Mining Invariants from 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 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. [12] 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 [9, **?**, **?**]. This includes work by Linh et al. [26] 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 [22]. 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 [2]. 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 [6].

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 [16, **?**] and [20, 23, 19] 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[25] 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. [13]. 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 [13] 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.)

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 [13]. 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. [16] and James et al. [13]. 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 [13], the semantics of a ladder logic program in terms of labelled transition systems.

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

Fig. 1: Ladder logic program for pelican crossing.

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

$$\begin{aligned}
Val_I &= \{\mu_I \mid \mu_I : I \to \{0,1\}\} = \{0,1\}^I \\
Val_C &= \{\mu_C \mid \mu_C : C \to \{0,1\}\} = \{0,1\}^C
\end{aligned}$$

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.

$$\begin{aligned}
&[\psi] : Val_I \times Val_C \to Val_C \\
&[\psi](\mu_I, \mu_C) = \mu'_C
\end{aligned}$$

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 [13] for full details.

Definition 1 (Ladder Logic Labelled Transition System). *We define the labelled transition system $LTS(\psi)$ for a ladder logic formula ψ as the tuple $(Val_C, Val_I, \to, Val_0)$ 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 $s_0 \in Val_0$.

2.3 Reinforcement Learning and MDPs

Rephrase

Reinforcement Learning (RL) is a machine learning paradigm typically used to solve sequential decision making problems by modelling the optimal control of some incompletely-known Markov Decision Process (MDP) [24].

Definition 2 (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 reinforcement learning the MDP serves as our *environment* $\mathcal{E}$, where through *episodic* or *continuous* simulation, software agents sample their action space, observe state transitions and learn to optimise some objective based on a differential reward signal issued over discrete time steps. Continuous settings allow agents to interact with their environment for an undefined number of time steps, until some termination criterion is met such as reaching a reward threshold or state deadlock. Alternatively tasks may be episodic, where training occurs over T_{Max} sequences of *episodes*. 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 finite number of time steps, constituting a single episode or continuous training run. Ensuing algorithmic descriptions or problem formulation in this work refer to the episodic case. Here, h denotes the *horizon*; a time step beyond which rewards are no longer influence an agent's prediction. Rewards observed from some time step t up to a 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 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.

Prediction and *control* refer to two principle challenges of RL. The first challenge is in approximating a value function which accurately estimates a quantitative benefit of being in that state. State values $v(s)$ or state-action values $q(s, a)$, describe this estimate. Values are differentiable and updated based on an empirical average, known as the *expectation* taken over observed returns, $\mathbb{E}[G_t|S_t = s, A_t = a]$. Thus computing expectation over return G_t depends on being in state $S_t = s$, having taken action $A_t = a$ during the most recent time step. Observed returns depend on the actions sampled at discrete time steps.

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. Optimal value functions and optimal policies are reciprocal in that converging to an optimal value function will converge to an optimal policy [18]. 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 industrial ladder logic programs is often intractable, making the complete MDP model unknown to us without exhaustively computing all state-action pairs. Consequently we trial several gradient-based learning algorithms using deep neural networks to approximate both value function and behaviour policy, subject to our reward objective.

Check if we ever refer to the reward objective before this

Policy gradient methods [15] 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 predicting and learning action selection and value estimates independently. In practice however, both actor and critic networks share learnable parameters yet produce different predictions. Gradient updates are performed with respect to parameter vector θ 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) [23], Advantage Actor-Critic (A2C) and its asynchronous counterpart (A3C) [19] 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 contemporary environment exploration strategies, the invariant finding problem for interlocking programs and existing invariant mining techniques.

3.1 Invariant Finding

Elaborate on these citations

From software engineering techniques [7, **?**], to hybrid methods incorporating machine learning [10], researchers have proposed various approaches to invariant finding with varying degrees of success. IC3 [4] 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 [8] and in the context of hardware model checking [5]. 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

Techniques on mining and shorten exploration section / focus on mining

Maintaining the desired balance between explorative traversal, for information maximisation, and exploiting existing knowledge of the environment to further improve performance remains a challenge in RL. Means of improving state exploration has seen particular interest in software or user testing communities. In [3], Bergdahl et al. apply vanilla PPO in an episodic 3D environment, incentivising state space coverage to automatically identify in-game bugs and exploits. Peake et al. [21] 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 [1] incorporate recurrent networks [17] 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 Mining Invariants

Before any attempts toward invariant finding can be made, it is essential learned properties in an RL environment can be used meaningfully in model checking ladder logic programs. Similarly, any invariant finding approach will require observations from a sufficient proportion of reachable states. In this section we introduce a mapping that, given a ladder logic LTS, constructs an MDP where reinforcement learning can be applied to maximise state space coverage and mine invariants from such observations.

4.1 Ladder Logic Markov Decision Process

Definition 3 (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 reflect those of the ladder logic LTS.

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 $\tau = (S0, [0,1,0,0], +1, S1, [0,0,1,0], +1, S4, [0,1,0,0], +1, S5, [0,1,0,0], -1, S5)$. 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 Data Collection

Training episodes initialise environments with an initial state, either with all variables set to 0 or, to incentivise exploration, we randomly initialise workers to some previously visited state, as demonstrated in [11]. Agents are initialised with separate environment instances to accumulate experience independently. A global set of unique observations is asynchronously updated by workers to compile shared experiences across separate CPU threads. Early stopping criteria terminates training if a predetermined number of consecutive model updates do not improve state coverage or increase reward. On episode termination an agent is left with its most recent trajectory, a sequence of states, actions and rewards observed during the latest episode. Consider figure **??**. The left-hand diagram illustrates the base pelican crossing state space from which the trajectories $\tau_1, ..., \tau_n$ are produced. During the training process we concatenate sequences of trajectories collected across separate episodes, forming an $N \times M$ matrix A_τ, where N refers to total observations collected during training and M the number of program variables $|Val_I \cup Val_C|$, comprising any given observation. A_τ serves as our dataset from which invariants can be mined.

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 in-

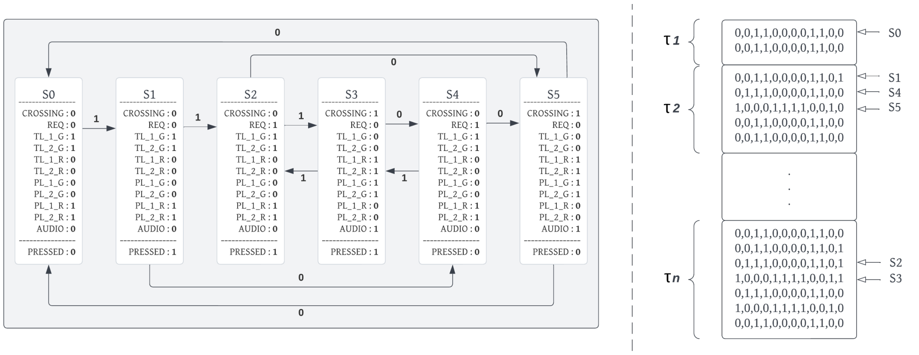


Fig. 2: Illustration of trajectory stacking from observation made during exploration of pelican crossing ladder logic program state space. Edge labels denote change in valuation of input variable PRESSED.

put and output layers according to the environment. Our network input layer consists of $|Val_C \cup Val_I|$ nodes for given ladder logic formula ψ. 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.

5 Results

5.1 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 depth are known. This enables us to analyse the performance of our approach on state spaces with known ground truths. Using existing models of ladder logic structures as a base environment template [13], we derive progressively larger programs by sequentially introducing additional rungs. This way a predictable growth pattern is devised. Where $|S(\psi_i)|$, 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. During interaction with each environment we record the number of states observed by agents to gauge the overall state space coverage.

Add fig: pelican state space highlighting gated condition holding for S0, branching off to an extended space

Results presented in Section 5 are based on a set of generated programs ranging from 2^{14} to 2^{50} states[1], referenced in Table 1.

5.2 State Space Coverage

Preliminary results applying the A3C algorithm to a number of generated programs are outlined in Table 1. Training was distributed among 32 CPU worker threads. 'Actions' referenced in the third column refer to the number of possible assignments over input variables in each ladder program, from every state. Depth, the first agent column, 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 size up to 2^{40}. Interestingly, we observed longer training durations without early stopping 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 depth 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.

Performance in terms of 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 on policy memory strategies[27] 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%

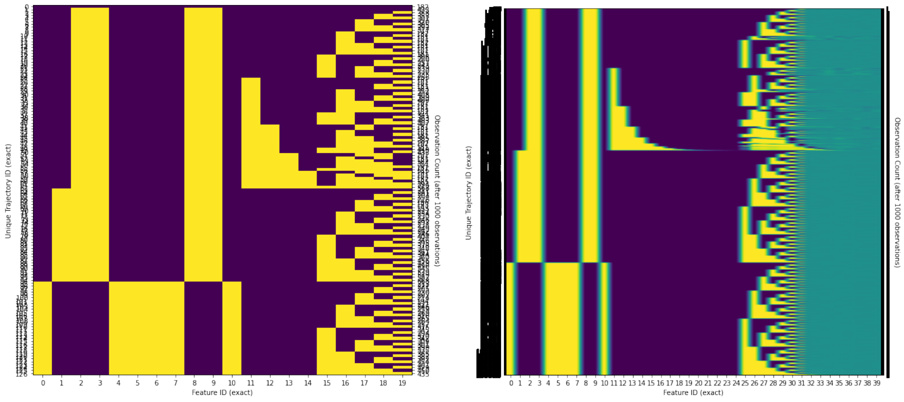
[1] We note that interlocking programs generate much larger state spaces for model checking. However, work on abstractions by James et al. [14] have shown that the state space can be reduced, in many cases, to an approximate size of 2^{50}

Table 1: A3C Coverage Metrics

Environment			Agent	
States (Theoretical)	States (Reachable)	Actions	Depth	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

5.3 State Properties

Recall the data generation process illustrated in Figure **??**. Initially, our observation data set A_τ consists of all stacked agent trajectories, from which invariant properties are to be mined. Invariants may be manifest temporally, as patterns across state sequences, such as whenever we observe X in S_t, we always observe Y in S_{t+1}. Stacking all agent trajectories to form our dataset preserves temporal properties, recorded during many training episodes, for future sequence mining. Similarly, we may infer certain program properties by through a static representation in terms of unique, reachable states. Figure **??** illustrates this for all 127 reachable states given a ladder logic program consisting 2^{20} theoretic states. Each row represents a state while each column corresponds to a program variable. Each cell in Figure **??** represents a value (purple : 0, yellow : 1) for the respective variable under a given state. Recall our base pelican crossing program (Figure **??**) concerned variables $CROSSING, ..., AUDIO$. We see from the set of reachable states in Figure **??** a candidate invariant disallowing both traffic lights and pedestrian lights to simultaneously display green aspects, i.e $PL\_1\_G \wedge PL\_2\_G \equiv \neg(TL\_1\_G \wedge TL\_2\_G)$, which is never explicitly stated in ladder logic. Note this property holds for all generated programs, irrespective of size, as shown in Figure **??**(b). We envisage applying such visualisations to larger interlocking programs may allow us to infer properties concerning those ladder logic programs.



(a)

(b)

5.4 Variable Correlation

Next we discuss the use of phi coefficients, or Matthew's Correlation Coefficient (MCC), as a basic statistical measure of association between binary variables comprising arbitrary ladder logic programs. For two binary variables, (x, y), we compute correlation coefficient $\phi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet} - n_{\bullet 1}n_{\bullet 0}}}$, where n_{ij} refers to a count of observed valuations:

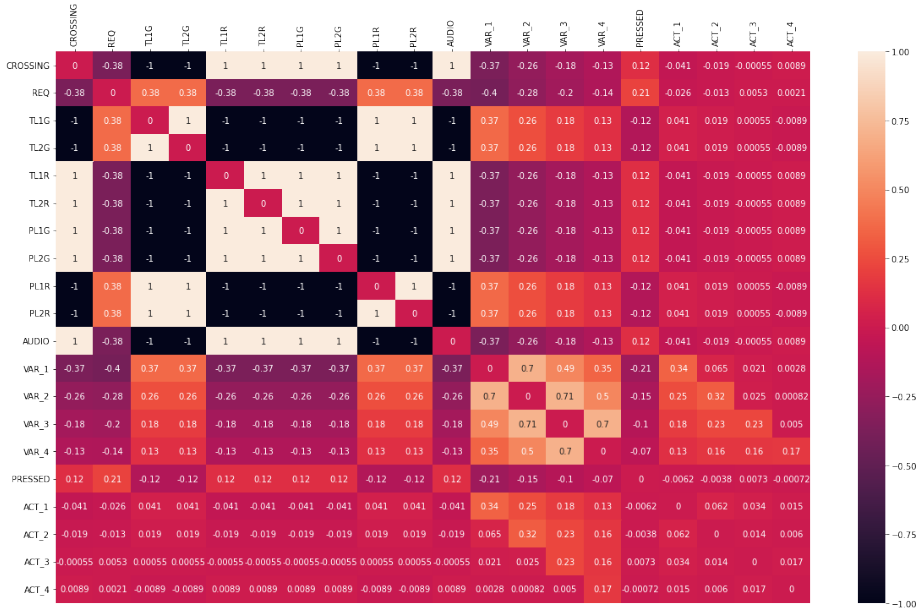
	$y = 1$	$y = 0$	total
$x = 1$	n_{11}	n_{10}	$n_{1\bullet}$
$x = 0$	n_{01}	n_{00}	$n_{0\bullet}$
total	$n_{\bullet 1}$	$n_{\bullet 0}$	n

In other words the relation between two binary variables depends on their variability with respect to one another. If we observe all possible valuations of our program variables, i.e observe all unique reachable states, then we have sufficient samples to accurately compute this association. We populate a correlation matrix, similar to the variance-covariance matrix, for all pairs of binary variables comprising a ladder program with four coils $VAR\_(1, ..., 4)$, four program inputs, $ACT\_(1, ..., 4)$,127 reachable states and 2^{20} theoretical. Values range $[-1, 1]$ where 1 is complete correlation and -1 is an inverse relation.

Invariant shows no two sets of lights may simultaneously show a green aspect via inverse / positive relations (refer to the latches).

Property holds for programs with larger state space, more coils and program inputs (127 states, 2^{20} theoretical), correlation confirms the observed pattern in state image. See program structure, i.e variable assignment rules, in positive and inverse relations. Samples used to compute phi are observed through unique

Fig. 3: Correlation matrix for ladder logic program with four additional synthetically generated rungs, 127 reachable states and 2^{20} theoretical states.



	CROSSING	REQ	TL1G	TL2G	TL1R	TL2R	PL1G	PL2G	PL1R	PL2R	AUDIO	VAR_1	VAR_2	VAR_3	VAR_4	PRESSED	ACT_1	ACT_2	ACT_3	ACT_4
CROSSING	0	-0.38	-1	-1	1	1	1	1	-1	-1	1	-0.37	-0.26	-0.18	-0.13	0.12	-0.041	-0.019	-0.00055	0.0089
REQ	-0.38	0	0.38	0.38	-0.38	-0.38	-0.38	-0.38	0.38	0.38	-0.38	-0.4	-0.28	-0.2	-0.14	0.21	-0.026	-0.013	0.0053	0.0021
TL1G	-1	0.38	0	1	-1	-1	-1	-1	1	1	-1	0.37	0.26	0.18	0.13	-0.12	0.041	0.019	0.00055	-0.0089
TL2G	-1	0.38	1	0	-1	-1	-1	-1	1	1	-1	0.37	0.26	0.18	0.13	-0.12	0.041	0.019	0.00055	-0.0089
TL1R	1	-0.38	-1	-1	0	1	1	1	-1	-1	1	-0.37	-0.26	-0.18	-0.13	0.12	-0.041	-0.019	-0.00055	0.0089
TL2R	1	-0.38	-1	-1	1	0	1	1	-1	-1	1	-0.37	-0.26	-0.18	-0.13	0.12	-0.041	-0.019	-0.00055	0.0089
PL1G	1	-0.38	-1	-1	1	1	0	1	-1	-1	1	-0.37	-0.26	-0.18	-0.13	0.12	-0.041	-0.019	-0.00055	0.0089
PL2G	1	-0.38	-1	-1	1	1	1	0	-1	-1	1	-0.37	-0.26	-0.18	-0.13	0.12	-0.041	-0.019	-0.00055	0.0089
PL1R	-1	0.38	1	1	-1	-1	-1	-1	0	1	-1	0.37	0.26	0.18	0.13	-0.12	0.041	0.019	0.00055	-0.0089
PL2R	-1	0.38	1	1	-1	-1	-1	-1	1	0	-1	0.37	0.26	0.18	0.13	-0.12	0.041	0.019	0.00055	-0.0089
AUDIO	1	-0.38	-1	-1	1	1	1	1	-1	-1	0	-0.37	-0.26	-0.18	-0.13	0.12	-0.041	-0.019	-0.00055	0.0089
VAR_1	-0.37	-0.4	0.37	0.37	-0.37	-0.37	-0.37	-0.37	0.37	0.37	-0.37	0	0.7	0.49	0.35	-0.21	0.34	0.065	0.021	0.0028
VAR_2	-0.26	-0.28	0.26	0.26	-0.26	-0.26	-0.26	-0.26	0.26	0.26	-0.26	0.7	0	0.71	0.5	-0.15	0.25	0.32	0.025	0.00082
VAR_3	-0.18	-0.2	0.18	0.18	-0.18	-0.18	-0.18	-0.18	0.18	0.18	-0.18	0.49	0.71	0	0.7	-0.1	0.18	0.23	0.23	0.005
VAR_4	-0.13	-0.14	0.13	0.13	-0.13	-0.13	-0.13	-0.13	0.13	0.13	-0.13	0.35	0.5	0.7	0	-0.07	0.13	0.16	0.16	0.17
PRESSED	0.12	0.21	-0.12	-0.12	0.12	0.12	0.12	0.12	-0.12	-0.12	0.12	-0.21	-0.15	-0.1	-0.07	0	-0.0062	-0.0038	0.0073	-0.00072
ACT_1	-0.041	-0.026	0.041	0.041	-0.041	-0.041	-0.041	-0.041	0.041	0.041	-0.041	0.34	0.25	0.18	0.13	-0.0062	0	0.062	0.034	0.015
ACT_2	-0.019	-0.013	0.019	0.019	-0.019	-0.019	-0.019	-0.019	0.019	0.019	-0.019	0.065	0.32	0.23	0.16	-0.0038	0.062	0	0.014	0.006
ACT_3	-0.00055	0.0053	0.00055	0.00055	-0.00055	-0.00055	-0.00055	-0.00055	0.00055	0.00055	-0.00055	0.021	0.025	0.23	0.16	0.0073	0.034	0.014	0	0.017
ACT_4	0.0089	0.0021	-0.0089	-0.0089	0.0089	0.0089	0.0089	0.0089	-0.0089	-0.0089	0.0089	0.0028	0.00082	0.005	0.17	-0.00072	0.015	0.006	0.017	0

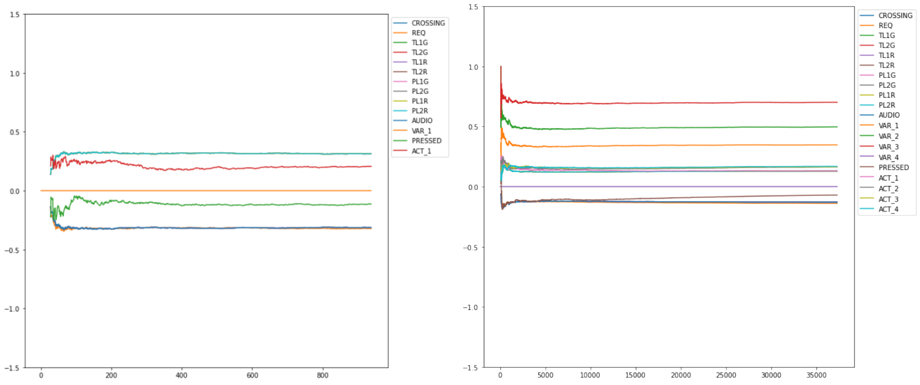
system states. Converging to an accurate estimate of phi requires maximising unique state space observations.

Consider Figure 3. Computing the phi coefficients for a set of binary variables from a generated ladder logic program with 2^{20} theoretical states, produces a symmetric $n \times n$ matrix A_ψ, where $n = |Val_I \cup Val_C|$. Element $a_{ij} \in A_\psi$ refers the phi coefficient between the i^{th} row and j^{th} column. Entries computing self correlation are read on the diagonal, and have been set to 0. Given ϕ coefficients require an empirical sample of observations in order to determine correlation, more unique observations concerning binary variables under evaluation produces more accurate estimates of their relation. Since we wish to learn relations over massive state spaces it is likely our agents begin training with a noisy estimate of variable correlation without observing the majority of states, i.e without knowledge of all variable configurations under the reachable states, ϕ will remain an approximation (isn't it anyway?).

Immediately we notice certain aspects of program semantics, particularly variable assignment conditions, from Figure 3. Upper left quadrant represents base pelican crossing program. See strong positive and inverse correlation between variables responsible for traffic and pedestrian control during crossing procedures. Expectedly, green aspects for traffic TL_1_G, TL_2_G are inversely related to green aspects for pedestrians, i.e both sets of lights should not simultaneously indicate a safe advance (Is this the invariant?). From this we can infer our own trivial invariant property given this relation is never explicitly defined in the program given all values for light aspects are assigned using the present value of $CROSSING$.

Figure **??** demonstrates the convergence of phi over training time steps, each plot illustrating correlation with respect to a single variable under evaluation. Lines fixed at 0 represent the variable under evaluation, ignoring self correlation. Lines fixed at 1 or -1 are variables positively or inversely related to the variable under evaluation. Note during earlier timesteps, estimates of phi are volatile until sufficient state space coverage stabilises the number of samples used in computing ϕ. We see for small programs, these values converge quickly as the number of reachable states are low.

Assessing correlation over time $w.r.t$ ACT_1 and VAR_1



(a) (b)

Fig. 4: bla bla bla

Reward Shaping

Discuss optimising phi coefficients to converge over time.

For larger programs we would expect greater variability in correlation as our RL agents take longer to sufficiently populate A_{τ} with new observations. ϕ coefficients fluctuating between no relation and one of the extremes may suggest insufficient exploration of states

5.5 Use of mining to validate generated data

Highlight generated program / artificially depth

In generating ladder logic programs to serve as training environments we introduce a condition $VAR\_1 = ACT\_1 \wedge (\neg CROSSING) \wedge \neg REQ \wedge PRESSED)$, which only holds for $S1$ of the original pelican crossing. Subsequent rungs build upon this condition for every additional ACT_N and VAR_N, ensuring every n^th action introduced increases the overall state space depth. We would then expect ACT_1,...,ACT_N to be correlated. Observing the lower right quadrant of Figure 3 we see properties of our program generation in how additional actions and variables become less related to our original program state space.

6 Conclusion & Future Work

- Acknowledges need for safety guarantees in railway domain and relevance of model checking - Introduced invariant finding problem for interlockings and how this can reduce occurrence of false positives - Use existing approaches to translate ladder logic to propositional formulae to be modelled as an LTS which can be translated to an MDP, where reinforcement learning can be applied. - Using RL and MDP to intelligently navigate program state space to maximise observations / state coverage - Presented data generation (for invariant mining) technique by stacking agent trajectories during exploration - Briefly discussed contemporary RL exploration strategies and pattern mining techniques - Highlighted some more conventional, state of the art methods for invariant finding - Presented set of synthetically generated programs to serve as evaluation settings where ground truth properties are known. - Presented preliminary results on exploration strategy - Discussed some basic invariants inferred from

6.1 Trajectory Clustering

References

1. 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
2. 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
3. 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
4. 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)
5. 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)
6. 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

7. 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)
8. 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)
9. 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)
10. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. ACM Sigplan Notices **51**(1), 499–512 (2016)
11. Gordillo, C., Bergdahl, J., Tollmar, K., Gisslén, L.: Improving playtesting coverage via curiosity driven reinforcement learning agents. arXiv preprint arXiv:2103.13798 (2021)
12. 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)
13. 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)
14. James, P., Roggenbach, M.: Automatically verifying railway interlockings using sat-based model checking. Electronic Communications of the EASST **35** (2011)
15. Kakade, S.M.: A natural policy gradient. Advances in Neural Information Processing Systems **14** (2001)
16. 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)
17. Medsker, L.R., Jain, L.: Recurrent neural networks. Design and Applications **5**, 64–67 (2001)
18. 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)
19. 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)
20. 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)
21. 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
22. 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
23. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)

24. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
25. Tiegelkamp, M., John, K.H.: IEC 61131-3: Programming industrial automation systems. Springer (2010)
26. 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)
27. 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)