Improving Railway Safety: Human-in-the-loop Invariant Finding

Ben Lloyd-Roberts
ben.lloyd-roberts@swansea.ac.uk
Swansea University
Swansea, United Kingdom

Phillip James
p.d.james@swansea.ac.uk
Swansea University
Swansea, United Kingdom

Michael Edwards
michael.edwards@swansea.ac.uk
Swansea University
Swansea, United Kingdom

Thomas Werner
wt.werner@siemens.com
Siemens Mobility UK
Chippenham, United Kingdom

Simon Robinson
s.n.w.robinson@swansea.ac.uk
Swansea University
Swansea, United Kingdom

ABSTRACT

Formal methods is a field that has a long standing history within Computer Science. At its core, it involves the use of mathematical formalisms to model and reason about computer systems and programs. The application of formal methods to verify that railway signalling systems operate safely and correctly is particularly well established within academia and is now beginning to see real applications in the railway sector. However, many contemporary approaches frequently detect false positive safety violations necessitating lengthy manual analysis by expert engineers. It has been shown that such errors can be mitigated with strengthening invariants, non-trivially generated properties which hold for all reachable states, or configurations, of a program under verification.

In this work, we report on the use of machine learning to explore such state spaces autonomously and provide various visual aids to assist engineers in understanding and inducing invariant properties to support verification. We conducted a focus group with an engineering team specialising in railway signalling systems, soliciting feedback on these visualisations while co-designing suggested improvements for the application domain. The results were two-fold; our visualisations allowed participants to explore candidate invariants, but also highlighted improvements to our machine learning approach by leveraging their domain knowledge.

CCS CONCEPTS

• **Computing methodologies** → *Reinforcement learning*; • **Human-centered computing** → **User studies**; **Visualization design and evaluation methods**.

KEYWORDS

formal verification, machine learning, data visualisation

ACM Reference Format:
Ben Lloyd-Roberts, Phillip James, Michael Edwards, Thomas Werner, and Simon Robinson. 2023. Improving Railway Safety: Human-in-the-loop Invariant Finding. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23), April 23–28, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3544549.3573853

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CHI EA '23, April 23–28, 2023, Hamburg, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9422-2/23/04.
https://doi.org/10.1145/3544549.3573853

1 INTRODUCTION

Railway signalling systems are complex, safety critical, and expected to operate within the confines of a formal safety specification. Deviation from these operational parameters can be catastrophic, leading to collisions, casualties and loss of life. The interlocking, a "safety layer" responsible for receiving operator inputs, such as route setting requests and enactment of physical track changes, is an integral component of such signalling systems. Typically, interlockings are implemented as Programmable Logic Controllers (PLC) [40] with the highest Safety Integrity Level (SIL 4) required for their development [8]. We consider those interlockings implemented in Ladder Logic [20, 30], a purely Boolean, graphical programming language. It is imperative these systems undergo rigorous testing procedures to ensure they adhere to strict and formally defined safety criteria. Formal verification facilitates such procedures by constructing a mathematical model against which properties can be tested. Model checking [15] is one formal verification technique employed to systematically check whether certain properties hold for all configurations (or states) that a system can operate within. Such approaches lack widespread use within the railway industry despite demonstrative promise in academia [3, 12, 18, 19, 25, 26, 31]. In particular, one limitation of model checking solutions is often *false positive* results due to over approximations introduced as part of the mathematical checking, typically when using techniques such as inductive verification [37]. In particular, inductive verification fails to consider whether system states that violate a given property are indeed reachable by the system from a defined initial configuration, rather than from any random initial configuration (the over approximation). These false positives often require lengthy manual inspection, with limited information available and substantial domain expertise required. One solution to overcome this issue is to introduce so-called *invariants* to suppress false positives [4]. Invariants are properties, expressed as simple propositional formulae in our case, which hold for all system states. The aim is to introduce invariants that help bound the region of reachable states when model checking. However, generating sufficiently strong invariants automatically is non-trivial [10].

To support systems engineers in this process, we have co-developed a process for automatically suggesting invariants for human-in-the-loop evaluation with Siemens Mobility UK. In this paper we report

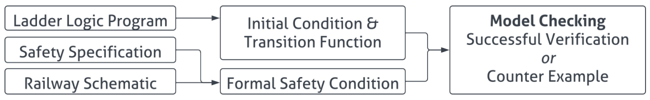


Figure 1: Pipeline for formally verifying railway signalling systems according to a set of safety criteria.

on this process and the development of visual representations for suggesting invariants. In particular, our approach involves training reinforcement learning agents to intelligently traverse program state spaces, inducing state transitions through manipulation of program inputs, whilst recording sets of unique states, or *observations*, and maximising state coverage. Through iterative simulation, a state dataset is generated from which we render a series of plots for both validation and explainability of our framework. We then engage in a process of participatory design in exploring whether these plots allow railway engineers to infer invariants and note structure across programs they develop. Given the complexity of interlocking systems, we do not expect study participants (engineers) to exhibit awareness of this structure. It is envisaged that this structure when inspecting "candidate" properties may help users in deducing invariants and reducing errors within the program.

While these abstractions required some introduction in terms of their "readability", we report on feedback from a focus group of signalling systems engineers. Results pertaining to employment of such visualisations to improve design workflows are discussed, but also somewhat more interestingly, domain expertise provided new developmental insights in terms of our approach's performance and interpretability.

2 BACKGROUND

Our contribution builds upon existing work for model checking Ladder Logic programs [27, 30], although the framework we develop can be applied more broadly to the field of model checking. In the following section we introduce the key concepts, but, as the mathematical models are not the focus of this work, we refer readers to more comprehensive material concerning Ladder Logic [17], model checking [15], program invariants [10] and reinforcement learning [32, 33, 38].

2.1 Model Checking of Ladder Logic

Ladder Logic is a graphical programming language commonly used for PLCs. Essentially, it allows users to construct programs consisting Boolean expressions whose evaluation, based upon the given inputs, results in a set of Boolean outputs being computed. Interlockings run these programs in a cyclic fashion, with inputs constantly being updated with information from the railway (such as train positions) and outputs (or coils) being used to update the state of railway infrastructure (such as signals). By cycling through all permutations of input variable valuations and computing the corresponding outputs, we can unfold the *reachable* state space for a given Ladder Logic program. This state space unfolding is the basis of the verification framework by James et al. [27] upon which we build.

Model checking concerns the theory and practice of mathematically proving system correctness in accordance with some formal specification. Typically, this entails representing systems abstractly, as a set of *states* describing unique configurations a system may assume during its operation, and *transitions* describing the change between these states. Figure 1 depicts the pipeline for interlocking verification taken by James et al. [27]. Reading from left to right, the Ladder Logic program is translated into a model consisting of a set of initial configurations and a mathematical transition function. This is used to compute the previously mentioned state unrolling. Moreover, a generic safety specification is translated into a concrete propositional safety property for a given railway plan. Every unrolled state is then checked to see if it upholds the given safety property. This results in either a positive result indicating the property is true for all system states, or a counterexample trace which highlights a run of the system where the property is violated.

Figure 2 portray an example state space concerning a pedestrian light controlled crossing implemented with safety violations. Each node displays a truncated list of program variables. The root node connected to a single in-edge represents the initial state from which property checks and state traversal begin. Valuations over the input variable `pressed` induce these state transitions, indicating a physical button press and pedestrian request to cross. States are systematically traversed in an attempt to verify whether some formal property holds for every reachable state. In the event verification fails, an error trace from the initial state is returned. States within the green section uphold the safety property, whilst those in the grey section violate it. Figure 2 highlights a valid error trace in blue. Additionally, red nodes denote the occurrence of false positive errors where states in breach of the safety property are unreachable in terms of program execution from the defined initial state. However, such states are still checked when using abstractions that make model checking feasible, such as induction based verification approaches [28–30]. To avoid such false positive results, contemporary solutions aim to introduce invariants that are known properties of the system to constrain the checking of states to those that are reachable. Introducing sufficiently strong invariants helps reduce this over approximation and can mitigate such errors. It is here we use reinforcement learning to navigate the state space and mine invariants from observations throughout exploration.

2.2 Reinforcement Learning for Program Exploration

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) [39]. In reinforcement learning the MDP serves as our *environment*, where through simulation, software agents manipulate program inputs, observe the induced state transitions and learn to optimise this traversal to maximise unique state observations. We expect invariants to manifest as properties persistent across all observations, thus a resultant collection of unique states serves as our dataset for invariant mining. Agent behaviour is shaped through our own reward function which motivates discovery of novel observations. Conversely, repeated observations are deterred with negative rewards and early termination of simulation, restarting the exploration from an initial state.

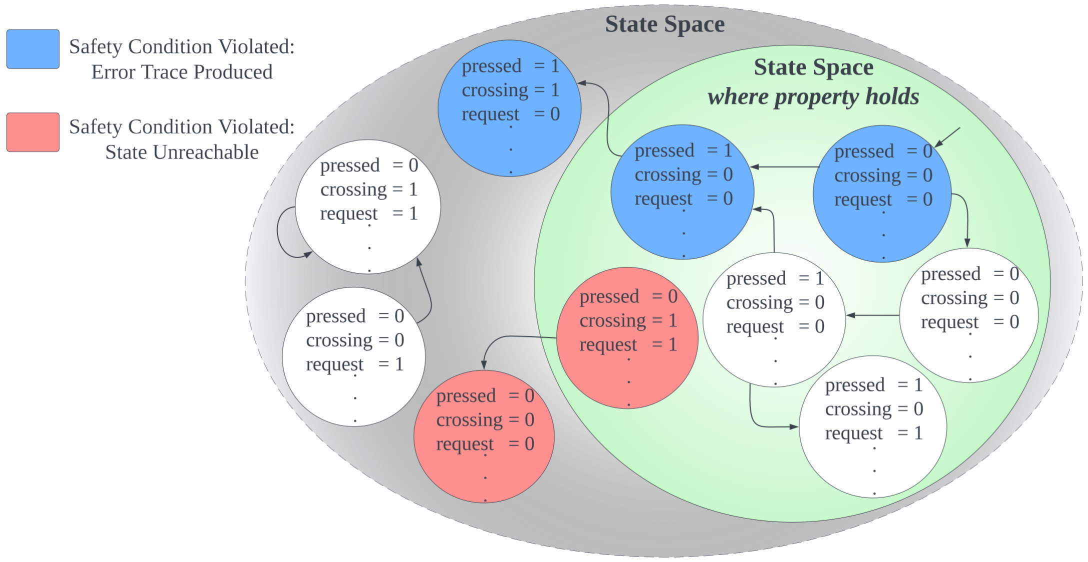

Safety Condition Violated: Error Trace Produced
Safety Condition Violated: State Unreachable
State Space
State Space where property holds
pressed = 1 crossing = 1 request = 0
pressed = 0 crossing = 1 request = 1
pressed = 0 crossing = 0 request = 1
pressed = 0 crossing = 0 request = 0
pressed = 1 crossing = 0 request = 0
pressed = 0 crossing = 0 request = 0
pressed = 0 crossing = 1 request = 1
pressed = 1 crossing = 0 request = 0
pressed = 0 crossing = 0 request = 0
pressed = 1 crossing = 0 request = 1

Figure 2: Illustration of state space for program under verification. Nodes and edges indicate program states and their transitions. States found within the central radius (green) satisfy the safety property. States falling within the outer radius (grey) violate the property.

3 RELATED WORK

Automatic generation of invariants has received significant academic attention in recent years in terms of algorithmic contributions [9, 34, 36]. Many of these approaches focus on generating invariants for high level programming languages, rather than purely Boolean languages. At a high level, several approaches [4, 7, 11] employ some form of iterative clause refinement through abstract representations or symbolic computation. While entirely explainable, i.e logical operations applied to explicit state valuations, the steps toward invariant proposal are typically obfuscated from any end users. This is fine for the application domain of programming languages and verification. However, in the context of safety critical work, it is necessary to explain and understand any invariants that are introduced. Some approaches to invariant proposal employing machine learning have adopted explainable practices, such as the use of decision trees by Krishna et al. [21] but here no attention is given to explaining how these properties are built. Having implemented our own invariant mining technique with easily explained and interpreted statistics, engagement with our focus group revealed potential benefits in explaining this process, and involving users in invariant formulation (see Section 6).

More broadly, the usability of formal methods tools has been considered with respect to debugging of programs and in particular understanding counterexamples. Groce et al. [22–24] build on existing model checking tools to localise faults in programs through an IDE-like environment [14]. Using a familiar development suite was seen as essential for developer usability, and their tool allowed developers to focus on localised errors in code by comparing differences between variations of the same counterexample. Several existing contributions build upon this notion of causality in traces [5, 13, 35, 42], however in contrast to our work, this removes the error from the context of the overall system state space. Similarly, interpretation of raw outputs returned by model checkers, which is often highly mathematical, has been addressed for interlocking verification by van den Berg et al. [41]. Considering animation of simulated counterexamples as prone to introducing modelling errors or open to misinterpretation, the authors tailor error depiction according to the application domain. However, new visuals must be drawn for every new counterexample. Our approach depends on no such heuristics, simply reducing the complexity of a view with which our end users are already familiar.

Approaches to visualising state spaces often use symbolic states and transitions, ignoring the absolute changes in variables. For example, UPPAAL [6] supports visualisation of system specification through symbolic states, i.e viewing traces as a message sequence chart, but not explicit state valuations. Ivy [36], an interactive verification tool, avoids the complexities of explicit representation by ensuring specifications are defined using a restricted modelling language. Other graph-based visualisation tools have proposed filtered or reduced views of explicit states to narrow counterexample analysis to some sub-region of the space [41]. Similary, Aljazzar et al. [1, 2] present DiPro, a tool built upon stochastic model checkers to find probabilistic counterexamples and render them as interactive directed graphs. Visual analytics are then utilised to highlight features likely to support debugging. We suggest simplifying the view of error traces and provide visual analytics to aid debugging within the context of an exploration graph, over which counterexamples can be depicted, simplifying the textual sequence with an interactive graphical one.

4 VISUALISATION & DATA GENERATION

In this section, we introduce three visualisations to co-design with our focus group. Each plot provides a unique view of features pertaining to the program under verification, representing different levels of abstraction for different analytical purposes. For each visualisation, we generate two examples: a proof of concept to explain the visualisation and an industrial program to solicit feedback. The first refers to a Ladder Logic program implementing a pedestrian light controlled crossing, presented in [27]. Plots concerning this program are presented in Figure 3 (e.g, variables prefixed `TL_*` denote traffic signals). The second refers to a proprietary Siemens interlocking program for a section of railway.

4.1 State diagram (Staterix)

In the case of invariant generation, it is possible to observe patterns across states without considering transitions. To that end, we can visualise the state space statically as an image. Consider the state matrix, or *staterix*, in Subfigure 3a. Each column represents a distinct variable while each row represents a distinct state. Cells are then coloured to denote a binary value (blue : 0, yellow : 1) for that variable, given the state. For example, we observe for states $S_0, ..., S_5$, variables `CROSSING` and `AUDIO` consistently share the same valuation. We hypothesise that this staterix allows expert users to quickly identify regions of interest based on salient patterns across states.

4.2 Phi Coefficient Matrix

Pearson's correlation coefficient, or the ϕ coefficient, is a statistical measure of relation between two binary variables [16]. Thus, given a program composed of N binary variables, we can compute an $N \times N$ correlation matrix between all pairs of variables. Coefficients range between $[-1, 1]$, where -1 denotes a complete inverse relation and 1 denotes a complete positive relation. By observing the correlation 1 between variables `CROSSING` and `AUDIO` in Subfigure 3b, we confirm the complete relation suggested by the staterix in Subfigure 3a. Estimates of ϕ are computed based on the variability of any two variables across the reachable state space, meaning all states must be observed once to compute the true correlation. Consequently, ϕ converges over time, warranting a distinction between three types of invariants in terms of these coefficients. Extrema of -1 or 1 in the event of complete state space coverage may be considered "true invariants" or "believed invariants" when only partial coverage is achieved. Coefficients nearing complete or inverse relations, irrespective of state coverage, are referred to as "candidate invariants". These have yet to be proven invariant but with some code modifications, could remove the state(s) preventing the complete relation. We also note that incorporating machine learning in the design of safety critical systems is likely to be met with skepticism. Statistical invariants suggested through empirical measures of variability should make sense conceptually (i.e are interpretable by developers responsible for building the system). Aiming to deliver a tool capable of invariant suggestion, our approach should incorporate explainability measures where possible. While the process of invariant generation can be automated, ultimately it is developers who are responsible for incorporating any suggested invariants when model checking.

4.3 Exploration Graph

The staterix allows users to identify invariants across state valuations at a glance while the correlation matrix attributes a quantitative value of that invariance. To contextualise where such invariants may hold in terms of the program state space, we draw an interactive directed *exploration* graph. Depicting the program's runtime behaviour, nodes represent distinct valuations over the program variables where connected edges denote variable changes inducing a transition between those states. Subfigure 3c depicts an exploration graph composed of the 6 reachable states (and initial state `IS`) represented in Subfigure 3a. Values corresponding to a unique state or transition are stored explicitly, allowing users to interact with and query the graph. Graph structure is also stored via an adjacency matrix, supporting basic connectivity metrics for visual analytics. We hypothesise expert users investigating candidate invariants would be able to quickly determine whether they should be true invariants by inspecting the subset of states for which a property does not hold.

5 FOCUS GROUP: METHODOLOGY

We conducted a study constituting the participatory design process in delivering a support tool for invariant mining. We presented our set of visualisations to a group of domain expert developers to solicit feedback regarding their utility in supporting manual verification processes and explainability of results from our RL setting. Study participants comprised a team of 6 systems engineers with a diverse skillset within railway signalling. Participants recruited were asked to characterise their principle roles within the engineering team. This contextualised feedback according to the participant's domain expertise. Each participant had a comprehensive understanding of the overarching system within which interlockings function, as well as specialising in a certain facet of that system. The team comprised a senior lead within research and development (Participant A), four software engineers responsible for interlocking implementation and verification (Participants B-E) and a project coordinator (Participant F) within the R&D department. Recruitment was performed based on convenience and snowball sampling as the global population for such engineers is relatively small, and with study data subject to intellectual property constraints. No identifiable participant data was collected beyond names and signatures submitted via consent forms. Only the opinions and suggestions expressed during group discussions were recorded for study purposes.

5.1 Procedures

We conducted a series of semi-structured discussions with focus group participants in an isolated office space, limiting external stimuli which could disrupt the session. The study was conducted by three researchers, one of whom led an informal introduction and opened the floor to any preliminary questions, before issuing a consent form. An accompanying information sheet outlining data collection, study objectives and implications of participation was also provided. Focus group participants were then introduced to the theoretical concepts behind the presented visualisations, as they are predominantly abstract representations of the formally defined system such users typically employ. An explanation as to how these plots were rendered, what it is they represent and how they are labelled was provided using examples from Figure 3,

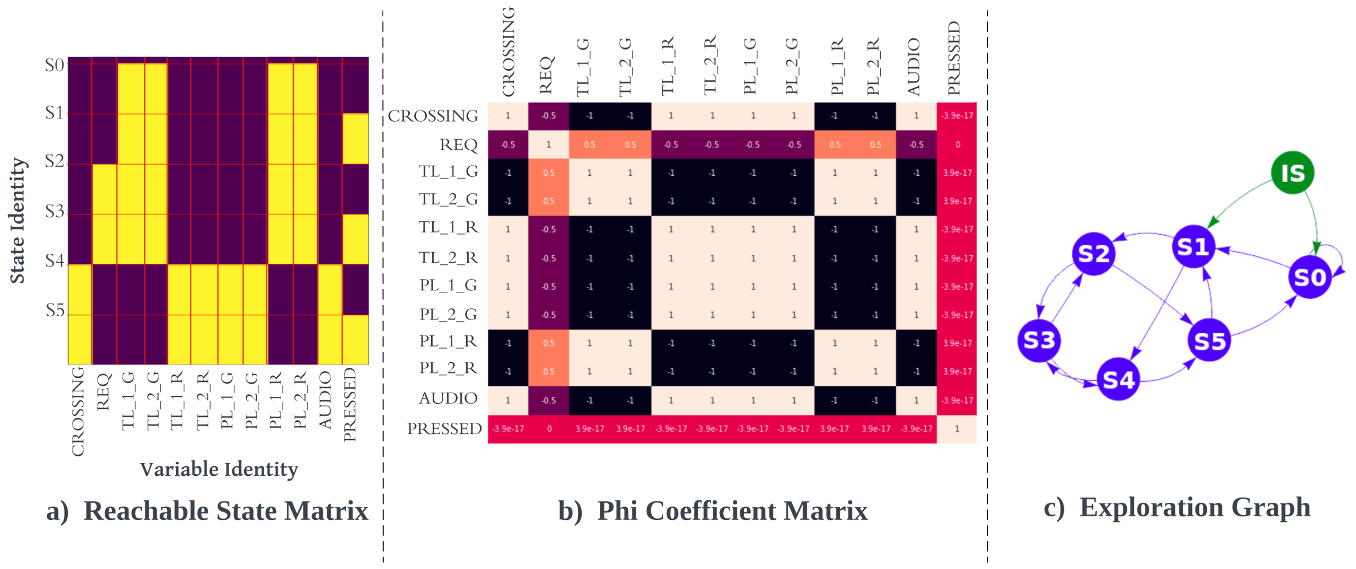


Figure 3: Visualisations of reachable state matrix, correlation matrix and exploration graph.

which concern the pelican crossing program referenced in section 4. Beyond understanding these “readability” requirements, we invited participants (i.e prospective users) to develop their own insights from the plots rendered. Our set of visualisations, both pelican crossing and interlocking examples, were presented via a laptop and shared display, controlled by one researcher, cycling through plots in accordance with the flow of discussion.

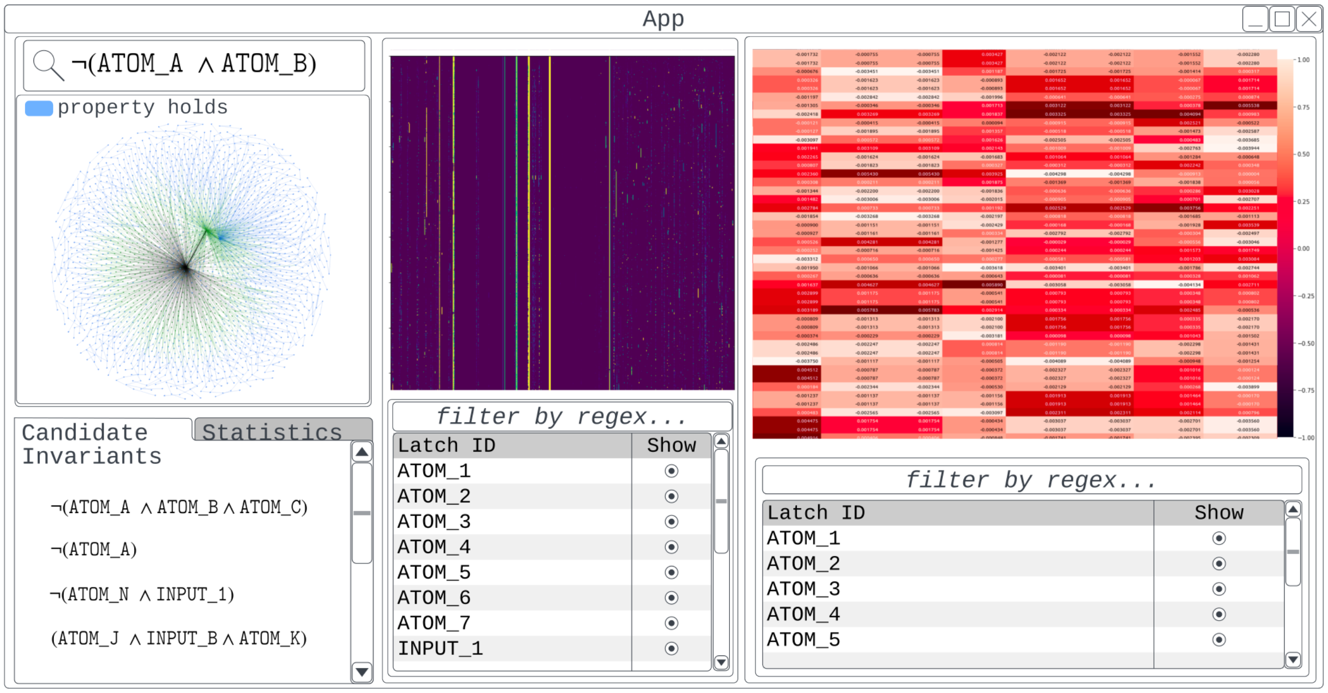
A principle aim of our focus group was to determine how interpretable our invariant mining techniques are. For each staterix, participants were asked to identify patterns which appeared invariant or of interest. If invariants could be found, they were asked to intuit why this property was present given the context. Given each correlation matrix, participants were asked to identify complete relations between two variables and posit why this correlation exists. Finally, within the context of an exploration graph, we asked participants how they would expect to view invariants.

6 FOCUS GROUP: OUTCOMES

This section outlines key findings from the focus group, concerning the usability of our proposed visualisations. Results went beyond the scope of our research questions, suggesting our program abstractions were informative in many respects, more so than we had initially hypothesised. The diversity of participants in terms of their role was also reflected in their feedback. Personnel in more senior positions within the development team showed the most interest in making the process hidden from verification engineers and as automated as possible. Participants more involved in program implementation demonstrated a greater interest in leveraging the existing plots to inform debugging processes. Unexpectedly, our visualisations abstracted enough of the RL learning framework for group participants to leverage their domain expertise and suggest improvements to guide training. Consequently, we identify the utility of an interactive debugging tool to support human-in-the-loop invariant generation. We include a mock design in Figure 4.

6.1 Staterix

As the highest level of abstraction, the staterix appeared easiest to understand and proved our hypothesis was indeed valid. All participants could suggest some property which appeared invariant and were able to offer an explanation as to why the property existed. Across members of the focus group, three properties regarding the underlying Ladder Logic program were noted. First, logical expressions present in lines of code were apparent as certain variable columns mirrored each other’s value. Solid columns within the staterix displaying a single colour were also noticed to be invariant, i.e the variable was either always True (1 : yellow) or always False (0 : blue). Another point raised during this discussion concerned values which appeared to rarely change, i.e were set as False for the majority of reachable states. A lead software engineer remarked that these values were likely “noise” introduced by intermediary values, variables introduced as a placeholder for some overarching operation. Indeed, inspecting the interlocking staterix depicted in Figure 4, this appears to be the case for a large number of variables. For the interlocking program, these patterns provided a quick method of localising areas of interest given the staterix comprises thousands of columns and potentially millions of rows. Being large systems comprising thousands of program variables and lines of code, representing the state space explicitly can become cumbersome. Abstracting states also removes granular information such as valuations over variables, which the staterix avoids. In the event verification produces a counterexample, more experienced engineers may choose to query operations pertaining to a subset of variables based on staterix patterns, such as which variables change frequently, which variables mirror each other or, in the case of solid columns, which values are invariant. It was proposed by participant D this staterix be made interactive, allowing users to filter the x-axis by toggling a list of program variables or by regular expression, to support this level of inspection.



¬(ATOM_A ∧ ATOM_B ∧ ATOM_C)

¬(ATOM_A)

¬(ATOM_N ∧ INPUT_1)

(ATOM_J ∧ INPUT_B ∧ ATOM_K)

Latch ID	Show
ATOM_1	◉
ATOM_2	◉
ATOM_3	◉
ATOM_4	◉
ATOM_5	◉
ATOM_6	◉
ATOM_7	◉
INPUT_1	◉

Latch ID	Show
ATOM_1	◉
ATOM_2	◉
ATOM_3	◉
ATOM_4	◉
ATOM_5	◉

Figure 4: Mock design for interactive debugging tool for invariant generation and counterexample analysis, resulting from focus group outcomes. Variable identifiers have been replaced where plots refer to results from proprietary programs.

6.2 Correlation Matrix

Our intuition as to the explainability of correlation estimates for invariant generation was correct. Understanding phi correlation is simply a measure of variability between a binary pair, participants quickly interpreted the values present in the correlation matrix for both examples. Unlike contemporary invariant generation methods, little background knowledge is required to understand how correlation is computed and what those values denote. Wherever an invariant exists, at least one binary pair will be found with a complete correlation -1 and 1, thus making measures of invariance easy to interpret and explainable. Participant B, specialising in the implementation of verification systems, suggested using ϕ coefficients to reduce the program model. If we find $\phi = -1$ between literal a and literal b, then we can replace all occurrences of variable b by the negation of a. Applying this reduction across all instances of inverse correlation may in turn reduce the set of reachable states. Participant C noted that deducing candidate invariants from the correlation matrix was done manually and concerned only literal pairs. Since ϕ can be computed between any binary valued pair, it was suggested we determine the relation between conjunctions of literals to form more complex invariants. Variable identifiers in Ladder Logic, particularly within the railway industry, tend to share certain suffixes/prefixes, which can be used to determine association between variable subsets and to what system functionality they pertain.

Manually constructed invariants are typically deduced by domain experts familiar with the system under verification. The use of a correlation matrix was also discussed in this respect. Developers with intuition as to which variables may compose an invariant could leverage this matrix as a way of quickly testing this intuition. It was highlighted that for programs comprising thousands of variables, this matrix soon becomes unintelligible. Similar to staterix interaction, filtering correlation values by a variable was suggested to reduce this complexity. Following a sufficiently long exploration run, an agent may compute the correlation matrix under the current set of observed reachable states. Once saved, we could allow users to filter results by variable ID, again through a list of toggles or by regular expression.

6.3 Exploration Graph

While our hypothesis concerning the usability of an exploration graph to illustrate invariants across a state space was initially valid, there were other applications discussed that we had not considered. All focus group participants were familiar with the role of directed graphs as a means of system abstraction. It is therefore understandable that depictions and generation of an exploration graph were quickly grasped. Participant C asked for clarification regarding what, explicitly, each node represented. Each node was, by design, used to represent the same vector of variable valuations that participants were familiar with in textual form. All participants then agreed any invariants visualised in terms of an exploration graph should be highlighted across all nodes where that property holds, whether across the entire graph or for some sub-region. Participant B noted that in the event no invariants were generated from RL exploration, the graph could serve in highlighting properties queried through manual search. This way users could could inspect sub-regions where some arbitrary property holds and determine why they fail to hold in others. Similarly, they could iteratively refine such a property while visualising how well that candidate invariant holds across reachable states.

In addition to this, it was highlighted that current procedures for debugging counterexample traces involve processing large bodies of text. Participant B suggested that using our exploration graph to illustrate error traces as a highlighted, interactive sequence of nodes and edges, indicating paths from initial state to the point of failure would be highly useful. Participant D noted such a tool would provide users a sense of structure and how the program behaves at runtime, and where that counterexample occurs in relation to what the RL agents have observed. It was suggested users were allowed to inspect states and their connected edges via tooltip interaction. Node selection would produces a list of tuples mapping variable identifiers to their state valuation, while edge selection would produce a list of tuples mapping program inputs to their valuation (inducing the transition). Participant B remarked that within the exploration graph view, users could specify properties to search for across states, highlighting nodes where the property holds. A mock user interface incorporating these proposals is shown in Figure 4. The default proposed view would show the complete graph, displaying a list of candidate invariants which hold for all reached states. Individual states could also be selected to inspect node metrics. Participant F proposed allowing users to select an initial state to seed further exploration. False positive debugging of error traces is complicated, time consuming and requires specialist knowledge. Debugging processes may be accelerated, or at least rendered more comprehensible, with error trace abstraction to simplify system views. If a false positive counterexample trace is produced, the agent should be unable to reach the state of property infraction. One proposal was to append a highlighted path from some ultimate reachable state to the infraction state. Consequently, we localise where the counterexample falls which may help users narrow down causal factors.

During focus group discussions concerning the interlocking exploration graph, some participants drew attention to a small number of densely connected nodes. Knowing the interlocking system effectively has two modes of operation, it was suggested these highly connected points represented key states where those modes initialised. Realising that basic graph properties could translate to some well understood aspect of the system motivated the idea of incorporating graph based metrics. These are split into statistics concerning the complete graph and individually selected states, most of which may provide some indication of state "importance". Simple global measures such as edge and node count provide a basic indication of system size and connectivity, as well as individual node statistics for in and out degree. Radiality centrality attributes values to nodes based on their connection to all other nodes, compared to graph diameter. High valued nodes suggest the state is frequently observed at runtime, possibly indicating some return to default values. Graph reciprocity reflects the number of bidirectional edges between states. Perhaps states with no reciprocal edges could be pruned, indicating some redundant intermediary operations.

6.4 Agent Refinement through Participatory Design

When discussing the staterix for interlocking programs, there appeared obvious invariants in the form of solid columns. Initially we had believed these were indicative of some redundant block of code within the program until participant D, a lead software tester, proposed this variable set was too large, and likely indicated poor exploration within a sub-region of reachable states. It was then suggested the ϕ coefficient, which converge over time, be used to modify the RL reward scheme and incentivise agents to mitigate overconfidence by disproving invariants.

An unexpected outcome of our focus group was that of participant suggestions to improve our reinforcement learning framework. In particular, it appeared our set of visualisations were sufficient in abstracting the complexities of agent and environment implementations such that our domain experts could leverage their own knowledge of interlocking systems to guide state space exploration. Primarily suggested by the software development specialists, a set of testing procedures easily replicated by some distributed collection of agents would guide exploration along an execution path similar to that traced by human engineers. A hybrid approach was devised where a "master" agent follows a state sequence replicating manual debugging while periodically spawning worker agents to explore that sub-region, initially at random. This was motivated by the engineers understanding of errors typically occurring at edge cases for well understood functions, not in some obscure, infrequently observed, poorly connected state.

7 CONCLUSION

This paper presents a novel set of visualisations for depicting invariants present in Ladder Logic programs, and explored how those representations can support effective human-in-the-loop invariant generation techniques. Our visual abstractions provided a mutual understanding of the underlying machine learning approach, facilitating a participatory design process where improvements to both agent training and human interaction were suggested. Two principle themes emerged throughout our study. First, the success of our focus group seemed highly contingent on the passion exhibited by users. Without their willingness to explore where human strengths and weaknesses lie, our research reaches an impasse. Second, while humans solving complex industrial tasks may initially desire for manual processes to be entirely automated and obfuscated, our work shows a clear benefit to leveraging both capabilities.

ACKNOWLEDGMENTS

Ben Lloyd-Roberts is funded by the EPSRC EPIC Centre for Doctoral Training (EP/S021892/1) and Siemens Mobility UK.

REFERENCES

[1] Husain Aljazzar, Florian Leitner-Fischer, Stefan Leue, and Dimitar Simeonov. 2011. Dipro-a tool for probabilistic counterexample generation. In *International SPIN Workshop on Model Checking of Software*. Springer, 183–187.

[2] Husain Aljazzar and Stefan Leue. 2008. Debugging of dependability models using interactive visualization of counterexamples. In *International Conference on Quantitative Evaluation of Systems*. IEEE, 189–198.

[3] Arturo Amendola, Anna Becchi, Roberto Cavada, Alessandro Cimatti, Andrea Ferrando, Lorenzo Pilati, Giuseppe Scaglione, Alberto Tacchella, and Marco Zamboni. 2022. NORMA: a tool for the analysis of Relay-based Railway Interlocking

Systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 125–142.

[4] Mohammad. Awedh and Fabio. Somenzi. 2006. Automatic invariant strengthening to prove properties in bounded model checking. In *ACM/IEEE Design Automation Conference*. 1073–1076. https://doi.org/10.1145/1146909.1147180

[5] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. 2009. Explaining counterexamples using causality. In *International Conference on Computer Aided Verification*. Springer, 94–108.

[6] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. 2006. Uppaal 4.0. (2006).

[7] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. 1996. Powerful techniques for the automatic generation of invariants. In *International Conference on Computer Aided Verification*. Springer, 323–335.

[8] Jean-Louis Boulanger. 2015. *CENELEC 50128 and IEC 62279 standards*. John Wiley & Sons.

[9] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 70–87.

[10] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. 2009. Strengthening Model Checking Techniques With Inductive Invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 1 (2009), 154–158. https://doi.org/10.1109/TCAD.2008.2009147

[11] Michael L Case, Alan Mishchenko, and Robert K Brayton. 2007. Automated extraction of inductive invariants to aid model checking. In *Formal Methods in Computer Aided Design*. IEEE, 165–172.

[12] Roberto Cavada, Alessandro Cimatti, Sergio Mover, Mirko Sessa, Giuseppe Cadavero, and Giuseppe Scaglione. 2018. Analysis of relay interlocking systems via SMT-based model checking of switched multi-domain Kirchhoff networks. In *Formal Methods in Computer Aided Design*. IEEE, 1–9.

[13] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. 2013. Flow-sensitive fault localization. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 189–208.

[14] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176.

[15] Edmund M Clarke. 1997. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 54–56.

[16] Joakim Ekström. 2011. The phi-coefficient, the tetrachoric correlation coefficient, and the Pearson-Yule Debate. (2011).

[17] Kelvin T. Erickson. 1996. Programmable logic controllers. *IEEE Potentials* 15, 1 (1996), 14–17. https://doi.org/10.1109/45.481370

[18] Alessandro Fantechi, Wan Fokkink, and Angelo Morzenti. 2012. Some trends in formal methods applications to railway signaling. In *Formal Methods for Industrial Critical Systems: A survey of applications*. John Wiley & Sons, 61–84.

[19] Alessandro Fantechi, Anne E Haxthausen, and Hugo D Macedo. 2017. Compositional verification of interlocking systems for large stations. In *International Conference on Software Engineering and Formal Methods*. Springer, 236–252.

[20] Wan Fokkink, Paul Hollingshead, J Groote, S Luttik, and J van Wamel. 1998. Verification of interlockings: from control tables to ladder logic diagrams. In *Proceedings of Formal Methods for Industrial Critical Systems*, Vol. 98. 171–185.

[21] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.

[22] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. 2006. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer* 8, 3 (2006), 229–247.

[23] Alex Groce, Daniel Kroening, and Flavio Lerda. 2004. Understanding counterexamples with explain. In *International Conference on Computer Aided Verification*. Springer, 453–456.

[24] Alex Groce and Willem Visser. 2003. What went wrong: Explaining counterexamples. In *International SPIN Workshop on Model Checking of Software*. Springer, 121–136.

[25] Anne E Haxthausen, Marie Le Bliguet, and Andreas A Kjær. 2008. Modelling and verification of relay interlocking systems. In *Monterey Workshop*. Springer, 141–153.

[26] Anne E Haxthausen, Jan Peleska, and Ralf Pinger. 2013. Applied bounded model checking for interlocking system designs. In *International Conference on Software Engineering and Formal Methods*. Springer, 205–220.

[27] Phillip James, Andy Lawrence, Faron Moller, Markus Roggenbach, Monika Seisenberger, Anton Setzer, Karim Kanso, and Simon Chadwick. 2013. Verification of solid state interlocking programs. In *International Conference on Software Engineering and Formal Methods*. Springer, 253–268.

[28] Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne. 2014. Techniques for modelling and verifying railway interlockings. *International Journal on Software Tools for Technology Transfer* 16, 6 (2014), 685–711.

[29] Phillip James and Markus Roggenbach. 2011. Automatically verifying railway interlockings using SAT-based model checking. *Electronic Communications of the EASST* 35 (2011).

[30] Karim Kanso, Faron Moller, and Anton Setzer. 2009. Automated verification of signalling principles in railway interlocking systems. *Electronic Notes in Theoretical Computer Science* 250, 2 (2009), 19–31.

[31] Christophe Limbrée, Quentin Cappart, Charles Pecheur, and Stefano Tonetta. 2016. Verification of railway interlocking-compositional approach with OCRA. In *International Conference on Reliability, Safety, and Security of Railway Systems*. Springer, 134–149.

[32] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1602.01783* (2016). arXiv:1602.01783 [cs.LG]

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[34] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[35] Vijayaraghavan Murali, Nishant Sinha, Emina Torlak, and Satish Chandra. 2014. What gives? A hybrid algorithm for error trace explanation. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 270–286.

[36] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 614–630.

[37] Hendrik Post, Carsten Sinz, Alexander Kaiser, and Thomas Gorges. 2008. Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking. In *IEEE/ACM International Conference on Automated Software Engineering*. 188–197. https://doi.org/10.1109/ASE.2008.29

[38] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017). arXiv:1707.06347 [cs.LG]

[39] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[40] Michael Tiegelkamp and Karl-Heinz John. 2010. *IEC 61131-3: Programming industrial automation systems*. Springer.

[41] Lionel van den Berg, Paul Strooper, and Wendy Johnston. 2007. An automated approach for the interpretation of counter-examples. *Electronic Notes in Theoretical Computer Science* 174, 4 (2007), 19–35.

[42] Chao Wang, Zijiang Yang, Franjo Ivančić, and Aarti Gupta. 2006. Whodunit? causal analysis for counterexamples. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 82–95.