges), where all variables are set to false.

### 2.3 Reinforcement Learning and MDPs

Reinforcement Learning (RL) is a machine learning paradigm with demonstrably impressive capacity for modelling sequential decision making problems as the optimal control of some incompletely-known Markov Decision Process (MDP) [31]. *Defn 3. Markov Decision Process*: A finite discounted Markov Decision Process $M$ is a five tuple $(S, \mathcal{A}, P_a(s, s'), R_a(s, s'), \gamma)$, where

- $S$, is a finite set of states, known as the observation space or state space, representing the model state at discrete time steps.
- $\mathcal{A}$, describes the action space; a set of actions performable at discrete time steps, used to compute new states from the observation space.
- $P_a(s, s') = P(s_{t+1} = s' | s_t, a_t)$, describe state transition probabilities; the likelihood of observing state $s_{t+1}$ given action $a_t$ is taken from state $s_t$.
- $R_a(s, s')$ is a reward function feeding a scalar signal, $r$ back to the agent at each time step $t$.
- $\gamma \in [0, 1]$ is a discount scalar successively applied at each time step.

In an RL setting we refer to the MDP as our environment, $\mathcal{E}$ where through simulation, software agents sample actions $a \in \mathcal{A}$, observe changes in states, $s \in S$ and learn to optimise some objective based on rewards, $r$ issued over discrete time steps $t$. Simulation can be continuous, where agents indefinitely interact with the environment until some termination criterion is met, such as reaching some reward threshold. Alternatively tasks may be episodic, where training is conducted over a sequence of episodes, defined by a finite number of time steps. It is implicitly assumed ensuing algorithmic descriptions or problem formulation in this work refers to episodic cases. An agent's trajectory, $\tau = (s_1, a_1, r_1, s_2, a_2, r_2, ..., s_h, a_h, r_h)$, summarises experience accumulated over a single episode or continuous training run. Here, $h$ refers to the horizon; a time step beyond which rewards are no longer considered. Rewards observed from time step $t$ up to some terminal time step $T$, are denoted $G_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$, and referred to as the return. The discount factor $\gamma \in [0, 1]$ downscales rewards over return time steps while future rewards are reduced as the discount exponent $i - t$ increases. This helps prioritise immediate rewards over distant ones and to enumerate returns over a potentially infinite horizon.

Two principle challenges of RL are those of prediction and control. The first refers to approximating the value function used to estimate state values $v(s)$ or

state-action values $q(s, a)$, describing the benefit of being in that state. Values are differentiable and updated based on an empirical average, known as the expectation, taken over observed returns, $\mathbb{E}[G_t|S_t = s, A_t = a]$. In other words the expectation over return $G_t$ depends on being in state $S_t = s$, having taken action $A_t = a$ during the present time step. Observed returns depend on the actions sampled at discrete time steps. The second challenge of control concerns optimising this selection process via the policy $\pi(a|s)$; a probability distribution mapping states to actions most likely to maximise the reward objective. Ultimately an optimal value function will converge to an optimal policy [23]. In practice, to scale with complex environments, these functions are parametric and approximated with gradient-based methods, such as stochastic gradient ascent.

Determining the reachable state space for ladder logic programs is often intractable, making the complete MDP model unknown to us without computing transitions for all state-action pairs. Consequently we trial several gradient-based learning algorithms using deep neural network state representations to approximate both value function and policy subject to the reward objective. First, a simple Deep Q-Network [25] is trained on small environments before applying more advanced approaches. Among this family of approximate reinforcement learning algorithms are policy gradient methods [19]. Such algorithms utilise parametrised policies for control, where $\pi(a|s, \theta) = P(A_t = a|S_t = s, \theta_t = \theta)$ describes action selection, without need of value function estimates. Actor-critic methods also approximate a parametric, typically state-value, function $\hat{v}(s, \omega)$ separating learning of action selection from predicting value estimates. Gradient updates are performed with respect to parameter vector $\theta$ and objective function $J(\theta)$. Definitions of $J(\theta)$ vary depending on the learning algorithm. In this work we explore the efficacy of three principle actor-critic algorithms; Proximal Policy Optimisation (PPO) [30], Advantage Actor-Critic (A2C) and its asynchronous counterpart (A3C) [24] in navigating our set of generated ladder logic programs. We discuss the merits and drawbacks of each approach for our setting in Section **??**.

## 3 Related Work

We briefly highlight key contributions within related literature, addressing the invariant finding problem for interlocking programs and contemporary RL strategies for environment exploration.

### 3.1 Invariant Finding

Elaborate on these

From software engineering techniques [9, **?**], to hybrid methods incorporating machine learning [13], researchers have proposed various approaches to invariant finding with varying degrees of success. IC3 [5] is one of the most successful approaches for model checking with invariants. IC3 makes use of of relatively simple SAT problems to incrementally constrain a state space towards only reachable

states. In this scenario, IC3 operates only at the Boolean level of the abstract state space, discovering inductive clauses over the abstraction predicates. It has been applied to verification of software [10] and in the context of hardware model checking [6]. Although, we note that this approach explores invariants for a state-space relative to a given property for verification. In our work, we aim to explore invariants that can be mined independent of the given property.

### 3.2 Exploration in Reinforcement Learning

Maintaining the desired balance between exploring unobserved state-action pairs for information maximisation and exploiting knowledge of the environment to further improve performance remains a challenge in RL research. Means of improving state exploration has seen particular interest in software or user testing communities. In [4], Bergdahl et al. apply vanilla PPO in an episodic 3D environment, incentivising state space coverage to automatically identify in-game bugs and exploits. Recently, Cao et al. [8] used a curiosity function based on an empirical count of state-action pairs to maximise traversal of state transitions for specific human-machine interfaces, using vanilla Q-learning with an $\epsilon$-greedy exploration. Similar applications in web application testing [36] simulate user actions to navigate site structures, recalling which state-action pairs the model is most 'uncertain'. State transitions with the greatest uncertainty are then prioritised for exploration when backtracking from state loops. Again, the authors use vanilla Q-learning and count-based reward scaling during Q-function updates [32]. Peake et al. [27] decompose exploration tasks over large, adaptive and partially observable environments into two sub-problems; adaptive exploration policy for region selection and separate policy for exploitation of an area of interest. Other works [2] incorporate recurrent networks [22] in policy design, using temporality to recall the performance of past actions and their subsequent consequences according to the reward function. We incorporate trends in the literature such as state-of-the-art learning algorithms sporting the best performance in research tasks and using environment reset logic to discourage early convergence. We detail this approach further in Section 5.

## 4 Mapping Formal Methods to Reinforcement Learning

Before any attempts toward invariant finding can be made, it is essential our model in the reinforcement learning setting captures the structure of model checking on ladder logic programs. In this section we introduce a faithful mapping that, given a ladder logic LTS, constructs an MDP model where reinforcement learning can be applied. Similarly, any invariant finding approach will require agents that maximise state space coverage. To explore this, we also introduce an approach for generation of state spaces with known metrics such as number of states and state space depth.

### 4.1 Ladder Logic Markov Decision Process

We now define the finite Markov Decision Process (MDP), or environment $\mathcal{E}$, used to represent the ladder logic program.

*Defn 4. Ladder Logic Markov Decision Process*: A Ladder Logic MDP $M(\psi) = (S, \mathcal{A}, P_a(s, s'), R_a(s, s'), \gamma)$ is a five tuple, where our observation space is the union of program inputs and ladder variables, and:

- $S = Val_C \cup Val_I$.
- $\mathcal{A} = Val_I$.
- $P_a(s, s') = P(s_{t+1} = s' | s_t, a_t)$.
- $R_a(s, s')$ is a reward function feeding a scalar signal back to the agent at each time step $t$.
- $\gamma \in [0, 1]$ is a discount scalar successively applied at each time step.

Here we note that any unique valuation of $Val_C$ under the dynamics of a ladder logic program constitutes a distinct state. Our action space, describes the set of ladder logic inputs used to compute new valuations after program execution. $P_a(s, s') = P(s_{t+1} = s' | s_t, a_t)$, describes the state transition function in terms of probabilities of observing $s_{t+1}$ given action $a_t$ is taken from state $s_t$. As here, more transitions are available than described by the ladder logic program, we use this probability distribution to ensure transitions match those defined by the ladder logic (essentially this will be 1 for transitions dictated by the ladder logic program and 0 for transitions that are not).

Subsequently as agents build a policy $\pi(s|a, \theta)$ according to $R_a(s, s')$ and state transitions observed under $P_a(s, s')$, the environment unfolds as a set of reachable states that mirror those of the ladder logic LTS.

Considering Figure 3, we observe differences in how agent action selection is represented compared to LTS transition labels. Where PRESSED and ACT_1 are ladder logic inputs, the indices of an agents action space refer to selecting one of the following valuations: [PRESSED=True, PRESSED=False, ACT_1=True, ACT_1=False]. One index may evaluate to True (1) while the remainder are False (0). Following Figure 3, an agent starting its training episode from initial state $S0$ has four available actions (illustrated in red) and three states reachable, $S1, S2, S4$ within the next time step. Selecting action $[0, 1, 0, 0]$ denotes setting PRESSED=False, observing a 'new' state $S1$, and receiving positive reward + 1, completing the time step. For ladder logic MDP with $N$ actions (LTS transition labels), there are at most $2N$ unique transitions $(s, a, s')$, thus the size of the action space from all states $s \in S$, is also $2N$.

Finally, our reward function and $\gamma$ can be tuned to modify the learning objective. Aiming to maximise state space coverage we implement a reward scheme which positively rewards novel observations over distinct episodes, deterring loop traversal through episode termination and negative rewards. Consider the example trajectory $\tau = (S0, [0, 1, 0, 0], +1, S1, [0, 0, 1, 0], +1, S4, [0, 1, 0, 0], +1, S5, [0, 1, 0, 0], -1, S5)$. Computing the expected return on the next episode, starting from $S0$ and using observed rewards from the latest trajectory, discounting factor $\gamma = 0.99$ applies accordingly; $G_t = 1 + 0.99(1) + 0.99^2(1) = 1 + 0.99 + 0.9801$. In Section 5 we discuss these point further.

### 4.2 Environment Generation

Given exhaustive search of large state spaces is often computationally intractable, we have generated a set of ladder logic programs where the number of reachable states and recurrence reachability diameter are known. This enables us to analyse the performance of our approach against well understood state spaces. Using existing models of ladder logic structures as a base template [17], we derive progressively larger programs by sequentially introducing additional rungs. This way a constrained yet predictable pattern of growth is devised. If $|S(\psi_i)|$ represents the number of reachable states for a program $\psi_i$, a subsequently generated program $\psi_{i+1}$ with one additional rung, has $2|S(\psi_i)|+1$ reachable states. Through a series of training runs on each environment we record the number of states observed by workers to gauge the overall state space coverage.

The following algorithm modifies the body of a pelican crossing ladder logic program given in Section 2.1. For every $i^{th}$ rung introduced to lengthen the base program, we define two additional variables, VAR_$i$ and ACT_$i$. This effectively doubles the existing state space, increases the size of each state observation and introduces as increments the size of $\mathcal{A}$ by 2.

```
procedure GENERATE LADDER(n_rungs, prog)
    cond ← (¬ PRESSED ∧ ¬CROSSING) ∧ ¬ REQ)
    rung ← ACT_1 ∧ cond
    coil ← VAR_1 ↔ rung
    i ← 1
    while i ≤ n_rungs do
        i ← i + 1
        new_rung ← ACT_[i] ∧ (rung)
        new_coil ← VAR_[i] ↔ (new_rung)
        append new_coil to prog
    end while
    return prog
end procedure
```

Fig. 2: Algorithm for automatic generation of ladder logic programs with controlled state space inflation.

Applying the above algorithm with $n\_rungs = 0$ produces the complete state space of Figure 3. For readability and illustration we have obfuscated three states from the original ladder logic program in Section 2.1, abstracted as the 'Pelican State Space'. Similarly, five states and their connected transitions are abstracted as the 'Extended State Space'. Again, dashed edges between states denote transitions induced by actions we have introduced while hard lined edges represent actions induced by valuations of the original input variable PRESSED. Transition labels from the LTS are highlighted in blue, while action selection according to the RL agent is shown in red. Results presented in Section 5 are

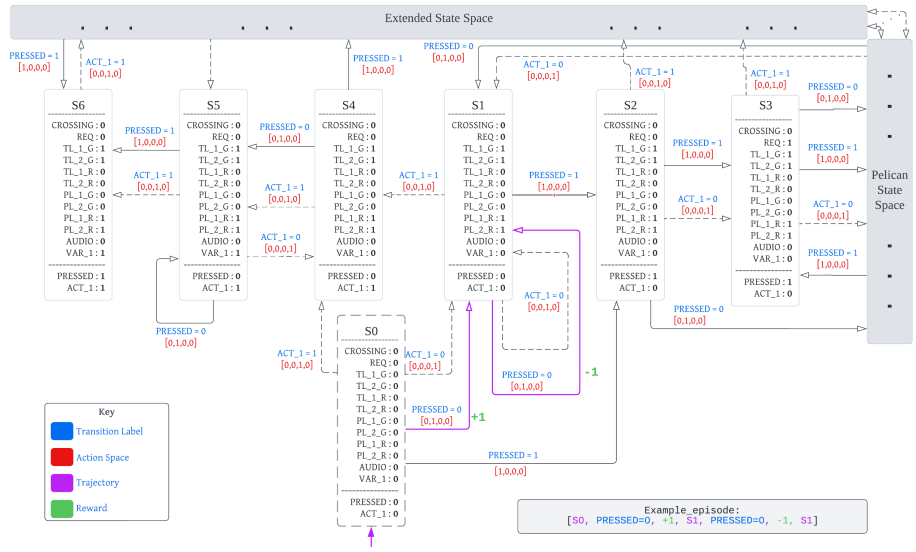


Fig. 3: Simplified state space representation of generated ladder logic program with one additional input variable ACT_1 and one output coil VAR_1.

based on a set of generated programs ranging from $2^{14}$ to $2^{50}$ states[1], referenced in Table **??**.

### 4.3 Agent Implementation

Here we introduce the network architecture used to train our RL agents. All algorithms presented in this work share the same architecture, adapting the input and output layers according to the environment. Our network input layer consists of $|Val_C \cup Val_I|$ nodes for given ladder logic formula $\psi$. Similarly, the output layer, responsible for mapping observations to the action space, comprises $|Val_I|$ nodes. All networks include a single fully connected hidden layer as this was empirically shown to produce the best performance. Learning rates for DQN, A2C, A3C and PPO algorithms were tuned individually according to the environment, falling within the range [0.0001, 0.001]. RMSProp was used to optimise the objective function.

[1] We note that interlocking programs generate much larger state spaces for model checking. However, work on abstractions by James et al. [18] have shown that the state space can be reduced, in many cases, to an approximate size of $2^{50}$

# 5 Results

## 5.1 State Space Coverage

## 5.2 Variable Correlation

## 5.3 Trajectory Clustering

# References


1. Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O., Zaremba, W.: Hindsight experience replay. Advances in Neural Information Processing Systems **30** (2017)
2. Apuroop, K.G.S., Le, A.V., Elara, M.R., Sheu, B.J.: Reinforcement learning-based complete area coverage path planning for a modified hTrihex robot. Sensors **21**(4) (2021). https://doi.org/10.3390/s21041067
3. Awedh, M., Somenzi, F.: Automatic invariant strengthening to prove properties in bounded model checking. In: ACM/IEEE Design Automation Conference. pp. 1073–1076 (2006). https://doi.org/10.1145/1146909.1147180
4. Bergdahl, J., Gordillo, C., Tollmar, K., Gisslén, L.: Augmenting automated game testing with deep reinforcement learning. In: IEEE Conference on Games. pp. 600–603 (2020). https://doi.org/10.1109/CoG47356.2020.9231552
5. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
6. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: Formal Methods in Computer Aided Design. pp. 173–180 (2007)
7. Cabodi, G., Nocco, S., Quer, S.: Strengthening model checking techniques with inductive invariants. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **28**(1), 154–158 (2009). https://doi.org/10.1109/TCAD.2008.2009147
8. Cao, Y., Zheng, Y., Lin, S.W., Liu, Y., Teo, Y.S., Toh, Y., Adiga, V.V.: Automatic HMI structure exploration via curiosity-based reinforcement learning. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 1151–1155 (2021). https://doi.org/10.1109/ASE51524.2021.9678703
9. Case, M.L., Mishchenko, A., Brayton, R.K.: Automated extraction of inductive invariants to aid model checking. In: Formal Methods in Computer Aided Design. pp. 165–172. IEEE (2007)
10. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
11. Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., Kavukcuoglu, K.: Impala: Scalable distributed deep-RL with importance weighted actor-learner architectures. arXiv preprint arXiv:1802.01561 (2018)
12. Fantechi, A., Fokkink, W., Morzenti, A.: Some trends in formal methods applications to railway signaling. Formal methods for industrial critical systems: A survey of applications pp. 61–84 (2012)

13. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. ACM Sigplan Notices **51**(1), 499–512 (2016)
14. Groote, J.F., van Vlijmen, S.F., Koorn, J.W.: The safety guaranteeing system at station Hoorn-Kersenboogerd. In: Proceedings of the Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'. pp. 57–68. IEEE (1995)
15. Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al.: Soft actor-critic algorithms and applications. arXiv preprint arXiv:1812.05905 (2018)
16. Houthooft, R., Chen, X., Duan, Y., Schulman, J., Turck, F.D., Abbeel, P.: VIME: Variational information maximizing exploration. arXiv preprint arXiv:1605.09674 (2017)
17. James, P., Lawrence, A., Moller, F., Roggenbach, M., Seisenberger, M., Setzer, A., Kanso, K., Chadwick, S.: Verification of solid state interlocking programs. In: International Conference on Software Engineering and Formal Methods. pp. 253–268. Springer (2013)
18. James, P., Roggenbach, M.: Automatically verifying railway interlockings using sat-based model checking. Electronic Communications of the EASST **35** (2011)
19. Kakade, S.M.: A natural policy gradient. Advances in Neural Information Processing Systems **14** (2001)
20. Kanso, K., Moller, F., Setzer, A.: Automated verification of signalling principles in railway interlocking systems. Electronic Notes in Theoretical Computer Science **250**(2), 19–31 (2009)
21. Malkauthekar, M.: Analysis of euclidean distance and manhattan distance measure in face recognition. In: International Conference on Computational Intelligence and Information Technology. pp. 503–507. IET (2013)
22. Medsker, L.R., Jain, L.: Recurrent neural networks. Design and Applications **5**, 64–67 (2001)
23. Melo, F.S., Meyn, S.P., Ribeiro, M.I.: An analysis of reinforcement learning with function approximation. In: Proceedings of the International Conference on Machine Learning. pp. 664–671 (2008)
24. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T.P., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. arXiv preprint arXiv:1602.01783 (2016)
25. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
26. Ostrovski, G., Bellemare, M.G., van den Oord, A., Munos, R.: Count-based exploration with neural density models. arXiv preprint arXiv:1703.01310 (2017)
27. Peake, A., McCalmon, J., Zhang, Y., Myers, D., Alqahtani, S., Pauca, P.: Deep reinforcement learning for adaptive exploration of unknown environments. In: International Conference on Unmanned Aircraft Systems. pp. 265–274 (2021). https://doi.org/10.1109/ICUAS51884.2021.9476756
28. Post, H., Sinz, C., Kaiser, A., Gorges, T.: Reducing false positives by combining abstract interpretation and bounded model checking. In: IEEE/ACM International Conference on Automated Software Engineering. pp. 188–197 (2008). https://doi.org/10.1109/ASE.2008.29
29. Schulman, J., Levine, S., Moritz, P., Jordan, M.I., Abbeel, P.: Trust region policy optimization. arXiv preprint arXiv:1502.05477 (2017)

30. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
31. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
32. Tang, H., Houthooft, R., Foote, D., Stooke, A., Xi Chen, O., Duan, Y., Schulman, J., DeTurck, F., Abbeel, P.: # exploration: A study of count-based exploration for deep reinforcement learning. Advances in Neural Information Processing Systems **30** (2017)
33. Tiegelkamp, M., John, K.H.: IEC 61131-3: Programming industrial automation systems. Springer (2010)
34. Vu, L.H., Haxthausen, A.E., Peleska, J.: Formal modeling and verification of interlocking systems featuring sequential release. In: International Workshop on Formal Techniques for Safety-Critical Systems. pp. 223–238. Springer (2014)
35. Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., de Freitas, N.: Sample efficient actor-critic with experience replay. arXiv preprint arXiv:1611.01224 (2017)
36. Zheng, Y., Liu, Y., Xie, X., Liu, Y., Ma, L., Hao, J., Liu, Y.: Automatic web testing using curiosity-driven reinforcement learning. In: IEEE/ACM International Conference on Software Engineering. pp. 423–435 (2021). https://doi.org/10.1109/ICSE43902.2021.00048