Todo list

Towards Reinforcement Learning of Invariants for Model Checking of Interlockings

Ben Lloyd-Roberts, Phillip James, Michael Edwards
Department of Computer Science, Swansea University UK
Email: {ben.lloyd-roberts, p.d.james, michael.edwards}@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 occurences 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. Proximal Policy Optimisation (PPO) demonstrates the most stable learning, particularly in large environments where the optimal behaviour function is most complex. Whereas, distributed, multi-agent algorithms, such as Asynchronous Advantage Actor-Critic (A3C) result in the greatest state coverage.**

1. Introduction

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 $s \in T$, such that $T \vdash F$.

The application of model checking in order to verify railway interlockings is well established within academia and is beginning to see real applications in industry. As early as 1995, Groote et al. [1] 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 [2], [3], [4]. This includes work by Linh et al. [] which explores the verification of interlockings 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 Rail 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 []. Inductive verification fails to consider whether system states which violate a given property are indeed reachable by the system from a defined initial configuration. These false positive often require manual inspection. One solution is to introduce so-called invariants to suppress false positives [5]. Invariants are properties that hold for subregions 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 [].

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. Finally 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 works in directing our approach to improve exploration and learn invariants from a dataset of state sequences generated by our RL agents.

2. Preliminaries

We now briefly discuss model checking of railway interlockings and reinforcement learning. For further details we refer the reader to [6], [7] and [8] 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 [**?**] 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 [**?**]. 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.

A interlocking executes such a program from top-to-bottom over and over, indefinitely.

More formally, following [**?**] 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 ψ 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 ψ_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.)

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. [6] and James et al. [7]. 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 [**?**], the semantics of a ladder logic program in terms of labelled transition systems.

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

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

be the sets of valuations for input and output variables.

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

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

where μ'_C is computed as follows: the value of each variable c_i is computed using the ith rung of the ladder, ψ_i, using the valuations μ_C and μ_I from the last cycle and the current valuations restricted to those evaluated be fore the current variable. We refer the reader to [**?**] for full details.

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

- $Val_C = \{\mu_C | \mu_C : C \rightarrow \{0,1\}\}$ is a set of states.
- $Val_I = \{\mu_I | \mu_I : I \rightarrow \{0,1\}\}$ is a set of transition labels.
- $\rightarrow \subseteq Val_C \times Val_I \times Val_c$ 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 2, which, within one Figure, illustrates multiple models of a simple ladder logic program for controlling a pelican crossing[1] as considered by James et al []. 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 example 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) [9].

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.
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t, a_t)$, 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.

1. Here we omit the ladder logic code and refer the reader to []

- $\gamma \in [0, 1]$ 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. 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 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 being in state $S_t = s$, having taken action $A_t = a$ during the present time step. Observed returns depend on the action(s) 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 []. In practice to scale with complex environments these functions are parametric and approximated with gradient-based methods, such as stochastic gradient ascent.

Review: Added notation (parameter vector wasn't introduced) and more detail on algorithms

Determining the reachable state space for LLPs 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 functions and the policy given our reward function. Initially basic Deep Q-Networks (DQN) [10] were used to test the training framework. Among this family of approximate reinforcement learning algorithms policy gradient methods [11]. Such algorithms utilise parametrised policies for control, where $\pi(a|s, \theta) = Pr(A_t = a|S_t = s, \theta_t = \theta)$ describes action selection, without need of value function estimates. Policy updates are performed with respect to parameter vector θ and objective function $J(\theta)$. Definitions of $J(\theta)$ vary depending on specific algorithms. Actor-critic methods also approximate a parametric, typically state-value, function $\hat{v}(s, \omega)$ separating learning of action selection from predicting value estimates. In this work we explore the efficacy of three principle actor-critic algorithms; Proximal Policy Optimisation (PPO) [12], Advantage Actor-Critic (A2C) and its asynchronous counterpart (A3C) [8] in navigating our set of generated LLPs. We discuss the merits and drawbacks of each approach for our setting in Section 5.1.

3. Related Work

Here 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

IC3 [13] 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 [14] and indeed, in the context of hardware model checking [15]. Although we note that this approach contains the 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. From software engineering techniques [16], [17] to hybrid methods incorporating machine learning [18], researchers have proposed various approaches to invariant finding with varying degrees of success.

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 [27], 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. [28] 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 ϵ-greedy exploration. Similar applications in web application testing [29] 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, authors use vanilla Q-learning and count-based reward scaling during Q-function updates [30]. Authors of [31] 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 [32] incorporate recurrent networks [33] in policy design, using temporality to recall the performance of past actions and their subsequent consequences according to the reward function. ACER algorithm used to make the A3C algorithm off-policy and constrain parameter updates according to a KL divergence measure between the current and previous policy. In robotics research [34], Apuroop et al. apply state-of-the-art learning algorithms to train hTrihex robot navigation in procedurally generated environments. PPO found to have near human level performance, recalling which states are 'expensive' to reach and minimising the frequency of their 'rediscovery'. Works introducing novel algorithms such as [35] have illustrated the efficacy of RL in learning combinatorial algorithms over large graph structures comprising billions of nodes. Others have employed distributing learning [36] to accelerate performance gains and maintain state novelty when exploring unfamiliar environments [37], [38], [39]. In this work we incorporate trends in the literature such as state-of-the-art learning algorithms sporting the best performance in research tasks and using selective environment reset logic to discourage early convergence to subregions. We detail this approach further in Section 5.

4. Mapping Formal Methods to Reinforcement Learning

Review: Added some more intro

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 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. Hence, agents which maximise state space coverage in our MDP model are capable of learning properties which can be used within model checking.

4.1. Ladder Logic Markov Decision Process

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

- $S = Val_C \cup Val_I$.
- $\mathcal{A} = Val_I$.
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t, a_t)$
- $R_a(s, s')$ 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 LLP constitutes a distinct state. Our action space, describes the set of ladder logic inputs used to compute new valuations after program execution. $P_a(s, s') = Pr(s_{t+1} = s' | s_t, a_t)$, 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.

Review: Does this work?

Considering Figure 1, 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 1, 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 γ 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 τ = (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 where $\gamma = 0.99$ applies accordingly; $G_t = 1 + 0.99(1) + 0.99^2(1) + 0.99^3(1)$. In Section 5 we discuss these point further.

4.2. Environment Generation

use ψ for the ladder logic program throughout, it mathes section 2.

Given exhaustive search of large state spaces is often computationally intractable, we have generated a set of ladder 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 [7], 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 ψ_i, a subsequently generated program ψ_{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 the pelican crossing ladder program referenced in [7]

explain the psuedocode in a short paragraph

procedure GENERATE LADDER(n_rungs, prog)
  cond ← (¬ PRESSED ∧ ¬CROSSING) ∧ ¬ REQ
  rung ← ACT_1 ∧ cond
  coil ← VAR_1 ≡ rung
  $i \leftarrow 1$
  while $i \leq$ n_rungs **do**
    $i \leftarrow i + 1$
    new_rung ← ACT_[i] ∧ (rung)
    new_coil ← VAR_[i] ≡ new_rung
    append new_coil **to ladder logic program** *prog*
  end while
end procedure

add a paragraph discussing Figure 1 in this contest, Figure 1 has been obtained by applying the above algorithm to the ladder logic program presented by James et al []... it shows...

5. Results

remove LLPs and call them ladder programs instead :)

We now present a set of results from applying our approach to a series of generated ladder programs, modelled as learning environments. The following section is divided into three parts, first discussing the merits and drawbacks of the respective learning algorithms. Second, we present results from applying the multi agent A3C algorithm to our full set of generated ladder programs, shown in Table 1. Finally we discuss performance differences between single agent implementations of A2C and PPO, as they exhibited divergent learning objectives.

5.1. Algorithmic Trends

Make this coherent

Tried DQN on smaller environments to test the viability of RL in our problem formulation and it solved them (with respect to our reward objective) within an acceptable time frame. DQN doesn't scale well and is too sensitive to hyperparameter tuning. buffer size environment dependent, training delay and gradient step size contingent on complexity of learnable function, which we'd expect to increase with environment size. Algorithm is also off-policy - trains from experiences generated by old (presumably less optimal) policies. Exploration is governed explicitly by the epsilon-greedy strategy and guaranteed to anneal toward a minimum threshold over a significant number timesteps - this could happen with large regions of an environment unobserved. Methods exist to improve the sample efficiency and learning stability, such as Prioritised experience replay or duelling DQN.

We also try two Actor critic methods, Advantage actor-critic (A2C) and its asynchronous counterpart (A3C) to improve sample efficiency in distributing environment interaction among several CPU cores, or 'workers'. Having several workers and an on-policy learning algorithm removes need of replay memory and training delay to accumulate sufficient experience. Actor and critic networks, approximating the behaviour policy and value function, provide better convergence guarantees at the cost of some additional complexity. Paired with randomised reset logic both algorithms accumulate experience faster than Vanilla DQN, achieving good coverage on a range of medium to large state spaces. They are susceptible to parameter updates pushing the model into an unfamiliar region of policy space from which subsequent updates are unable to recover, potentially triggering model collapse.

Diagram trajectory missing terminal state after reward

We then move to trust region methods which constrain the magnitude of gradient updates within some clip range or according to KL-divergence between the current and most recent policy parameters, making the model more resilient to collapse.

5.2. Asynchronous Exploration

Check for CopyPasta erros

Preliminary results applying our approach to a number of generated programs are outlined in Table 1. 'Actions' referenced in the third column refer to the number of possible assignments over input variables in each LLP, from every state. 'K', refers to the greatest number of steps taken before repeating observations in the environment, across all workers.

Coverage metrics are expectedly maximised for environments with a small number of reachable states with acceptable levels of coverage for programs with a theoretical state space up to 2^{40}. Interestingly, we observed longer training durations occasionally increased coverage beyond a certain threshold. It is possible workers learn an optimal search strategy within a subregion of the state space. Additionally, performance in terms of max k and states reached increased by approx. 5% when decreasing the total number of episodes from 3×10^5 to 1.5×10^5 episodes. This may be a product of random episode initialisation spawning workers in more desirable states where stochastic action sampling happened to lead to unfamiliar subregions of the environment.

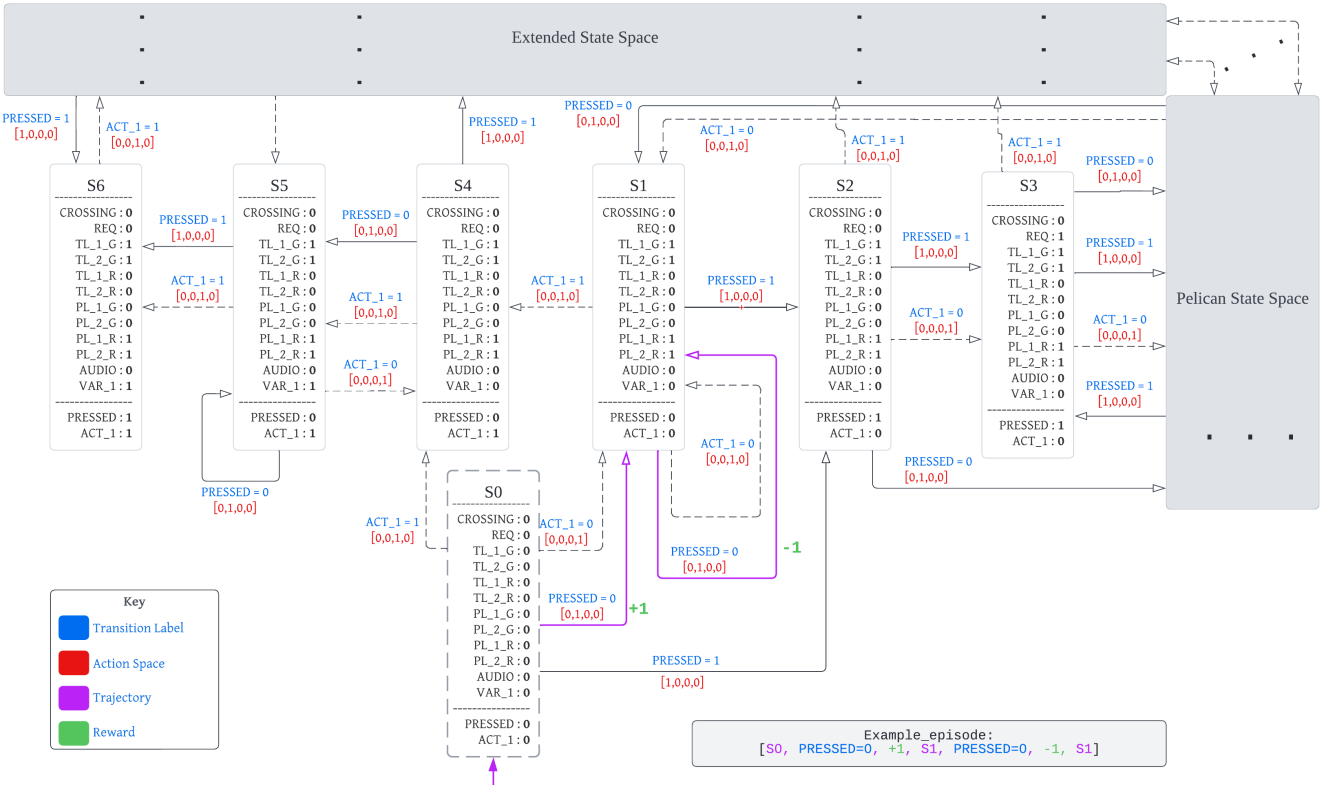


Figure 1: Simplified state space representation of generated ladder logic program with one additional input variable ACT_1 and one output coil VAR_1

TABLE 1: A3C Coverage Metrics

Environment			Agent	
States (Theoretical)	States (Reachable)	Actions	K-bound	Coverage
2^{14}	15	2	14	100.0
2^{16}	31	4	28	100.0
2^{18}	63	6	48	100.0
2^{20}	127	8	33	100.0
2^{22}	255	10	76	100.0
2^{24}	511	12	49	100.0
2^{26}	1023	14	306	100.0
2^{28}	2047	16	538	100.0
2^{30}	4095	18	1418	99.731
2^{32}	8191	20	1712	96.532
2^{34}	16383	22	1498	95.550
2^{36}	32767	24	2879	84.694
2^{38}	65535	26	1969	89.071
2^{40}	131071	28	2692	82.884
2^{42}	262143	30	1406	76.033
2^{44}	524287	32	1782	62.137
2^{46}	1048575	34	1593	64.053
2^{48}	2097151	36	1598	57.547
2^{50}	4194303	38	2566	41.483

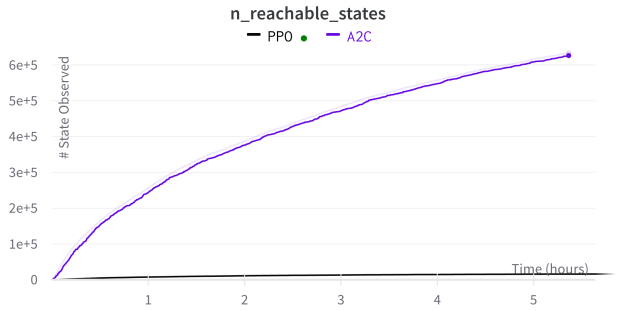
Performance plots illustrating the cumulative reward which failed to maximise coverage often increased linearly before collapsing to some suboptimal reward. This may be due to tendencies for large network updates to shift the network gradients into a bad local minima, from which performance does not recover within the allotted training duration. The on-policy nature of actor critic means trajectories generated via an old policy are no longer sampled during minibatch updates for the current policy, thus biasing behaviour to the most recent model updates and introducing sample inefficiency. Adding experience replay [40] may help avoid this in future applications

Given the A3C algorithm requires workers to asynchronously update their shared network every $T_{\max}$ steps or on episode termination, larger values for $T_{\max}$ consolidate more information regarding worker trajectories before applying gradient updates to their local network. We found the most significant improvements to performance in terms of coverage metrics and increasing the k bound when introducing workers to larger environments, was lower update frequencies and random start state initialisation. Prior to these adjustments workers, irrespective of their number, seldom covered 80% of most smaller environments. Similarly, for the largest environment with 2^{50} states, coverage improved from 3.2% to 41.48%

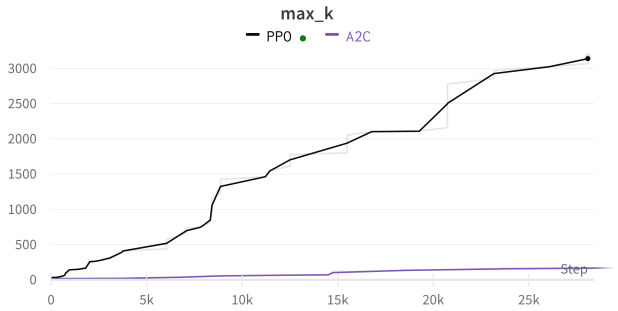
5.3. Trust Regions and Reward Shaping

Performance plots for PPO on pelican_19

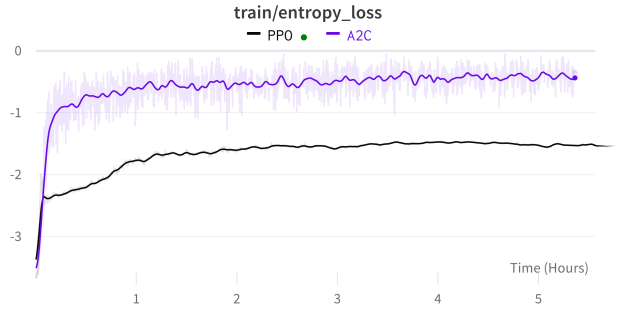
Make coherent



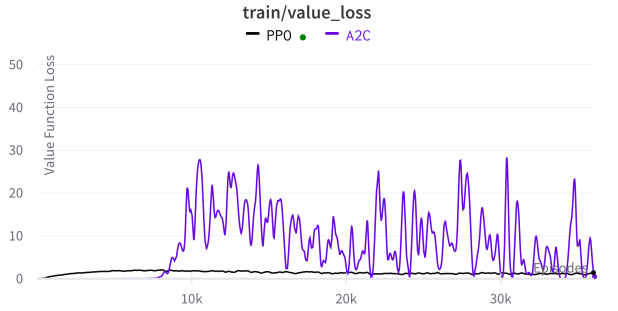
(a) Case I



(b) Case II



(c) Case I



(d) Case II

Figure 2: Simulation results for the network.

- Advantage actor critic methods seem to perform better in terms of coverage. Distributed nature of algorithm and no update clipping may allow for greater exploration.
- PPO optimises better to increasing the k-bound for the explored subregion while slowly increasing state coverage.
- A2C increases state coverage significantly faster than PPO but struggles to learn optimal loop free paths within its discovered subregion.
- Value function loss is more stable for PPO compared to A2C. This measures the TD error between the current value function and actual observed returns.
- A2C entropy loss converges toward 0 sooner than PPO, a possible indication the policy learned under A2C experiences less uncertainty when making predictions. Could mean PPO policy maintains it knows very little about the environment but still maximises rewards better than its A2C counterpart.

6. Conclusion & Future Work

Summarise paper, dream big

In this work we have taken 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. In addition, we have applied asynchronous and trust region deep reinforcement learning methods to programtically generated state spaces and analysed their ability in terms of state coverage and state space depth. Our findings highlight that RL approaches can be succesfully rewarded to explore a large percentage of a given state space in terms of state coverage, however that incentivising depth based exploration is more challenging. As any machine learning approach to finding invariants will likely need to explore such a state space these results show the credability of such an appraoch. In our subsequent works we envisage the current learning framework to serve as a means of dataset generation, from which patterns or sequences of can be mined from agent trajectories.

In light of our findings, we also aim to improve several aspects of our approach, predominantly concerning learning stability, sample efficiency and training speed. Experience replay for distributed learning may improve on-policy bias and sample efficiency. The low dimensionality of our state spaces representation may allow us to introduce count-based exploration models to dampen the reward issued for states repeatedly observed [37]. Intrinsic motivation has also illustrated successes in environment exploration [41].

In this paper we have applied a basic asynchronous and trust region deep reinforcement learning methods to maximise program state coverage, motivated by a reward function of state novelty.

In light of our findings, we aim to improve several

aspects of our approach, predominantly concerning learning stability, sample efficiency and training speed. Experience replay for distributed learning may improve on-policy bias and sample efficiency.

The low dimensionality of our state spaces representation may allow us to introduce count-based exploration models to dampen the reward issued for states repeatedly observed [37]. Intrinsic motivation has also illustrated successes in environment exploration [41].

Applying IMPALA [42] to improve both sample efficiency over A3C and robustness to network architectures and hyperparameter adjustments. The adoption of a Long Short-Term Memory model (LSTM) also improves performance given GPU acceleration is maximised on larger batch updates.

For invariant finding this work has indicated the viability of reinforcement learning algorithms in increasing state space coverage. In our subsequent works we envisage the current learning framework to serve as a means of dataset generation, from which patterns or sequences of can be mined from agent trajectories.

Acknowledgments

We thank Tom Werner and Andrew Lawrence at Siemens Mobility UK & EPSRC for their support in these works.

References

[1] J. F. Groote, S. F. van Vlijmen, and J. W. Koorn, "The safety guaranteeing system at station hoorn-kersenboogerd," in *COMPASS'95 Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security'*. IEEE, 1995, pp. 57–68.

[2] A. Fantechi, W. Fokkink, and A. Morzenti, "Some trends in formal methods applications to railway signaling," *Formal methods for industrial critical systems: A survey of applications*, pp. 61–84, 2012.

[3] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi, "Model checking interlocking control tables," in *FORMS/FORMAT 2010*. Springer, 2011, pp. 107–115.

[4] A. E. Haxthausen, M. L. Bliguet, and A. A. Kjær, "Modelling and verification of relay interlocking systems," in *Monterey Workshop*. Springer, 2008, pp. 141–153.

[5] M. Awedh and F. Somenzi, "Automatic invariant strengthening to prove properties in bounded model checking," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 1073–1076.

[6] K. Kanso, F. Moller, and A. Setzer, "Automated verification of signalling principles in railway interlocking systems," *Electronic Notes in Theoretical Computer Science*, vol. 250, no. 2, pp. 19–31, 2009.

[7] P. James, A. Lawrence, F. Moller, M. Roggenbach, M. Seisenberger, A. Setzer, K. Kanso, and S. Chadwick, "Verification of solid state interlocking programs," in *International Conference on Software Engineering and Formal Methods*. Springer, 2013, pp. 253–268.

[8] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[9] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[11] S. M. Kakade, "A natural policy gradient," *Advances in neural information processing systems*, vol. 14, 2001.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[13] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87.

[14] A. Cimatti and A. Griggio, "Software model checking via ic3," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 277–293.

[15] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design (FMCAD'07)*, 2007, pp. 173–180.

[16] M. L. Case, A. Mishchenko, and R. K. Brayton, "Automated extraction of inductive invariants to aid model checking," in *Formal Methods in Computer Aided Design (FMCAD'07)*. IEEE, 2007, pp. 165–172.

[17] S. Bensalem, Y. Lakhnech, and H. Saidi, "Powerful techniques for the automatic generation of invariants," in *International Conference on Computer Aided Verification*. Springer, 1996, pp. 323–335.

[18] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," *ACM Sigplan Notices*, vol. 51, no. 1, pp. 499–512, 2016.

[19] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[20] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[21] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[22] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 3389–3396.

[23] M. Bloesch, J. Humplik, V. Patraucean, R. Hafner, T. Haarnoja, A. Byravan, N. Y. Siegel, S. Tunyasuvunakool, F. Casarini, N. Batchelor *et al.*, "Towards real robot learning in the wild: A case study in bipedal locomotion," in *Conference on Robot Learning*. PMLR, 2022, pp. 1502–1511.

[24] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," *Computers & Operations Research*, vol. 134, p. 105400, 2021.

[25] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[26] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[27] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén, "Augmenting automated game testing with deep reinforcement learning," in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 600–603.

[28] Y. Cao, Y. Zheng, S.-W. Lin, Y. Liu, Y. S. Teo, Y. Toh, and V. V. Adiga, "Automatic hmi structure exploration via curiosity-based reinforcement learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1151–1155.

[29] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 423–435.

[30] H. Tang, R. Houthooft, D. Foote, A. Stooke, O. Xi Chen, Y. Duan, J. Schulman, F. DeTurck, and P. Abbeel, "# exploration: A study of count-based exploration for deep reinforcement learning," *Advances in neural information processing systems*, vol. 30, 2017.

[31] A. Peake, J. McCalmon, Y. Zhang, D. Myers, S. Alqahtani, and P. Pauca, "Deep reinforcement learning for adaptive exploration of unknown environments," in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2021, pp. 265–274.

[32] K. G. S. Apuroop, A. V. Le, M. R. Elara, and B. J. Sheu, "Reinforcement learning-based complete area coverage path planning for a modified htrihex robot," *Sensors*, vol. 21, no. 4, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/4/1067

[33] L. R. Medsker and L. Jain, "Recurrent neural networks," *Design and Applications*, vol. 5, pp. 64–67, 2001.

[34] D. I. Koutras, A. C. Kapoutsis, A. A. Amanatiadis, and E. B. Kosmatopoulos, "Marsexplorer: Exploration of unknown terrains via deep reinforcement learning and procedurally generated environments," *Electronics*, vol. 10, no. 22, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/22/2751

[35] S. Manchanda, A. Mittal, A. Dhawan, S. Medya, S. Ranu, and A. Singh, "Learning heuristics over large graphs via deep reinforcement learning," *arXiv preprint arXiv:1903.03332*, 2019.

[36] M. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli *et al.*, "Acme: A research framework for distributed reinforcement learning," *arXiv preprint arXiv:2006.00979*, 2020.

[37] G. Ostrovski, M. G. Bellemare, A. van den Oord, and R. Munos, "Count-based exploration with neural density models," 2017.

[38] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

[39] C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslén, "Improving playtesting coverage via curiosity driven reinforcement learning agents," *arXiv preprint arXiv:2103.13798*, 2021.

[40] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," 2017.

[41] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel, "Vime: Variational information maximizing exploration," 2017.

[42] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," 2018.