Testing Safety PLCs Using QuickCheck

David Thönnessen[1], Nick Smallbone[2], Martin Fabian[2], Koen Claessen[2], and Stefan Kowalewski[1]

Abstract— **The testing of safety-related industrial systems is usually carried out by means of checklists. A tester has a list of scenarios that he or she manually applies to the system to check whether the system behaves according to its specification. However, operators behave unpredictably. Their behavior may not be covered by the set of scenarios tested and may lead to dangerous situations. To avoid this, randomized test case generation can be useful as it allows for unanticipated scenarios. The presented framework uses a tool for randomized test case generation, QuickCheck, to trigger event sequences that are then applied to a Safety Programmable Logic Controller (Safety PLC). Experiments show that this concept is capable of finding errors in safety code or increasing the tester's confidence in the correctness of the code by exhibiting a large number of passing test cases. While this concept proves to be powerful, it does not require much effort from the tester as the execution of test cases is done without user interaction.**

I. Introduction

The main contribution of the presented approach is to improve the process of testing safety-related parts of a plant. Virtual commissioning is one method of testing industrial controllers before setting them into operation [1], [2]. In previous work, we specialized Hardware-in-the-Loop (HiL) [3], [4] simulation to testing of PLCs [5]. This work applies our approach to safety controllers. Our scenario consists of a Safety PLC that supervises a set of safety-related sensors in a plant, such as emergency buttons, door switches, and light barriers.

One of the key issues with testing Safety PLCs is that most manufacturers use checklists as the main testing procedure. Such a checklist consists of a set of actions, which are applied to the plant manually by the tester, such as "open the front door". The tester checks if the safety equipment behaves according to a given specification and declares a system to be safe if all test cases pass without failure. Our motivation for improving this process is as follows: First, the checklist only contains those scenarios that a tester has thought of, i.e. there might be scenarios that are potentially hazardous but never tested. Secondly, the test cases defined in the checklist are applied and supervised only by hand, i.e. the tester might make a mistake and miss important details during testing. Finally, safety-related systems are meant to protect operators working in the surroundings of a plant. However, operators behave unpredictably so that writing test cases covering their behavior is a non-trivial task.

As a solution for this, we consider randomized generation of test cases [6], [7]. For the generation process, the tester has to define a test suite with so-called atomic events. An atomic event is an action that changes the state of one safety component of the System Under Test (SUT), for example opening a door. Another atomic event closes the same door again. Given atomic events for a plant, the test case generator creates randomized sequences of those, which can then be executed on the SUT.

Our approach is designed in a way that all possible real-world scenarios could be generated. Consequently, the more test cases are generated and executed, the higher the tester's confidence in the system not containing any errors will be. We will show the capabilities of our approach using a real-world plant as a use case in Section IV.

This paper describes how test cases are generated and executed. Test results are evaluated using the framework previously described in [5]. In this framework, test cases are evaluated using an *oracle*, which carries out the evaluation following an Input-Output Conformance (IOCO)-based approach [8], that is, having acceptance criteria that calculate reference signals for a SUT and a so-called *supervisor* that compares these signals with the actual signals of the SUT [9], [10]. In related work, we apply our approach to on-the-fly IOCO testing of Safety PLC code [11] using Supervisory Control Theory [12].

In the test case generation process, we use a so-called *fuzzer* [13], [14], [15] to come up with a randomized sequence of given events. Fuzzing is a technique of automated software testing where a program is provided with randomized data as an input and monitored for unintended behavior. Fuzzers can generate an initial test case to execute and, if the test fails, reduce the complexity of the failing test case in order to support the tester with diagnosis [16], [17].

II. Test Case Generator

The fuzzer we use is called QuickCheck [18]. Testing a piece of software usually involves coming up with test cases and defining the expected outcome of each test case. In QuickCheck, the tester instead gives a general specification of how the software should act in response to an unknown test case. QuickCheck then automatically generates a large set of randomized test cases and checks them against the specification. Any failing test case is simplified as far as possible and then reported to the tester. In more detail, QuickCheck works as follows:

1) Generate a random test case.

[1]RWTH Aachen University, Informatik 11 – Embedded Software, Ahornstraße 55, 52074 Aachen, Germany, {thoennessen, kowalewski}@embedded.rwth-aachen.de

[2]Chalmers University of Technology, Gothenburg, Sweden, {nicsma, fabian, koen}@chalmers.se

This work was supported by a fellowship within the FITweltweit programme of the German Academic Exchange Service (DAAD), and by the Swedish Research Council (VR) grant 2016-06204, Systematic testing of cyber-physical systems (SyTeC).

2) Execute the test case and check its result against the specification.
3) If the test case fails, simplify it as far as possible.
4) Otherwise, repeat from step 1.

This remainder of this section describes how random test cases are generated and how failing test cases are simplified. We have specialized both of these processes for PLC testing. Section III describes how the generated test cases are executed.

A. What is a test case?

A PLC has a collection of Boolean inputs, each of which may be set to low or high at each execution of its scan cycle [19]. In this context, inputs denote input signals of the SUT (sensors) which are generated by the HiL simulator. In principle, a test case should define the value of each input at each PLC cycle. However, to be able to print test cases compactly, we instead represent a test case as a list of *timed events*. Each timed event consists of an event $e \in E$, which is either setting an input to low, setting an input to high, or doing nothing, followed by a delay $d \in D$. $D := [d_{min}, d_{max}]$ is the range the delay can be in, i.e. there is a minimum and maximum delay for the generator. Once an input is set to high, it remains high until a subsequent event sets it low, and vice versa.

The following is an example of a list of timed events:

```
input 1 high, wait 500 ms
do nothing,   wait 300 ms
input 3 high, wait 200 ms
input 2 high, wait 800 ms
input 1 low,  wait 200 ms
input 3 low,  wait 500 ms
```

Input 1 is set to high first. After $500\,\mathrm{ms}$ there is a "do nothing" event followed by a further delay of $300\,\mathrm{ms}$, after which input 3 is set to high. After another $200\,\mathrm{ms}$ input 2 is set to high. After $800\,\mathrm{ms}$, input 1 is reset to low. After $200\,\mathrm{ms}$, input 3 is set to low again. The execution then waits $500\,\mathrm{ms}$ before ending the test.

B. Generating good test data

In order to get good test coverage, it is vital to get a good distribution of random data when generating test cases. One approach would be to generate data uniformly at random: each input would be assigned a fresh random value on each PLC cycle. This approach would not work well, because all inputs would be constantly changing, and any behavior requiring an input to be held constant ("if the emergency button is held down for 1 second, then…") would have little chance of being tested.

We can solve this problem by generating a random list of timed events, such as the one above. Even if we choose the events and the delays uniformly at random, inputs will often be held constant for a while because only a few inputs change at a time.

There are some other properties we would like our test data to have that are not satisfied by a uniform distribution. Firstly, we should often change several inputs simultaneously, i.e., delays of $0\,\mathrm{ms}$ should be overrepresented in the generated test cases. Secondly, we should only set an input to high if it is currently low, and vice versa. Finally, it takes a few seconds to reset the PLC after each test case, so we should generate test cases that are several seconds long.

For our case study, there is one property that is more desirable. Many of the inputs correspond to sensors that shut down the plant when triggered; we should not leave too many sensors triggered on average, otherwise the plant will be constantly shut down. For that reason, we allow the tester to identify certain events as *negative*; all other events are called *positive*. Negative events are those events that tend to cause the safety controller to shut down the system or parts of it. An example of a negative event might be opening a door, which shuts down some machinery to protect the human that possibly enters. Note that the inverse of a negative event (in this case, closing the door again) is positive.

Our test data generator captures all the requirements above and works as follows. First, it generates a random number, which represents the total number of events that the test case should contain. This number is chosen to be between 50 and 150, so that we get reasonably long test cases.

Next, the delay of each event is chosen. To capture the idea that inputs may change simultaneously, we choose each delay to be $0\,\mathrm{ms}$ with $50\,\%$ probability. Otherwise, the delay is uniformly chosen from the delay range D.

Finally, we have to choose events to accompany the selected delays. Each event either changes the value of an input or does nothing. The choice of events is independent of the delays. The mechanism we use to choose an event is a QuickCheck feature called *weighted choice*. A weighted choice takes a list of alternatives $x_1, \ldots, x_n$ and a list of associated weights $w_1, \ldots, w_n$. It then makes a random choice between the alternatives, choosing alternative x_i with a probability $P(x_i)$ that is proportional to w_i:

$$P(x_i) := \frac{w_i}{\sum_{i:=1}^{n} w_i}$$

Events with higher weights are thus more likely to be chosen.

The events that we choose from are doing nothing, and *changing* the value of an input, i.e. setting an input to high that is currently low or vice versa. In order to avoid triggering too many negative events, we choose the following weights, based on empirical observations: doing nothing has weight $w_{nothing} = 1$, triggering a negative event has weight $w_{neg} = 2$, and triggering a positive event has weight $w_{pos} = 10$. For example, suppose that there are 10 inputs i_1 to i_{10}, that setting any input to high is a negative event, and that i_1 is currently high and the others are low. We choose between resetting i_1 to low (with weight $w_{pos} = 10$), setting one of the other inputs to high (each with weight $w_{neg} = 2$), or doing nothing (with weight $w_{nothing} = 1$). The weights sum up to 29 so we reset i_1 to low with probability $10/29$, set one of the other inputs to high with probability $2/29$ each, or do nothing with probability $1/29$.

This choice of weights results in about one sixth of negative events on average being active at any given time, which captures the requirement that not too many doors or emergency buttons be opened/pressed at the same time.

The test data generator itself is generic and works for any PLC code, but the particular weights are chosen for our case study, and might not be appropriate for every application. The weights are important: if we set $w_{pos} = w_{neg} = 1$, most test cases make the plant shut down and not start up again. It would be straightforward to let the tester control the weights, or to set them automatically given e.g. a target number of inputs that should be simultaneously high. Furthermore, the tester may want to state assumptions about what sort of test data is physically realistic or reasonable; future work is to make the test data generator configurable in this way.

C. *Shrinking of test cases*

Our test data generator produces test cases of between 50 and 150 events, which are too long to be easily understood by humans. When a test case fails, QuickCheck therefore tries to reduce it to a minimal failing example, a process called *shrinking* [20].

The way shrinking works in general is as follows. Given a failing test case, QuickCheck generates various simpler test cases. Each one of these simpler test cases is then run, and if it fails, it replaces the original test case. The process is then repeated: simpler variants of the new test case are generated and tried. When the process stops, we have a failing test case, such that simplifying it in any way results in a passing test case. Thus, this is a minimal failing test case, and it is reported to the tester.

To customize the behavior of shrinking, the user provides rules to QuickCheck that, given a failing test case, produce a set of simpler test cases to try. We have defined rules that are appropriate for shrinking a list of timed events.

The first rule is about removing all events from the list that are not necessary to provoke the failure. QuickCheck is already able to shrink a list of items by trying to remove each element of the list in turn. We keep this shrinking rule for the timed event list. If we assume that the example on page 2 is a failing test case, then having only this rule for shrinking might result in the following failing test case:

```
input 1 high, wait 500 ms
input 3 high, wait 200 ms
input 2 high, wait 800 ms
input 1 low,  wait 200 ms
```

Now suppose that the event “input 3 high” is not necessary for the failing test case, but that we need to have a $500\,\text{ms}+200\,\text{ms} = 700\,\text{ms}$ delay between “input 1 high” and “input 2 high”. In that case, “input 3 high” will not be removed from the test case, even though it is not really needed. We fix this by adding two more shrinking rules: any event can be replaced with “do nothing”, and a “do nothing” event e_i can be absorbed into the preceding event e_{i-1}, adding the delays of the two events such that $d'_{i-1} := d_{i-1}+d_i$. Given a failing test case, shrinking tries all possible ways in which these

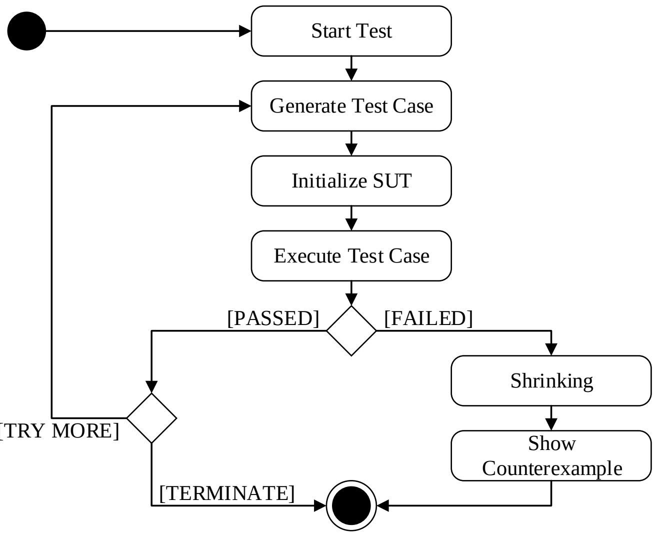

Fig. 1. Execution of Multiple Test Cases

rules could apply. Now the original test case would shrink to:

```
input 1 high, wait 700 ms
input 2 high, wait 800 ms
input 1 low,  wait 200 ms
```

Finally, we also try shrinking the delays. We try replacing each delay with a smaller one, trying all multiples of $100\,\text{ms}$ up to but not including the current delay. This may give us the minimal test case:

```
input 1 high, wait 700 ms
input 2 high, wait 300 ms
input 1 low,  wait 100 ms
```

An important aspect of shrinking is that, when the final failing test case is reported to the tester, it is guaranteed to be minimal. In our case, this means that no event-delay pair can be removed from the test case, no event can be replaced with a “do nothing” event, and the delays are as small as possible, within a tolerance of 100ms. Because of shrinking, the tester can be sure that every single part of the test case is important. This often makes it simpler to diagnose faults.

III. Test Case Execution

We use the HiL testing framework as presented in [5], [9], [10] to apply test cases to a SUT. HiL testing allows incorporating the control hardware into the testing process, which is beneficial for safety testing as it incorporates more components of the control system than other approaches like Model-in-the-Loop or Software-in-the-Loop. The testing framework includes an oracle which comes with an IOCO based analysis [8], [21], comparing the actual behavior of the SUT against a given specification; refer to [5] for more details. In the context of this paper, the details of the HiL testing framework are not important. It is used to execute test cases as described in this section and delivers the test result to QuickCheck to be processed.

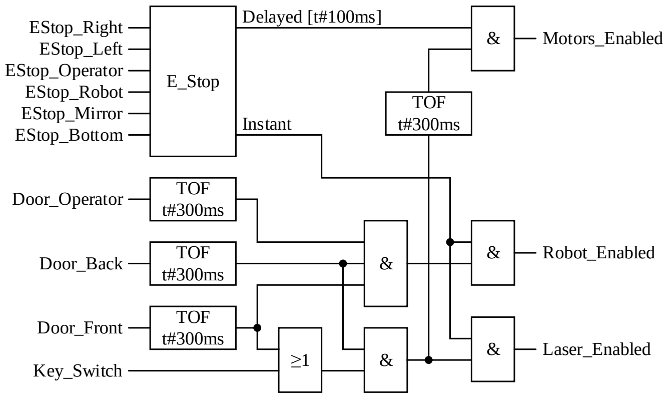

Fig. 2. Simplified Safety Code of Use Case Plant

The execution of test cases is illustrated in Figure 1. The tester initiates a test run. QuickCheck then generates a test case as described in Section II, i.e. consisting of 50 to 150 events. The test case is transferred to the HiL framework, which has to initialize the SUT first.

Initialization consists of resetting the SUT and transferring it into a state-to-test, which is defined by the tester. The state-to-test is usually a state of the control system in which it is in normal operation such that events like pressing an emergency button have an effect on the system. The time consumed by this step heavily depends on the control system being tested.

After initialization, the test case is executed by applying the generated events one-by-one to the SUT while obeying the delay associated with each event. As soon as the test case execution has finished and the oracle has finished evaluating, the result is passed to QuickCheck.

If the test passed, i.e. no failures occurred with respect to the given specification, either more test cases can be generated and applied or the tester can terminate the test.

If a test case failed, the actual behavior of the SUT is inconsistent with the specification. QuickCheck starts its shrinking process as described in Section II-C (not shown in Figure 1 due to lack of space). After the shrinking process is completed, the found counterexample, i.e. the minimal test case leading to a failure, is shown to the tester for further diagnosis. With this step, the test case execution terminates.

This methodology is an instance of black-box testing [22], since the testing framework has no information about the control code of the SUT. Consequently, the code must be inspected manually to find the location of the failure, but the failing test case hints at where the problem is.

IV. Use Case

We utilize a real-world plant as a use case for our approach. We test the safety code running on this real plant against a specification using generated test cases.

The plant consists of a laser, a robot and several motors. It processes fibers, which are supplied by the robot and then processed by the laser. Motors are used to position the fibers. As the plant operation has a potential risk for operators, it is enclosed by a fence. Three doors serve as entrances to provide the robot with raw fibers and to allow maintenance. Six emergency buttons are distributed around the plant to allow an operator to stop the operation in case of an emergency. A key switch allows the operator to open the front door without shutting down the laser for maintenance purposes. All doors, emergency buttons and the key switch are realized using two-channel, redundant signal wires but considered as one signal in this paper. Figure 2 illustrates a simplified version of the safety code, implemented in Function Block Diagram. This simplified version was used for the implementation of the safety code.

Pressing any emergency button stops the robot and laser immediately and the motors after $100\,\mathrm{ms}$. Opening the operator's door stops the robot after $300\,\mathrm{ms}$. Opening the back door stops the robot and the laser after $300\,\mathrm{ms}$ and the motors after $600\,\mathrm{ms}$. Opening the front door stops the robot after $300\,\mathrm{ms}$. Furthermore, it stops the laser after $300\,\mathrm{ms}$ as well and the motors after $600\,\mathrm{ms}$ if the key switch was not turned on (to maintenance mode). According to the manufacturer's information, the given Timer Off-Delay (TOF) blocks are not required for safety purposes but to compensate for inaccurate (bouncing) signals coming from the door switches. Please note that the real plant delivers the emergency signals to the non-safety PLC as well. In case of an emergency the control code tries to shut down laser, robot and motors softly within the time windows, i.e. $300\,\mathrm{ms}$ or $600\,\mathrm{ms}$. If the PLC does not shut down all actuators within these time windows, the Safety PLC shuts down the actuators by emergency, using their emergency ports and switching off their power supplies.

To execute tests on this plant, we were provided with the simplified safety code as described above as well as the actual safety code. Due to PLC incompatibilities, we needed to re-implement the actual safety code in the development environment we use. The acceptance criterion is that the simplified and actual safety code behave the same modulo small timing differences. The HiL testing framework [5] provides an oracle that checks this acceptance criterion.

The test setup consisted of a PLC running the safety code, considered as the SUT in the following, a HiL simulator [5] including the oracle, and a computer running QuickCheck with the described test case generator. Test cases were executed by the HiL simulator by resetting and initializing the SUT before each test case and then applying the given event sequence as described in Section III.

A. *Found bugs*

This section describes two bugs found by our approach.

The first bug is rather trivial and was due to an implementation error during the re-implementation of the real safety code. The real safety code does not contain this error. That it was found in the very first test case execution demonstrates its triviality.

The initial counterexample consisted of 50 events and was automatically simplified to the following test case:

```
EStop_Right press, wait 100 ms
```

This means that pressing the emergency button on the right

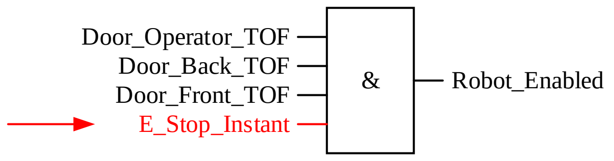

Fig. 3. The Function Block implementation of the Robot_Enabled signal. The missing signal connection of Bug 1 is marked.

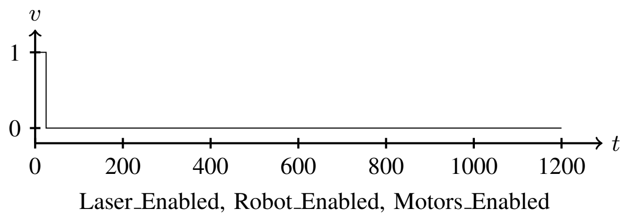

Fig. 4. Reference Signals of Bug 2

side and waiting 100 ms leads to unwanted behavior. Inspecting the system state after the test case execution showed that the robot was not shut down although it was specified to be.

We traced the cause of the failure back to a missing signal connection in the safety control code as shown in Figure 3. The figure shows one And-block of the safety control code. The block was only connected to the door-states (after their respective TOF blocks) but not to the emergency stop unit `E_Stop`. Adding this connection led to 15 passed test cases with an average event sequence length of 62 events.

The second bug was discovered in the 16th test case and simplified to the following counterexample:

```
EStop_Right press,   wait 300 ms
Door_Back open,      wait 100 ms
EStop_Right release, wait 100 ms
```

This bug is not trivial and needs some extra explanation about the way the acceptance criteria are specified. Regarding the simplified safety code given in Figure 2, pressing any emergency button or opening any door should lead to a system shutdown after at most 600 ms (we neglect the key switch, as it is not part of the failed test case). The acceptance criteria

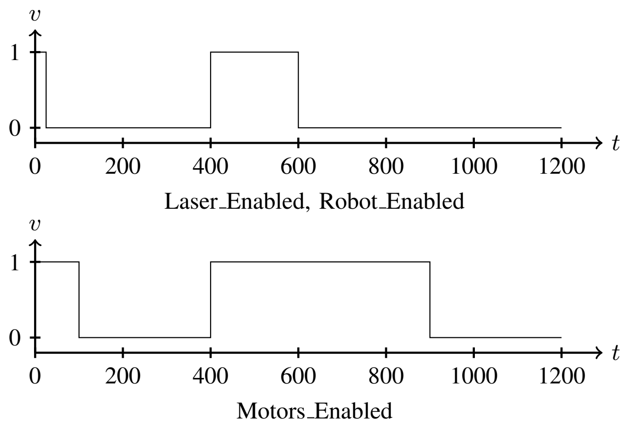

Fig. 5. Actual Signals of Bug 2

ignore the timers given in the safety code (see Figure 2) as they are implementation-specific. Consequently, it expects the SUT to shut down all related actuators right away. As this is not practicable and not required for safety, we replaced the maximum possible shutdown delay of 600 ms by a constraint saying that all related actuators have to be shut down at most 800 ms (including 200 ms time buffer) after triggering shutdown. The timespan 800 ms is the specified maximum time for the system to be shut down in case of an emergency. We realized this constraint by using a signal change tolerance of 800 ms of the HiL testing framework as defined in [9]. In simple terms, this tolerance means that a signal must change to the value of the reference signal within 800 ms. In other words, the plant has to shut down at the latest 800 ms after a critical event took place. The resulting reference signal for the given failed test case is shown in Figure 4. The x-axis shows time t in ms and the y-axis value $v \in \{0, 1\}$. As the emergency button `EStop_Right` is pressed at the very beginning of the test case, all three reference signals `{Laser, Robot, Motors}_Enabled` are set to 0.

The actual signal is stated in Figure 5 with the same axes as Figure 4. Signals `{Laser, Robot}_Enabled` are equivalent and therefore combined in one diagram. Coming back to the failed test case, the first event `EStop_Right press` shuts down the laser and robot right away and the motors after 100 ms. The evaluation detects correct signal changes to the reference signal at $t = 100$ ms and would not allow the signals to turn on again. At $t = 300$ ms event `Door_Back open` takes place. This event has no effect on the signals `{Laser, Robot, Motors}_Enabled` at this point in time. However, they start operation of the TOF blocks of the safety control code (see Figure 2), meaning that they would shut down devices after 300 / 600 ms (laser, robot / motors) if they were not shut down already. At $t = 400$ ms event `EStop_Right release` takes place. The reference signal stays constant because although the emergency button is released, the back door is still open. Therefore, the devices should remain shut down according to the acceptance criteria. However, the safety control code allows all three devices to operate again as they are not shut down by the (now released) emergency button anymore and the shutdown from the opened door is still pending for 200 / 500 ms.

This is an inconsistency between the implementation and the acceptance criteria and leads to failing the test case. Nevertheless, the actual implementation does conform to the simplified implementation but not to our acceptance criteria. While one could think that the acceptance criteria are just wrong, the key is that the safety code would allow the system to operate for 200 / 500 ms even though there was an emergency and a door to the plant is open. We consider this as a fault and recommend redesigning the control code in a way that it does not allow system operation in any state where an emergency button is pressed or a door is open.

V. Conclusion

The presented testing framework showed in the given use case that it is capable of finding both trivial and complex errors. We use this section to highlight and discuss the characteristics of the approach before we conclude the paper.

A. Finding bugs

Our approach performed well in finding bugs for our use case. The presented bugs were found after 1 and 16 test case executions, respectively. We were surprised that the second bug was found after only a few test cases.

It cannot be calculated or estimated how long it takes to find an error, which however is not the focus of randomized testing. The focus is on coming up with test scenarios that a tester would not come up with. Accompanied with the automatic generation and execution of these test scenarios it is possible to execute a large amount of tests without user interaction. Nevertheless, the given approach should be considered as an addition to existing testing methods, such as checklists or virtual commissioning.

B. Costs

Costs are an important factor for testing techniques and can be split up in costs for user interaction (by a tester) and the overall amount of time consumed. The effort a tester has to put into testing using the presented approach is to set up the testing environment and specify the input signals (see Section II) and the acceptance criteria that are used for evaluation. After starting execution of the test suite our framework operates without user interaction as long as it has not found a bug. If a bug was found, the approach applies the described shrinking process until the minimal failing event sequence is found and terminates. The tester then has to debug the control code and start a new testing sequence.

C. Diagnosis of failed test cases

Shrinking down test cases is an essential part of the concept, as test cases are usually too complex for a tester to understand, i.e. 50 or more events. The shrinking process works automatically and tries to first remove unnecessary events out of the sequence and then reduce the timings of single events. The result is a minimal test case leading to a failure, which is, depending on the complexity of the error, much easier to understand for a human than the original failed test case. It has to be noted that a shrunken test case does not necessarily provoke the *same* fault as the original failed test case (if the control code contains more than one fault). Since our approach is black-box, our testing framework has no knowledge about the safety control code. It relies on the oracle and does not get more feedback than a “passed” or “failed” per executed test case.

D. Future work

In future work our approach should be compared to existing testing methods applied to safety control code regarding time consumption and error detection rate. The test case generator we designed can be improved to generate better test cases using probabilities for groups of events (see Section II-B). Tuning these probabilities by observing the output state of the system is one interesting aspect and should be considered in future work.

References

[1] C. G. Lee and S. C. Park, “Survey on the virtual commissioning of manufacturing systems,” *Journal of Computational Design and Engineering*, vol. 1, no. 3, pp. 213–222, 2014.

[2] M. Dahl, K. Bengtsson, P. Bergagård, M. Fabian, and P. Falkman, “Integrated virtual preparation and commissioning: supporting formal methods during automation systems development,” *IFAC-PapersOnLine*, vol. 49, no. 12, pp. 1939–1944, 2016.

[3] F. Gu, W. S. Harrison, D. M. Tilbury, and C. Yuan, “Hardware-in-the-loop for manufacturing automation control: Current status and identified needs,” in *Proceedings of the 3rd IEEE International Conference on Automation Science and Engineering, IEEE CASE 2007*, 2007, pp. 1105–1110.

[4] B. Lu, W. McKay, S. Lentijo, A. Monti, X. Wu, and R. Dougal, “The real time extension of the virtual test bed,” in *Huntsville Simulation Conference*, 2002.

[5] D. Thönnessen, N. Reinker, S. Rakel, and S. Kowalewski, “A concept for PLC hardware-in-the-loop testing using an extension of Structured Text,” in *Emerging Technology and Factory Automation (ETFA)*. IEEE, 2017, pp. 1–8.

[6] R. Hamlet, “Random testing,” *Encyclopedia of software Engineering*, 2002.

[7] J. W. Duran and S. C. Ntafos, “An evaluation of random testing,” *IEEE transactions on Software Engineering*, no. 4, pp. 438–444, 1984.

[8] J. Tretmans, “Test generation with inputs, outputs and repetitive quiescence,” *Software-concepts and tools*, vol. 17, no. 3, pp. 103–120, 1996.

[9] D. Thönnessen, S. Rakel, N. Reinker, and S. Kowalewski, “Matching discrete signals for hardware-in-the-loop-testing of PLCs,” in *Conference on Embedded Systems, Computational Intelligence and Telematics in Control (CESCIT)*. IFAC, 2018, pp. 1–8.

[10] D. Thönnessen, N. Reinker, S. Rakel, A. Svetlakov, and S. Kowalewski, “Correctness properties and exemplified applicability of a signal matching algorithm with multidimensional tolerance specifications,” in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2018, pp. 1197–1202.

[11] A. Khan, D. Thönnessen, and M. Fabian, “On-the-fly conformance testing of safety PLC code using QuickCheck,” in *International Conference on Industrial Informatics (INDIN) Industrial Applications of Artificial Intelligence*. IEEE, 2019, to be published.

[12] P. J. Ramadge and W. M. Wonham, “Supervisory control of a class of discrete event processes,” *SIAM journal on control and optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[13] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[14] P. Godefroid, M. Y. Levin, and D. Molnar, “SAGE: whitebox fuzzing for security testing,” *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[15] S. K. Cha, M. Woo, and D. Brumley, “Program-adaptive mutational fuzzing,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.

[16] M. Zalewski, “American fuzzy lop,” 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[17] G. Misherghi and Z. Su, “HDD: hierarchical delta debugging,” in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 142–151.

[18] K. Claessen and J. Hughes, “QuickCheck: a lightweight tool for random testing of Haskell programs,” *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.

[19] H. Berger, *Automatisieren mit STEP 7 in AWL und SCL: speicherprogrammierbare Steuerungen SIMATIC S7-300/400*. Publicis Publ., 2011.

[20] J. Hughes, “QuickCheck testing for fun and profit,” in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2007, pp. 1–32.

[21] C. Gregorio-Rodríguez, L. Llana, and R. Martínez-Torres, “Input-output conformance simulation (iocos) for model based testing,” in *Formal Techniques for Distributed Systems*. Springer, 2013, pp. 114–129.

[22] S. Nidhra and J. Dondeti, “Black box and white box testing techniques-a literature review,” *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.