3-2020

Learning fault models of cyber physical systems

Teck Ping KHOO
Singapore University of Technology and Design

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Sudipta CHATTOPADHYAY
Singapore University of Technology and Design

Published in Formal methods and software engineering: 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, March 1-3: Proceedings. 12531 LNCS, pp. 147-162. DOI: 10.1007/978-3-030-63406-3_9

Learning Fault Models of Cyber Physical Systems

Teck Ping Khoo[1(✉)], Jun Sun[2], and Sudipta Chattopadhyay[1]

[1] Singapore University of Technology and Design, Singapore, Singapore
teckping_khoo@mymail.sutd.edu.sg, sudipta_chattopadhyay@sutd.edu.sg
[2] Singapore Management University, Singapore, Singapore
junsun@smu.edu.sg
https://www.smu.edu.sg/, https://www.sutd.edu.sg/

Abstract. Cyber Physical Systems (CPSs) comprise sensors and actuators which interact with the physical environment over a computer network to achieve some control objective. Bugs in CPSs can have severe consequences as CPSs are increasingly deployed in safety-critical applications. Debugging CPSs is therefore an important real world problem. Traces from a CPS can be lengthy and are usually linked to different parts of the system, making debugging CPSs a complex and time-consuming undertaking. It is challenging to isolate a component without running the whole CPS. In this work, we propose a model-based approach to debugging a CPS. For each CPS property, active automata learning is applied to learn a fault model, which is a Deterministic Finite Automata (DFA) of the violation of the property. The L* algorithm (L*) will find a minimum DFA given the queries and counterexamples. Short test cases can then be easily extracted from the DFA and executed on the actual CPS for bug rectification.

This is a black-box approach which does not require access to the PLC source code, making it easy to apply in practice. Where source code is available, the bug can be rectified. We demonstrate the ease and effectiveness of this approach by applying it to a commercially supplied miniature lift controlled by a Programmable Logic Controller (PLC). Two bugs were discovered in the supplier code. Both of them were patched with relative ease using the models generated. We then created 20 mutated versions of the patched code and applied our approach to these mutants. Our prototype implementation successfully built at least one model for each mutant corresponding to the property violated, demonstrating its effectiveness.

Keywords: Debugging · Active automata learning · L* algorithm · Programmable logic controllers

Cyber Physical Systems (CPSs), being distributed and embedded systems, are the drivers of modern applications such as smart buildings, smart healthcare,

Supported by TÜV SÜD Asia Pacific Pte Ltd.

highly automated driving and Industry 4.0. As such, bugs in CPSs can have severe or fatal consequences. Debugging CPSs is therefore an important real world problem with ongoing research efforts.

Existing methods for debugging CPSs fall into the category of simulation, offline debugging, and online debugging. Debugging by simulation works by injecting suspicious inputs into a CPS simulator and checking the output. Simulating the CPS generally requires developing a representative digital twin. It is usually hard to simulate the physical processes accurately. Offline debugging by log file analysis works by gathering large amounts of log files produced by the CPS. Normal and buggy traces are then compared to determine which variables are not changing in the expected way and causing the bug. Usually, the buggy traces mostly contain information irrelevant to the bug, and filtering them away can be difficult. Online debugging by setting breakpoints works by instrumenting a debugger with the CPS in real-time, and setting breakpoints in the CPS's program execution to determine the faulty input. Setting breakpoints may interfere with the buggy behavior being rectified. Additionally, for effective debugging, these breakpoints should be set in multiple components of the CPS. Getting logging to work in a distributed system is hard due to synchronization issues.

We aim to simplify the debugging of a CPS by developing a two-step methodology to build fault models of the CPS:

1. For each CPS property, develop an oracle which accepts a buggy sequence of inputs and rejects a normal one.
2. Apply active automata learning to build one DFA for each CPS property using its oracle, using suitable parameters, for debugging. Repeat Step 1 if needed till the fault model evolves into a sufficient representation of the bug.

L* [1] iteratively generates test sequences of inputs to the PLC, which may or may not lead to an error. The algorithm will automatically and systematically build a small DFA which generalizes all the various ways that the fault can be reproduced up to some number of steps. If none of the test sequences end in an error, then the returned DFA will simply have a single rejecting state. The returned DFA can provide a concise representation of the bug to the test engineer - instead of just knowing one sequence to reproduce the bug by testing, *the engineer now knows multiple shorter sequences to do so.*

We conducted a case study using a PLC-controlled miniature elevator system which was delivered with its specifications. Initial testing of this system revealed that it contained a number of bugs. For example, during some operation, it was possible for its doors to open while it was moving - a clear safety violation. Due to the system complexity, both the test engineers and the vendors have struggled to identify the cause of the bug for months. By specifying just two inputs, which are calling for levels 1 and 2 from the lift car, we were able to learn a fault model with only four states for this bug. After recovering the PLC source code from the PLC, we were able to fix the bug in a day.

In short, we make the following technical contributions:

1. We developed a two-steps model based approach to debugging a CPS. Given a particular property of the system, an extended L* algorithm is used to build a

minimum DFA which capture sequences of events which lead to the violation of this property, up to some number of steps and with suitable parameters. Bug-relevant short test cases can then be extracted from these models to aid in debugging the system.
2. We demonstrate the usefulness of this approach by applying it to study and patch multiple actual bugs in a miniature lift.
3. We demonstrate the effectiveness of our framework via comparison with random testing to fix the two actual CPS bugs. The comparison reveals that our framework consistently provided shorter test cases compared to random testing. Additionally, we statically analysed the source code of the PLC and mutated it to generate 20 bugs. Each mutant was created to trigger a violation of at least one of the ten CPS properties of interest. The expected fault models were all generated.
4. We share on how to select parameters when performing active automata learning on real systems.

Organization. The rest of this paper is organized as follows: Sect. 1 describes the system motivating our study. Section 2 gives an overview of our approach. Section 3 describes how we implemented our approach on the miniature lift. Section 4 poses and addresses some research questions. We evaluated our approach by applying it to debug some supplier PLC source code successfully. We also share the results of applying our approach to 20 mutations of the patched PLC source code. In Sect. 5 we review related work. Lastly, in Sect. 6 we conclude and provide some suggestions for further work.

1 System Description

The system under test is a fully functioning miniature lift developed for training purposes. This system was commercially purchased for the development of smart lift technologies. The system has four lift levels, a level sensor at each floor, a slow-down sensor in between each floor, a traction machine, a pulley system, door motors, buffer stops, and buttons for the lift car and at every floor for user input. The lift is controlled by a Mitsubishi FX3U-64M PLC and comes programmed as a double-speed lift. Upon moving off, a default normal speed will be used. If it reaches a slow-down sensor and if the next floor is to be fulfilled, a default slow speed shall be used. Each time the lift passes a slow-down sensor, the displayed floor will also be updated. This PLC has 32 input devices which are named as X0–X7, X10–X17, X20–X27, and X30 to X37. It has 32 output devices which are named as Y0–Y7, Y10–Y17, Y20–Y27, and Y30 to Y37.

Our approach does not require the source code. However, source code is needed for actual bug fixing. A Mitsubishi toolkit was used to extract the ladder logic source code from the PLC for analysis. This code comprises 1,305 rungs, Rungs 1 to 309 specifies the application, while the remaining rungs define 39 sub-routines named as P0 to P37, and P42. The comments for the program were not provided by the suppliers - as is a common practice for suppliers after

Table 1. PLC inputs of interest

	Purpose	Logic		Purpose	Logic
X17	L1 pressed	Normally Off	X1	Lift car is level with a floor	Normally Off
X20	L2 pressed	Normally Off	X2	Slowdown sensor active	Normally Off
X21	L3 pressed	Normally Off	X10	Lift car is at L4	Normally On, Off at L4
X23	L4 pressed	Normally Off	X11	Lift car is at L1	Normally On, Off at L1

Table 2. PLC Outputs of Interest

	Purpose		Purpose
Y2	Commands the doors to open	Y12	Switches the car L3 button light
Y3	Commands the doors to close	Y13	Switches the car L4 button light
Y6	Commands the lift to rise	Y30	Lights and dims the up display
Y7	Commands the lift to lower	Y31	Lights and dims the down display
Y4	Commands the lift to move slow	Y32	Shows L1 as the current floor
Y5	Commands the lift to move fast	Y32, Y33	Shows L3 as the current floor
Y10	Switches the car L1 button light	Y33	Shows L2 as the current floor
Y11	Switches the car L2 button light	Y34	Shows L4 as the current floor

system delivery, for protection of their intellectual property. Written approval to use their source code in a research paper was provided by the suppliers.

Table 1 summarizes the input devices, name, purpose and logic of the PLC's key inputs of interest, derived from both the system specifications as well as empirical observations. On the panel representing the lift car buttons, four of the inputs are for L1 to L4, and two are for door open and close. There is an "up" button on L1 and a "down" button on L4. There are "up" and "down" buttons on both L2 and L3, making a total of 12 user buttons. We selected only the four lift car buttons for L1 to L4 to reduce the experiments' complexity. Note that using these four buttons can already move the lift to all the four floors, as well as open and close the doors.

Table 2 provides a summary of the PLC's outputs of interest.

The system was originally purchased for the development of smart lifts technology - which could not proceed for a year due to the discovery of two bugs:

1. Occasionally, the lift doors can open while the lift is moving
2. Occasionally, the lift doors do not open after arriving at a floor

The suppliers were also unable to fix the bug despite extended manual debugging. This work was therefore motivated by actual limitations in debugging methodologies for CPS.

2 Overview

Figure 1 shows an overview of our approach. A fault model is learnt for each test oracle developed. Test cases are recovered from the fault model for debugging.

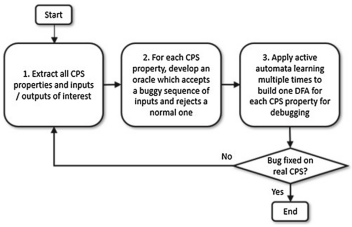


Fig. 1. Overview of approach

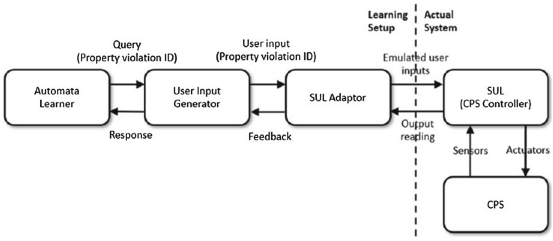


Fig. 2. Automata learning setup

2.1 Step 1 - Develop Oracles

Table 3 provides the system properties which we derive from the system specification and lift standards. The system inputs/outputs of interest are provided in Tables 1 and 2.

Testing requires a pass criteria, known as an oracle. We apply a *derived* test oracle, which categorizes buggy and normal behaviour by reading the outputs of the CPS. These output signals are systematically processed to evaluate real time state variables to determine if each oracle passed or failed. The last column of Table 3 provides the developed oracles corresponding to the system properties. Although this step is always system specific, it can be generalized to other CPSs by gathering the correct output signals and processing them according to the needs of the defined oracles.

2.2 Step 2 - Apply Active Automata Learning

Debugging assumes a fault which is reproducible using a minimal set of inputs. A key contribution of our work is the development of the framework to apply L* to a real system to learn a small DFA of the fault, which is beyond a straightforward deployment of the algorithm. Issues such as the modular design of the learning setup, equivalence query approximation, redundant membership queries and the message passing mechanism were addressed. Figure 2 shows the learning setup, which comprises three main parts - an Automata Learner, a User Input Generator, and a System Under Learning (SUL) Adaptor.

L* learns an unknown DFA using examples and counterexamples made up of an input alphabet Σ. L* will compute and pose membership queries to the SUL to keep an observation table closed and consistent. Once a closed and consistent observation table is achieved, L* generates a hypothesis automaton, for comparison with the actual SUL.

L* requires a *Minimally Adequate Teacher* (MAT) which knows the specification model of the SUL, and is able to exactly answer the equivalence query of whether the hypothesis automata and the SUL are equivalent. In practice, the actual system is the MAT. However, checking whether a hypothesis automaton is equivalent to the actual system is computationally complex. To overcome this,

Table 3. Identified CPS properties

	CPS property	Test oracle		CPS property	Test oracle
1	The lift doors must never be opened while the lift is moving	$(Y6 \vee Y7) \wedge \neg Y2$	6	The correct lift car buttons are always shown at most 1s after an update	Compare Y10, Y11, Y12, Y13 with emulated lift displays
2	Lower the lift must be off at most 1s after the lift reached L1	$(X11 \downarrow \wedge time(\leq 1s)) \wedge Y7 \downarrow$	7	Slow down the lift must be on at most 1s after the slow down sensor is activated, if the current floor is demanded	$((X2 \uparrow \wedge time(\leq 1s) \wedge curr\ flr\ demanded) \wedge \uparrow Y4$
3	Raise the lift must be off at most 1s after the lift reached L4	$(X10 \downarrow \wedge time(\leq 1s)) \wedge Y6 \downarrow$	8	Lower or raise the lift must be off at most 5s after slow down the lift is on	$(Y4 \uparrow \wedge time(\leq 5s)) \wedge (Y6 \downarrow \vee Y7 \downarrow)$
4	The correct floor is always shown at most 1s after an update	Compare Y32, Y33 and Y34 with emulated lift floor	9	If raise or lower the lift is off, it is always done at most 1s after the level sensor is activated	$(Y6 \downarrow \vee Y7 \downarrow) \Rightarrow (X1 \uparrow \wedge time(\leq 1s))$
5	The correct direction of travel is always shown at most 1s after an update	Compare Y30 and Y31 with emulated lift direction	10	Open the doors must be on at most 5s after the raise or lower the lift is off	$((Y6 \downarrow \vee Y7 \downarrow) \wedge time(\leq 5s)) \wedge Y2 \uparrow$

we implemented an approximate way of answering equivalence queries based on depth-bounded search. For each hypothesis automaton, *all* traces up to a maximum number of steps, which we denote as N, from the start state are extracted. In this algorithm, paths which end in an accepting state are not searched any further for new paths.

The User Input Generator will take the needed query from the learner and command the SUL Adaptor to execute the inputs one by one over the configured inter-input duration, τ. This parameter is both SUL and bug dependent. The value of τ must be realistic for actual CPS operation. Certain bugs, especially timing-related ones, can be triggered only if a specific sequence of inputs are injected fast enough. While the query is being executed, at any time, the SUL Adaptor can report back that the property being tested has been violated. In this case, subsequent remaining inputs of the query are not sent to the SUL Adaptor. Otherwise, if no fault has been detected, after the last input of the query has been executed, the SUL Adaptor will wait for the last I/O timeout of Ds. If there is still no fault detected, the SUL Adaptor will notify the User Input Generator that the last query has timed-out without fault.

During development, it was observed that on rare occasions, it is possible for a membership query to return true (or false) in one run but false (or true) in another run. This may be due to slight variances in hardware behavior (espe-

cially timings), or deeper issues in PLC coding, such as double coil syndrome [2]. We recognize that L* works only for deterministic systems. However, we observed that the non-deterministic behavior of our system is rarely observed, and therefore, our approach can still be applied but requires redundant membership queries for reliable model learning. A mechanism is implemented in the User Input Generator to provide more reliable membership query executions. All membership queries are executed at least twice. If both return the same result (both true or both false), it is fed back to the learner for continued model learning. However, if the results differ (one true and the other false), a third execution of the same query is done and its result is fed back to the learner. This reliability is achieved at the cost of doubling the query execution time, but is deemed to be worthwhile because L* will build a totally different model even if just one of the query results is different from a previous run.

The SUL Adaptor implements the actual execution of an input on the PLC, as well as the emulation of the sensor inputs. Note that the emulation is not needed if the actual CPS is being used to test the controller. In the running state, all the PLC outputs are read and the emulated lift states, which are the lift car and door positions, are updated based on these outputs. The property specified by the query received from the User Input Generator shall then be checked for violation. When either events - the needed property violation is detected or the query timed out - the SUL Adaptor will wait for reset input. It shall then inform the User Input Generator accordingly and wait for it to issue a command to execute a reset of the PLC. Once the command has been issued, the SUL Adaptor will reset the PLC and inform the User Input Generator after it is completed, so that the User Input Generator can send the next query. All the SUL Adaptor's relevant internal variables are reset as well, for a fresh cycle of query execution. In summary, Table 4 shows the parameters and their default values used by our framework and all experiments.

2.3 Illustration

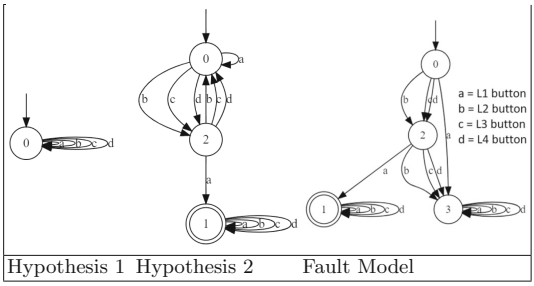
We illustrate our framework using the first bug - that occasionally, the lift doors can open while the lift is moving. Throughout this paper, the default parameter values are $\Sigma = \{a, b, c, d\}$, $N = 3$, $\tau = 0.5\,\text{s}$ and $D = 8\,\text{s}$, as shown in Table 4. The overheads incurred are shown in the last row of Table 11. Building these models required 351 queries which took almost two and a half hours. Table 5 shows the three automata built by L*. These DFAs were learnt by L* maintaining its membership tables, iteratively keeping it closed and consistent, and coming up with a hypothesis automaton when this is achieved. The short test cases generated from this DFA are ba, ca and da. Knowing that the initial state is at L1, and that the delay between inputs $\tau = 0.5\,\text{s}$, the bug can be interpreted as being triggered after an input demanding that the lift move to some floor other than L1 (inputs b,c or d) is set, wait $0.5\,\text{s}$ (the value of τ), and then press L1 (the input a). It can be inferred that the bug is due to a command to move the lift (inputs b, c or d) and the command to open the door (input a) being executed very near in time ($0.5\,\text{s}$). A possible bug fix is to provide an interlock on these

two commands, which was found to fix the bug. Further details about this bug fix can be found in Sect. 4.

Table 4. Parameters

Parameter	Meaning	Default
Σ	Automata Alphabet(set of inputs)	{a,b,c,d}
N	Maximum number of steps from the start state for equivalence query approximation	3
τ	Inter-input duration	0.5 s
D	Last I/O timeout	8 s

Table 5. Hypothesis fault models of Bug 1



Hypothesis 1 Hypothesis 2 Fault Model

3 Implementation

This section provides details of the implementation of our approach to fix bugs found in the miniature lift system.

The overall learning system is run by a mini computer. This device uses the quad-core 64-bit 1.44 Ghz Intel Atom X5-Z8350 processor with 2 GB of RAM. Relay switches were used to switch the user and sensor input terminals on the PLC to 24V (on) and 0V (off), based on the learning traces provided by L*. A simple switch detection circuit was used to capture the PLC's outputs.

We adopt LearnLib [11] to implement our approach. LearnLib is a Java-based framework popular for active automata learning. We built the automata learner on top of LearnLib using Apache Maven [3] in the Eclipse IDE [4]. The parameters Σ and N are used by the Automata Learner. Table 6 shows the implemented alphabet Σ used by the Automata Learner. We have previously justified the use of these button inputs in the *System Description* section. The maximum number of steps from the start state, N, used for our equivalence query approximation, is 3. This is based on our observation that certain bugs in the SUL can already be triggered in two steps - we therefore only require the hypothesis and the unknown automata to be compared up to three steps to be able to get meaningful fault models.

The parameter τ, which is the inter-input duration, is used by the User Input Generator and is both SUL and bug dependent. We selected τ as 0.5 s after some testing, as this value is deemed to be essential to trigger certain bugs in the SUL. In general this is the minimum sampling duration, and in this case, a reasonable approximation for reproducing the bug.

The SUL Adaptor parameter D, which is the last I/O timeout, is used by the SUL Adaptor and is both SUL and bug dependent. We observed that the longest idling duration in the normal PLC operation, meaning that the PLC is no getting any input/output, is about 6 s. This is the period of time that the

door has fully opened, and is waiting for its internal timer to expire, before being commanded to close the door. We need to set D to be longer than this duration to prevent cutting off the PLC's normal operation, and settled on $D = 8$ s.

The PLC has a pre-programmed reset state which shows only L1, provided X1, X3, X4, X5, X10, X16 and X22 are active. For L*, we require a means to reset the PLC before each query and selected the Mitsubishi proprietary MELSEC Communications Protocol (MC Protocol) [6].

We used MQTT as the transport protocol which delivers messages among the Automata Learner, User Input Generator and SUL Adaptor. MQTT is a lightweight protocol designed to be used by Internet-of-Things (IoT) devices. We selected it due to its ease of implementation and support for publish and subscribe. The selected MQTT broker is Mosquitto [7].

4 Research Questions and Experiments

In this section, we shall systematically evaluate the effectiveness and efficiency of our approach. We address the following research questions:

1. How effective is our approach in debugging a real CPS?
2. Does increasing the size of the alphabet Σ increase the time overhead significantly?
3. Can our approach effectively reduce the length of discovered buggy traces?
4. Can our approach find bugs effectively?

It is important to assess the effectiveness of our debugging framework on a real CPS, as a comparison with manual debugging. Studying the relationship between the size of the alphabet and the time overheads provides a practical bound on how many CPS inputs can be used to build fault models in a realistic time frame. Studying the reduction in the length of the discovered buggy traces provides a basis for comparing with normal testing. Finally, the effectiveness of our approach should be studied to prove that our approach can build fault models of bugs of a variety of nature.

As an optimization, a basic emulation of the lift sensor inputs to the PLC was developed to filter away irrelevant inputs caused by reading the actual inputs directly - therefore the emulation does not have to be very precise. The emulated lift's state variables are the car speed and position, door speed and position, buttons state, current floor and motion state.

4.1 How Effective Is Our Approach in Debugging a Real CPS?

We answer this question by using our framework to debug the two observed bugs in the lift controller. These bugs violates Properties 1 and 10 respectively. As a baseline, at least one other property which is not observed to be violated should be included in this experiment, and we randomly pick Property 2. We need to test that our framwork can build a "no fault model" for a specified property, meaning a DFA without any accepting states, if the property is never violated

in all membership queries. If such a model cannot be built for a property, this means that the property is falsifiable based on the experimental parameters - at least one input sequence will cause a property violation.

Table 6. Alphabet Σ

PLC Input	Purpose	Symbol
X17	Lift car L1 button	a
X20	Lift car L2 button	b
X21	Lift car L3 button	c
X23	Lift car L4 button	d

Table 7. Code versions

Code version	Representation
"A"	Supplier Code
"B"	Version "A" patched with the fix for Bug 1 (violation of Property 1)
"C"	Version "B" patched with the fix for Bug 2 (the violation of Property 10)

For this research question, we opted for a reduced alphabet $\Sigma = \{a, b\}$ for simplicity, meaning that we only press the lift car buttons for L1 and L2. This is deemed to be enough to trigger violations of Properties 1, 2, and 10. For clarity, we use a notation to represent the version of the code used for debugging, as shown in Table 7.

We use the notation $M_{code\_version,property}$ to denote the DFA built. For example, $M_{A,1}$ is the DFA built using code version "A" and set with property 1. Table 8 shows the models built.

Table 8. Fault models from code version "A"

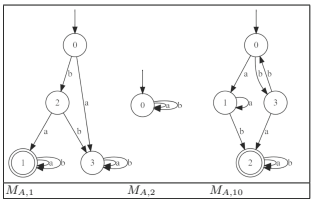
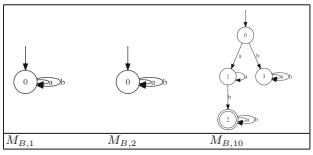


Table 9. Fault models from code version "B"



The regular expression for $M_{A,1}$ is $ba(a|b)*$. Interpreting this expression requires knowing that the fixed time interval between inputs, τ, is 0.5 s. This means that the steps to trigger this violation are: start from the reset state, press L2, wait for 0.5 s, press L1, and thereafter press 0 or more L1 or L2. The violation will be triggered after pressing the first L1. Using this knowledge, we were able to get many relevant, short traces for triggering the violation of Property 1. Moreover, by looking at $M_{A,1}$, it is clear that after the inputs ba, it does not matter how often or what inputs are provided to the system - the bug will be triggered. Knowing that the test starts when the lift is at L1, the input b causes Y6 to be activated (lift rise) while the input a will cause Y2 to be activated (doors open). The bug is patched by adding a check that the lift is not

commanded to move up (Y6) or down (Y7) when it is being commanded to open its doors (Y2). In ladder logic, the symbol -||- represents a check that a device is switched on. The symbol -|/|- represents that it is switched off. Figure 3 shows the bug fix.

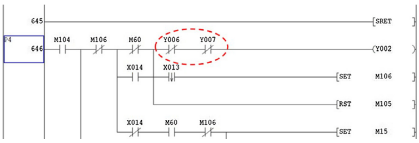


Fig. 3. Patch for Bug 1

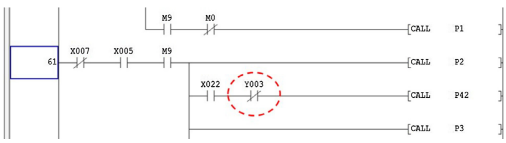


Fig. 4. Patch for Bug 2

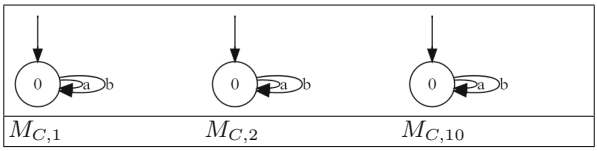
The model $M_{A,2}$ does not have an accepting state and is the "no fault" model. This is expected as the violation of Property 2 was not observed for code version "A". By observation, the controller is in a fault state after the violation of Property 1, ie after the lift door opens while the lift is in motion. This means that after the Bug 1 has occurred, L* will learn a model based on Bug 1 occurring first, followed by Bug 2. Therefore there is no value analysing $M_{A,10}$ at this stage - The model built from violating Property 10 should be analysed after Bug 1 is fixed.

Table 9 shows the models built after applying the patch for Bug 1 (and therefore getting code version "B") and running our framework.

Both $M_{B,1}$ and $M_{B,2}$ are now the "no fault" model. This confirms that code version "B" fixes Bug 1. For debugging Bug 2, which is the violation of Property 10, the reading of internal PLC device values is needed and we used MC protocol. From $M_{B,10}$, its regular expression is $(a+)b(a|b)*$. Bearing in mind that $\tau = 0.5\,\text{s}$, the steps to trigger this violation are: start from the reset state, press L1 at least once, wait 0.5 s, press L2, then press 0 or more L1 or L2. By looking at $M_{B,10}$, it is clear that after the inputs ab, the bug will occur no matter how often or what inputs are provided. There must be some differing occurrence that a, b will cause, compared to another simple word like b, which is clearly rejected by $M_{B,10}$. On deeper analysis, the input a will open the door at L1. The bug happens when b is input 0.5 s after that. The bug does not happen if a does not occur before b. Some internal variable must have been set wrongly after a occurred - leading to the door being unable to open when the lift moved to L2 later on.

We used the discovered test cases to execute buggy runs of the PLC program, as well as some normal runs. For these runs, we used MC Protocol to log the PLC variables deemed needed for debugging. Both sets of logs were compared to identify variances. Analysis of the faulty runs uncovered that the device Y2 (to open the doors) was not activated due to the auxiliary device M104 being off. This was in turn due to the devices M105 and M106 being off, which was due to Y3 (door close) remaining active from L1 to L2, turning off only at L2. We guessed that to fix this bug, we need to add a check that Y3 needs to be off before the subroutine P42 (which moves the lift up or down) is called. Checking

Table 10. Fault models built from code version "C"



that Y3 is turned off before lift movement will turn on M105 and M106, which will turn on M104 and hence allow Y2 to be activated when the lift reaches L2. Figure 4 shows the bug fix.

Table 10 shows the models built after applying the patch for Bug 2 (and therefore getting code version "C") and running our framework:

As can be seen, running our framework on code version "C" yielded $M_{C,1}$, $M_{C,2}$ and $M_{C,10}$ which are all the "no fault" model. This confirms that code version "C" fixed Bug 2. From this effort, we are confident that our approach is able to fix actual CPS bugs.

4.2 Does Increasing the Size of the Alphabet Σ Increase the Time Overhead Significantly?

In order to answer this question, we ran our framework for the two actual bugs with varying alphabet sizes. As explained previously, Code version "A" was used to model Bug 1 while code version "B" was used to model Bug 2. Tables 11 and 12 show the results.

As expected, increasing the alphabet size increases the learning time significantly. As a rule, the choice of inputs should include only the ones which are deemed likely to cause the bug.

Table 13 shows the fault model of Bug 1 built from the respective alphabets, as well as the regular expression representing the model.

4.3 Can Our Approach Effectively Reduce the Length of Discovered Buggy Traces?

To address this question, we apply normal debugging on the two actual bugs. This is done by repeatedly sending inputs randomly picked from $\Sigma = \{a, b, c, d\}$ to the system. The interval between sending the inputs is randomly picked from 0.5 s to 30 s in steps of 0.5 s. These values are selected to simulate normal debugging inputs. When a bug is triggered, the system is reset and the process is repeated. The results of the tests are shown in Table 14.

Comparing these results with the alphabet $\Sigma = \{a, b, c, d\}$ in Tables 11 and 12, there is a reduction in the mean length of buggy queries. For Bug 1, normal debugging had a mean buggy query length of 215.6 while applying our framework required only 3.6. For Bug 2, the same measure was 55.6 for normal debugging and 4.1 for our framework. Therefore, while our framework requires upfront effort

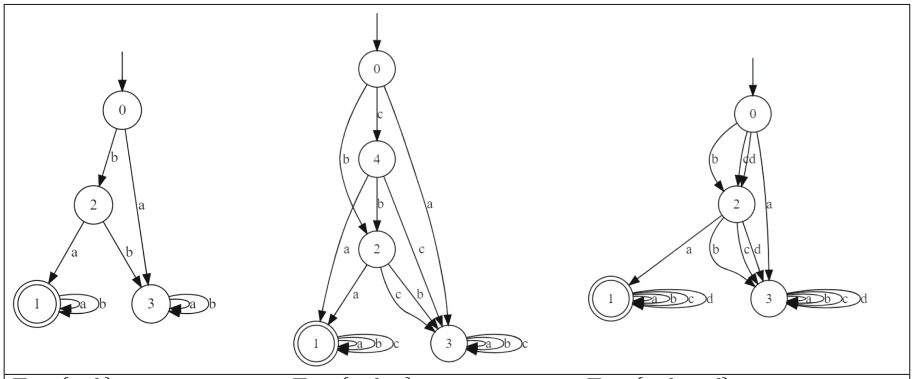
Table 11. Alphabet size and overheads for actual Bug 1

	Σ	# of Queries	# of States	Mean length of buggy queries	Total learning time
1	$\{a, b\}$	29	4	3.71	20.7 min
2	$\{a, b, c\}$	64	5	3.57	1 h 4.3 min
3	$\{a, b, c, d\}$	109	4	3.6	2 h 25.1 min

Table 12. Alphabet size and overheads for actual Bug 2

	Σ	# of Queries	# of States	Mean Length of Buggy Queries	Total Learning Time
1	$\{a, b\}$	32	4	4.25	30.7 min
2	$\{a, b, c\}$	66	4	4.14	1 h 19.7 min
3	$\{a, b, c, d\}$	118	4	4.1	2 h 51.0 min

Table 13. Fault models of Bug 1 built from various alphabet sizes



$\Sigma = \{a, b\}$	$\Sigma = \{a, b, c\}$	$\Sigma = \{a, b, c, d\}$
$ba(a\|b)*$	$(b\|c\|cb)a(a\|b\|c)*$	$(b\|c\|d)a(a\|b\|c\|d)*$

to be set up and tweaked correctly, it can be seen that the reduction in the mean length of buggy queries, as compared to normal, can be significant.

4.4 Can Our Approach Find Bugs Effectively?

The best way to measure the effectiveness of our framework is to apply it to a CPS with many actual bugs which affect the majority of the system requirements. However, despite our best efforts, we found only two actual bugs in the system and they pertained only to the lift and door motion. Therefore, we take code version "C" and mutated it 20 times. We did basic static analysis to ensure that each mutation causes at least one of the identified properties to be violated. This ensures that the mutations affect the majority of system requirements. To ensure that the mutation actually causes a bug, each mutation was tested on the actual system and the expected property was observed to be violated.

The static analysis and mutation were done by identifying a device directly or indirectly related to the bug, and applying some mutation to that device. The following shows the ways mutation was applied:

Table 14. Random testing results

	Bug 1	Bug 2
Number of buggy queries	9	12
Mean length of buggy queries	215.6	55.6
Total time taken	8 h 7 min	2 h 43 min

1. Replacing a device with another device
2. Replacing the "Normally On" device with the "Normally Off" device, i.e. replace -||- with -|/|-
3. Removing a device
4. Adding a new device in series to an existing device
5. Adding a new device parallel to an existing device
6. Change an operator from comparison for equality to comparison for non-equality

We applied our methodology to these mutants. Each run built at least one fault model, making a total of 36 models.

At the minimum, our approach was able to learn the expected fault model of the mutant, proving the effectiveness of our approach in finding bugs. In some instances, more than one fault model was learnt by a mutant - this means that the mutation can trigger more than one violation of the properties. Our approach therefore works as expected, and can find bugs effectively.

5 Related Work

From [8], the concept of Model Based Debugging assumes the existence of a system model which precisely captures the specified system behavior. A fault model is captured by directly observing the system. A comparison of the system model and the fault model will then yield insights into the explanation of the bug. Our work deviates slightly from the established concept - in that we do not have a system model, but rather, for example, a well-known proposition about the system that the lift doors cannot open while the lift is in motion.

From [9], the authors formulated a two-step framework for model based debugging of a PLC - In the first step, the desired, sequence of PLC outputs will be learnt by a Recurrent Neural Network (RNN). From [10], Aral et al. showed that an RNN can model an finite state machine. The captured buggy PLC output will be learnt by another RNN. In the second step, these two RNNs will then be used to train an Artificial Neural Network (ANN) which can then be used to debug the PLC. A small ladder logic diagram representing a clamp, punch and eject manufacturing system was used to demonstrate the concept. This methodology assumes the existence of the correctly specified system model, from which outputs can be recovered so as to build the specification-based RNN.

Marra et al. [12] reported their experience in applying online debugging of a CPS. The Pharo debugger [13] and the author's IDRA [14] remote debuggers

were used to debug a CPS, which is a simple temperature sensing system built by the authors. The authors applied online debugging remotely, i.e. from another machine. They classified remote debugging into "traditional", represented by the use of the Pharo debugger, and "out-of-place", represented by the use of the IDRA debugger. The authors concluded that for this case study, using IDRA is faster than Pharo, at the expense of increased network activity.

6 Conclusion

We believe that bug reproduction is an important first step to debugging, especially for a graphical programming language like ladder logic which makes code step-through a painful experience due to its lack of familiar programming constructs. This paper reports our experience in applying a two-step methodology to determine the minimum sequence of inputs to reproduce a bug, by building a fault model of the system under testing. We believe that this methodology is applicable to other systems of varying nature, provided that the identification of the system properties of interest and the inputs/outputs is done correctly.

Testing the system for bug reproduction can be done either passively or actively, although the latter is preferred because system control brings with it the possibility of triggering the bug faster and expedites data collection. Moreover, system control is mandatory if the bug causes the system to become inoperable after occurrence, and such is the case for our miniature lift.

A simulator such as Safety Critical Application Development Environment (SCADE) with Design Verifier [5] can be explored in future. The combination of using graphical models to capture system logic and a proof assistant provides the possibility of exhaustively proving some system propositions. Active automata learning can also be applied to the system to learn a comprehensive system model. This allows testers to iteratively refine the system model. The resulting model can then be put to use for test case generation or system verification. A comparison of various approaches, such as the use of ANN or graph analysis, to capture a model of a PLC program should be done as well.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
2. Unknown. Double Coil Syndrome In Plc Ladder logic and how to cure it (2017). http://electrodoctrine.blogspot.com/2017/02/double-coil-syndrome-though-sound-like.html. Accessed 4 May 2020
3. Miller, F.P., Vandome, A.F., McBrewster, J.: Apache Maven. Alpha Press (2010)
4. Eclipse Foundation. The Platform for Open Innovation and Collaboration (2020). https://www.eclipse.org/. Accessed 4 May 2020
5. Inc ANSYS. SCADE Suite (2020). https://www.ansys.com/products/embedded-software/ansys-scade-suite. Accessed 4 May 2020

6. Mitsubishi. MELSEC Communication Protocol Reference Manual (2017). http://www.int76.ru/upload/iblock/9c0/q_l_series_reference_manual_communication_protocol_english_controller.pdf. Accessed 4 May 2020
7. Light. Mosquitto: server and client implementation of the MQTT protocol. J. Open Source Softw. **2**(13), 265 (2017). https://doi.org/10.21105/joss.00265. Accessed 4 May 2020
8. Mayer, W., Stumptner, M.: Model-based debugging - state of the art and future challenges. Electron. Notes Theoret. Comput. Sci. **174**(4), 61–82 (2007). ISSN 1571–0661, https://doi.org/10.1016/j.entcs.2006.12.030
9. Abdelhameed, M.M., Darabi, H.: Diagnosis and debugging of programmable logic controller control programs by neural networks. In: IEEE International Conference on Automation Science and Engineering, Edmonton, Alta, pp. 313–318 (2005). https://doi.org/10.1109/COASE.2005.1506788
10. Aral, K., Nakano, R.: Adaptive scheduling method of finite state automata by recurrent neural networks. In: Proceeding of the Fourth International Conference on Neural Information Processing, Dunedin, New Zealand, pp. 351–354 (1997)
11. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next generation learnlib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_18
12. Marra, M., et al.: Debugging cyber-physical systems with pharo: an experience report. In: Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies (IWST 2017), p. 10. ACM, New York, Article 8 (2017). https://doi.org/10.1145/3139903.3139913
13. Pharo. Pharo: The Immersive Programming Experience. Pharo Board. https://pharo.org/web. Accessed 23 Sept 2019
14. Marra, M.: IDRA: An Out-of-place Debugger for non-stoppable Applications. http://soft.vub.ac.be/Publications/2017/vub-soft-ms-17-01.pdf. Accessed 23 Sept 2019