SEARCH-BASED AUTOMATED PROGRAM REPAIR OF CPS CONTROLLERS MODELED IN SIMULINK-STATEFLOW

Aitor Arrieta

Mondragon University Mondragon, Spain aarrieta@mondragon.edu

Pablo Valle

Mondragon University Mondragon, Spain pvalle@mondragon.edu

Shaukat Ali

Simula Research Laboratory Oslo Metropolitan University Oslo, Norway shaukat@simula.no

ABSTRACT

Stateflow models are widely used in the industry to model the high-level control logic of Cyber-Physical Systems (CPSs) in Simulink-the defacto CPS simulator. Many approaches exist to test Simulink models, but once a fault is detected, the process to repair it remains manual. Such a manual process increases the software development cost, making it paramount to develop novel techniques that reduce this cost. Automated Program Repair (APR) techniques can significantly reduce the time for fixing bugs by automatically generating patches. However, current approaches face scalability issues to be applicable in the CPS context. To deal with this problem, we propose an automated search-based approach called FLOWREPAIR, explicitly designed to repair Stateflow models. The novelty of FLOWREPAIR includes, (1) a new algorithm that combines global and local search for patch generation; (2) a definition of novel repair objectives (e.g., the time a fault remained active) specifically designed for repairing CPSs; and (3) a set of mutation operators to repair Stateflow models automatically. We evaluated FLOWREPAIR with three different case study systems and a total of nine faulty stateflow models. Our experiments suggest that (1) FLOWREPAIR can fix bugs in stateflow models, including models with multiple faults; (2) FLOWREPAIR surpasses or performs similarly to a baseline APR technique inspired by a well-known CPS program repair approach. Besides, we provide both a replication package and a live repository, paving the way towards the APR of CPSs modeled in Simulink.

Keywords Automated Program Repair · Cyber-Physical Systems · Simulink · Stateflow

1 Introduction

Cyber-Physical Systems (CPSs) integrate digital cyber computations with physical processes [1]. As these systems involve complex and expensive hardware, their software is usually tested through simulation-based testing in early stages of development [2, 3, 4, 5, 6, 7]. MATLAB/Simulink has been the de-facto CPS modeling and simulation tool [2, 3], allowing complex features like automated test generation and standard-compliant code generation. The control of CPSs is usually divided into high-level decision-making components and low-level controllers [8]. The high-level decision-making controllers are typically delegated to Stateflow models when the CPS is modeled in Simulink. Stateflow is a graphical language for modeling state charts. It is common to find logic bugs in Simulink models involving Stateflow models. While many approaches have been focusing on automated techniques to generate test cases for Simulink [2, 3, 9, 10, 11], once a fault has been detected, the repair process has remained manual. This poses a high cost to engineers, and therefore, it is necessary to develop novel techniques for the automated program repair (APR) of Simulink models.

Recently, the area of APR has significantly grown, proposing novel techniques to automatically repair software bugs in different languages (e.g., Java, C, Python). These approaches encompass different techniques, such as search-based [12, 13], semantic-based [14], and neural-machine based repair approaches [15]. More recently, the APR community has focused on using Large Language Models (LLMs) [16, 17] and conversational techniques (i.e.,

ChatGPT) [18] for APR, with impressive results. However, repairing CPSs' software has two core challenges that limit the applicability of existing APR approaches to CPSs:

Challenge 1 – Long test execution time: When testing CPSs, test cases need to be executed at the system level employing simulation-based techniques [19, 20]. The physical system and the software control need to be integrated because the physical outcomes affect the software and vice-versa [20]. This is commonly known as online testing [21, 22]. These physical systems are usually modeled through complex mathematical models that are computationally expensive, and often require rendering 3D images, which increases test execution time [21, 22]. For instance, previous APR approaches targeting industrial CPSs report average test execution times of around 5 minutes [19] and 20 to 30 minutes [20], for different industrial case study systems. Therefore, unlike most APR approaches, which employ unit test cases that are executed in milliseconds, in the context of CPSs, the execution time poses limitations on the sizes of test suites used for repair [20].

Challenge 2 – Inadequate repair objectives: Most existing repair objectives focus on high-level passing and failing of test cases to guide the search [23]. The problem with CPSs is that many test cases cannot be executed due to the aforementioned challenge. On the other hand, it is common that test cases for CPSs are generated with a falsification-based approach (e.g., [10, 24]). This means that when a system requirement is violated, the test generation process stops, returning a single failure-inducing test input. This limits the guidance of search-based APR approaches, converting the search process into a random one due to the flat fitness landscape. As a result, novel repair objectives need to be investigated.

To address the above-mentioned challenges, we instead, propose a search-based APR method for Stateflow models, paving the way towards the APR of CPSs. Specifically, we solve the first issue by proposing an iterative algorithm that handles an archive of partial patches instead of a population-based search algorithm. The key idea behind the algorithm is to use a global search algorithm to explore different kinds of patches at different parts of the code. When a partial patch is found (i.e., a patch that enhances the buggy code but does not fix it entirely), the global search is swapped by a local search process that exploits patch generation in a restricted Stateflow area. As a result, we solve the long execution time of the population-based search algorithms. The second issue is solved by redefining the repair objectives. Specifically, instead of focusing on the number of pass and fail test cases to guide the search, we define three novel repair objective functions that consider CPS particularities, such as the notions of time and failure severity. In summary, our paper makes the following contributions:

- **Method:** We propose a novel method to repair faults in Stateflow models that control CPSs. This method encompasses a novel search algorithm, which integrates global and local search and adopts novel repair mutations for Stateflow models. In addition, we propose three new repair objectives that are generic for any kind of APR tool for CPSs. To the best of our knowledge, this is the first paper that targets the APR of Simulink models, the de-facto CPS modeling and simulation tool [2, 3].
- **Tool:** We prototype the method in a tool and provide it as open-source on GitHub, together with a replication package on Zenodo. Our code and up-to-date progress can be found at Github. See the replication package for details.
- Evaluation and Dataset: We carry out a comprehensive evaluation of our tool by complementing an existing dataset of Stateflow bugs [25] with a new dataset encompassing two new CPS models. In total, the dataset encompasses 9 real bugs, out of which 7 are provided by this paper. Despite not being a large number, in terms of the number of faults, it is the largest evaluation related to APR in the field of CPSs.

The rest of the paper is structured as follows: Section 2 presents relevant background, and Section 3 introduces our approach. Empirical evaluation, results, and threats are presented in Section 4, Section 5, and Section 6, respectively. We position our work with respect to relevant works in Section 7. We conclude the paper in Section 8.

2 Background

2.1 Automated Program Repair

Given a buggy program P, the corresponding specification S that makes P fail and the program transformation operators O, Automated Program Repair (APR) can be formalized as a function APR(P,S,O). PT is the set of its all possible program variants by enumerating all operators O on P. The problem of APR is to find a program variant P' ($P \in PT$) that satisfies S. The difference in the software program between P' and P is known as a *plausible patch*. S is usually defined as *passing* and *failing* test cases, where passing test cases demonstrate the functionality that needs to be preserved, whereas the failing one demonstrates the bug [23]. A plausible patch might be overfitted to S, therefore, an additional manual process may be required to validate that the plausible patch is a valid patch that fixes the bug.

Three main techniques exist for APR. Search-based program repair techniques attempt to recast program repair to that of an optimization problem by navigating through the search space of program edits. The goal is usually to reduce the number of failing test cases while increasing the number of passing ones [12], although other objectives exist too (e.g., reduce the number of program edits [13]). Another approach involves semantic program repair [14], which focuses on making the repair process explicit by deriving a specification from it. Starting with a correctness requirement, a constraint is deduced that describes the necessary changes. Subsequently, a minimal patch is generated to satisfy this constraint. The last technique is based on deep learning techniques. This is based on learning repair strategies from human patches. The learning-based repair techniques (e.g., Getafix [26]) first mine human patches that fix defects in existing software repositories, train a general repair rule and apply the model to buggy programs to produce patches. Besides, the use of LLMs has been recently leveraged for APR [17]. These, instead of mining and training an algorithm, make use of pre-trained LLMs. In this paper, we focus on the first type of technique, i.e., search-based program repair.

2.2 Stateflow Models

Stateflow [27] is a visual programming language and simulation environment for Simulink created by MathWorks. It is primarily used for modeling and simulating complex control systems and state-based behaviors in various applications. The main components of a Stateflow diagram are the states and the transitions. The states are a set of events shown as a box representing a set of behaviors or actions that the system can perform at that moment. To switch from one state to another, a transition must be triggered. The transitions are represented as arrows that connect the states. To activate a transition, a set of boolean conditions must be met.

Figure 1 shows an example of a Stateflow diagram that controls the temperature of a fridge. The example has four states: (1) CLOSE_NORM indicates that the door of the system is closed and the temperature is cold enough not to activate the cold-down function; (2) CLOSE_HOT indicates that the temperature is too warm, and needs to be cold down through the variable COLD; if the door is opened, the system enters in the state (3) OPEN. When this happens, the cold-down function is disabled, and a light is turned on through the variable LIGHT. If the door remains open for 15 seconds, it enters the state (4) OPEN_15_SEC. This state triggers an alarm to indicate to the user that the door shall be closed, which is carried out through the the ALARM variable.

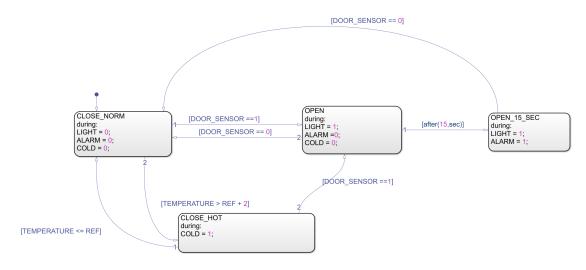


Figure 1: Stateflow diagram of a fridge temperature controller

3 FlowRepair

3.1 Overview of the APR Process

Figure 2 shows an overview of FLOWREPAIR. Like most APR approaches, FLOWREPAIR's first stage involves a fault localization process. This narrows the search space for finding plausible patches, making the repair process more efficient. We opted for using Spectrum-based Fault Localization (SBFL) for this task, as it is one of the most commonly employed approaches to localize faults [28]. To this end, in the first stage, the tool automatically instruments the Stateflow models inside a Simulink model. This is carried out to obtain, for each test case, which of the transitions and states were executed. We also considered employing Simulink Design Verifier (SDV) for this task. SDV is a Mathwork's

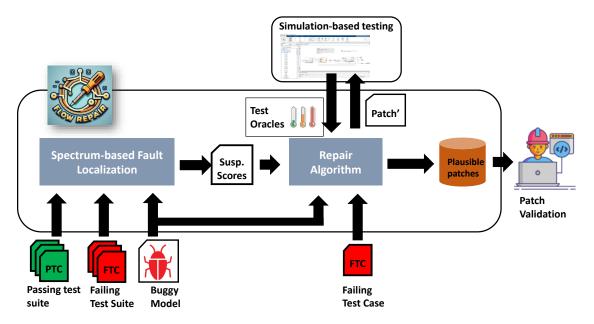


Figure 2: High-level overview of FLOWREPAIR

toolbox that gives the option for analyzing coverage data. However, we discarded this option because an additional license is required, restricting FLOWREPAIR's access to any practitioner and researcher without the license. When the model is instrumented, FLOWREPAIR executes the entire test suite. By tracing the data of each failing and passing test case with the executed transitions and states, we obtain, for each transition and state of the model, a suspiciousness score. The suspiciousness scores of the components of the stateflow models in Simulink models are extracted by using the well-known SBFL Tarantula metric [28]. Note, however, that the tool is flexible and that any other SBFL metric can be easily incorporated. The suspiciousness scores are employed in the upcoming repairing stage.

After obtaining the suspiciousness scores of each transition and state in the Simulink model, FLOWREPAIR triggers the repair algorithm, explained in Section 3.2. This algorithm combines a global search process with a local one to propose patches. These patches are executed by using simulation-based testing, which returns simulation results, which are processed by FLOWREPAIR to obtain a set of repair objectives (explained in Section 3.3). Notably, a single failing test case is employed during the repair process. This is (1) because executing simulation-based test cases of CPSs is highly expensive, thus, increasing the tool's efficiency; and (2) because CPSs are usually tested by employing falsification-based approaches that stop generating test cases once a failure is triggered [10], therefore, having a single failing test case in the test suite. The repair algorithm stops when a time budget is exceeded. All the plausible patches are given to the software engineer for manual inspection and validation.

3.2 Repair Algorithm

Algorithm 1 shows the overview of FLOWREPAIR's steps for repairing Stateflow bugs. The algorithm takes as inputs a Simulink model to repair (modelToRepair) and the suspiciousness ranking obtained with SBFL (suspiciousnessRanking). As output, it provides an archive of plausible patches, i.e., patches that pass the original test suite and that should undergo a manual validation process to assess their correctness. The algorithm maintains two archives: *PlausiblePatchArch*, containing all plausible patches found during the repair process, and *PartialArchive* having partial patches found during the repair process. A partial patch is a partially repaired solution, i.e., a solution that enhances the previous model, but that still does not pass the failing test case. These two archives are initialized in Lines 1 and 2 of Algorithm 1.

After this, the algorithm enters in a while loop (Lines 3-26, Algorithm 1) that finishes when the time budget has been exceeded. This algorithm has two main parts, divided into two while loops. On the one hand, the first part of FLOWREPAIR, lines 5-14 in Algorithm 1, applies a global search routine to find plausible patches or partial patches. To that end, the algorithm first selects the model to be repaired among all models included in PartialArchive. This selection is carried out randomly. After, this model is mutated (Line 6, Algorithm 1) by applying a mutation to the Stateflow model of the CPS based on the developed mutation operators (Section 3.4). To select the item of the model to be mutated (i.e., a state or a transition of the model), the algorithm uses the suspiciousness score information of

Algorithm 1: Overview of FLOWREPAIR's repair algorithm

```
Input: modelToRepair
      suspiciousnessRanking
      Output: PlausiblePatchArch
 1 PlausiblePatchArch \leftarrow \emptyset
 _2 PartialArchive \leftarrow FaultyModel
      while TimeBudgetNotExceeded do
                 VerdictEnhanced \leftarrow False
                 while VerdictEnhanced == False and TimeBudgetNotExceeded do
 5
                           modelToRepair \leftarrow SELECTMODEL(PartialArchive)
                           mutatedModel, mutatedComponent, mutationOperator \leftarrow
                              APPLYGLOBALMUTATIONS(modelToRepair,suspiciousnessRanking)
                           verdict \leftarrow EXECUTETESTS(mutatedModel)
                           if verdict == pass then
10
                                     PlausiblePatchArch \leftarrow PlausiblePatchArch \cup mutatedModel
                           else if CHECK VERDICTENHANCED (verdict, Archive) then
11
                                      VerdictEnhanced \leftarrow true;
12
                                      PartialArchive \leftarrow PartialArchive \cup mutatedModel
13
                                     modelToApplyLocalMutations \leftarrow mutatedModel
14
                numOfLocalTries \leftarrow 0
15
                 while numOfLocalTries < totalLocalTries and TimeBudgetNotExceeded do
16
                           numOfLocalTries \leftarrow numOfLocalTries + 1
17
                           locally Mutated Model \leftarrow APPLYLOCAL MUTATIONS (model To Apply Local Mutations, mutated Component, and the support of the supp
18
                              mutationOperator)
                           verdict \leftarrow EXECUTETESTS(locallyMutatedModel)
19
                           if verdict == pass then
20
                                     PlausiblePatchArch \leftarrow PlausiblePatchArch \cup locallyMutatedModel
21
                           else if CHECKVERDICTENHANCED(verdict, Archive) then
22
                                     Partial Archive \leftarrow Partial Archive \cup locally Mutated Model
23
                                     modelToApplyLocalMutations \leftarrow locallyMutatedModel
24
                                     numOfLocalTries \leftarrow 0
25
                REORGANIZEARCHIVE(PartialArchive)
```

each state/transition of the Stateflow model. Specifically, we employ a roulette wheel selection, which gives a higher probability of being mutated to those states and transitions with higher suspiciousness scores, whereas those with the lowest suspiciousness scores have lower chances of being selected. Once this is mutated, the model is tested (Line 7, Algorithm 1). The test execution provides a verdict. If the verdict is a pass, we include the mutated model in the archive of plausible patches (Line 9, Algorithm 1). If the verdict is not a pass, we check whether the verdict was enhanced with respect to the selected model (Line 11, Algorithm 1). We consider a verdict is enhanced if any of the repair objectives is enhanced and none of the repair objectives is worsened. If the verdict is enhanced, then we include the mutated model in the archive of partial patches (Line 13, Algorithm 1), and we trigger the local repair process.

On the other hand, the second part of FLOWREPAIR's repair algorithm applies a local search routine to locally enhance a partial patch (Lines 16-25, Algorithm 1). This part of the algorithm is triggered when the global search routine finds a partial patch (i.e., an enhancement based on the repair objectives, explained in Section 3.3). The local search process exploits the search of patches by applying local mutations in the element of the Stateflow model (i.e., state or transition) that was mutated during the global search routine and enhanced the verdict. We limit the number of local tries to a predefined number (in our experiments to 30). We limit this number due to two main reasons. Firstly, because the enhancement could be due to an overfitted partial patch. Secondly, to avoid the search algorithm getting trapped in a local optima, which is common in local search processes. After applying the local mutation (Line 18, Algorithm 1), we execute the failing test case to the model (Line 19, Algorithm 1). If the verdict is a pass, then we include the model in the archive of plausible patches (Line 21, Algorithm 1), and continue with the local mutation process to find additional plausible patches. If a partial patch is found, i.e., the verdict is enhanced but not fully passed, then a partial patch is included in the archive of partial patches, and the local search process is restarted with the new partial patch model (Line 24, Algorithm 1) and initializing the number of local tries to 0 (Line 25, Algorithm 1).

The local search process is finished either when the total time budget has been exceeded or when the number of local tries has exceeded the predefined number of total tries. When finishing the local search process, the archive of partial patches is reorganized. This is because it may include many partial patches, making the global search exhaustive. This

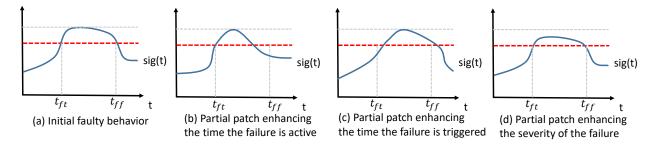


Figure 3: Given a signal over time (sig(t)) that violates the requirement involving the exceed of the red signal, an example of three repair objectives being improved

reorganization can be done by following different strategies, such as considering Pareto-optimality among the selected repair objectives. After carrying out a set of preliminary experiments, we saw that this archive did not exceed a large number of partial patches; thus, we decided not to remove any partial patch from the archive in the current version of FLOWREPAIR. Moreover, we found that some partial patches may be overfitted (i.e., not correct). Therefore, removing patches from this partial patch may pose the risk of removing non-overfitted partial patches, which would increase the probability of not producing valid patches. Therefore, unlike previous APR approaches (e.g., [20, 19]), we found it sensible not to remove patches from this archive.

3.3 Repair Objectives

Search-based APR approaches require a set of repair objectives to guide the search. Most approaches (e.g., GenProg [12], Arja [13]) employ the number of pass and failed test cases for guidance. However, in the context of CPSs, as explained in previous sections, executing a long number of test cases is impossible. Moreover, it is typical to only have a single failure-inducing test case. Subsequently, current repair objectives that are based on the number of failed and passed test cases are invalid for search-based APR in the context of CPSs, as the fitness landscape would become flat, precluding an appropriate search process.

To solve this issue, we take advantage of the notion of time and the severity of failures for guiding the search algorithm, provided that test cases in the context of CPSs are signals over time.

Motivating Example: Consider as an example Figure 3, where the output signal over time (sig(t)) should not exceed the threshold value delimited by the red dashed line. Note, however, that in Figure 3-a this requirement is violated. The time at which the signal violates the requirement begins at time t_{ft} , and the time at which the failure finishes is t_{ff} .

Specifically, we define three repair objectives, out of which two are novel for the field of APR:

- Time the failure is active: When executing the simulation of the CPS with a failure-inducing test case, we measure the time a failure is active. That is when the CPS is considered not to be working properly. The goal of FLOWREPAIR is to reduce this metric, as we conjecture that the lower the time a failure is active, the closer the repair algorithm is to providing a plausible patch. An example of this repair objective being enhanced by a partial patch is shown in Figure 3-b. In this case, it can be observed that the time the failure is active is less than $t_{ff} t_{ft}$.
- Time the failure is triggered: When executing the simulation of the CPS with a failure-inducing test case, we measure when the failure is triggered. If, for a potential patch, the time at which the failure is triggered is postponed with respect to the selected buggy model from the archive, we consider that the potential patch is closer to being a plausible patch. Therefore, the goal of FLOWREPAIR is to increase the time at which the failure is triggered. An example of this repair objective being enhanced by a partial patch is shown in Figure 3-c. In this case, it can be observed that the time the failure is triggered is higher than t_{ft} . This repair objective is particularly interesting in those cases where the CPS does not recover from the failure (e.g., in autonomous driving systems, it is common that the vehicle goes out of bounds).
- Severity of the failure: Test verdicts provided by test oracles in the context of CPSs are not only boolean classifications of either pass or fail, but also provide, in the case of failed test cases, the severity of the failure, and in the case of pass test cases how far they are from their optimal performance [11]. As a result, most APR techniques applied in the field of CPSs have relied on the severity of the failure to guide the search process towards finding plausible patches [19, 7, 29]. Since this objective can help repair the Stateflow model too, and is complementary to the two new novel repair objective functions we define, we also include it in

FLOWREPAIR. An example of this repair objective being enhanced by a partial patch is shown in Figure 3-d. In this case, it can be observed that although sig(t) is still above the established threshold, its maximum value does not reach the grey dashed line.

3.4 Mutation Operators

The mutation operators of FLOWREPAIR represent program edits on the Stateflow model. Similar to other search-based APR tools, like GenProg [12] or Arja [13], FLOWREPAIR provides three types of mutation operations in the faulty Stateflow model: *delete*, *replace*, and *insert*. More specifically, for suspicious components of the Stateflow model, FLOWREPAIR applies deletion (D), replacement (R), or insertion (I) patches to repair the buggy component(s). The considered components of the Stateflow model apply either to States or Transitions of the model. A problem with search-based APR tools is the number of implemented mutation operators. On the one hand, the lower the number of mutation operators, the lower the probability of generating a correct patch. On the other hand, the higher the number of operators, the larger it is the search space, which could impact the scalability of the approach. FLOWREPAIR contains a total of 15 mutation operators, which is large enough to provide sufficiently high repairability to generate correct patches in many different buggy scenarios. Besides, we employ these mutation operators in two ways to solve the scalability problem that may suppose having many operators: applying global mutation operators and applying local operators (further explained in Section 3.5). We now explain the 15 selected operators:

- **Relational Operator Replacement (R):** It replaces a relational operator with another one (e.g., > for >=). This operator can be applied to transitions.
- Conditional Operator Replacement (R): It replaces a conditional operator with another one (e.g., && for ||). This operator can be applied to transitions.
- Mathematical Operator Replacement (R): It replaces a mathematical operator for another one (e.g., + for

 This operator can be applied to both transitions and states.
- Unit Change in After Function (R): It changes the time unit in an after function (e.g., after(5,sec) for after(5,msec)). This operator can be applied to transitions.
- Numerical Replacement Operator (R): It changes a numeric value for another value (e.g., 1 for -1). This operator can be applied to both transitions and states.
- Transition Destination Replacement (R): It changes the destination state of a transition to another state; the destination state is randomly chosen.
- Transition Root Replacement (R): It changes the origin state of a transition to another state; the new origin state is randomly chosen.
- Initial Transition Change (R): It changes the initial transition; consequently, it can only be applied in those initial transitions.
- State Deletion (D): It deletes one of the states of a Stateflow model. It can be applied to states, but all affected transitions are also removed.
- Transition Deletion (D): It deletes one of the transitions of the Stateflow model. This operator can be applied to transitions.
- State Variable Deletion (D): It deletes a variable from a state. This operator can be applied to states.
- Transition Condition Deletion (D): It deletes a condition from a transition. This operator can only be applied to those transitions with multiple conditions.
- Mathematical Operation Insertion (I): It inserts a mathematical operation based on the available inputs and outputs of the Stateflow model in a targeted transition. This operator can be applied to states.
- Variable Insertion (I): It inserts a new variable in a state. This operator can be applied to states.
- Condition Insertion (I): It inserts a new condition in a transition; thus, this operator can be applied to transitions.

3.5 Global vs Local Mutations

To generate a patch, we apply a mutation to a Stateflow model based on one of the 15 mutation operators defined in Section 3.4. However, we do this in two ways, depending on the search routine at which the operator is triggered. We define these routines as global or local search. During *global search*, the goal is to explore a large set of patches across different (suspicious) components of the Stateflow model. To generate a patch during the global search, the global

mutation operator obtains one model to be repaired and the suspiciousness ranking for all the components in such a model (Line 7, Algorithm 1). A component (either state or transition) of the Stateflow model is selected to be mutated. This selection is carried out via a roulette-wheel selection, which gives a higher mutation probability to a component with a higher suspiciousness and a lower probability to a component with a lower suspiciousness score. Based on this, the global mutation randomly selects one of the mutation operators compatible with the selected component and applies the mutation to it.

In contrast, during *local search*, the goal is to focus on one specific component of the Stateflow model. This is carried out when a mutation during global search has enhanced at least one of the three objective functions and has not deteriorated any of the other two objective functions. When this situation is given, the local mutation obtains the successful model, the mutated component, and the applied mutation operator (Line 18, Algorithm 1). The local mutation only changes the component that has been enhanced. This is because it is likely that a change in a component that has enhanced at least one of the repair objectives has a high probability of being the buggy component. This theory aligns with the theory behind unified debugging [30, 31]. On the other hand, we follow a different strategy when selecting the mutation operator when compared to the global search. In this case, we give a 50% probability that the same mutation operator will apply. For instance, if the relational operator replacement changes > for <, and this has enhanced the verdict, we try to apply the same operator to see if another replacement (e.g., <=) is capable of repairing the patch. This decision was taken because we believe that, in many cases, not only the component to be mutated is important, but also the mutation operator. However, we also give a 50% chance of applying another mutation operator, as many patches involve changing different parts (e.g., in a transition, you may change the relational operator but also a conditional operator).

4 Empirical Evaluation

4.1 Research Questions

In our evaluation, we aimed to answer the following research questions (RQs):

- **RQ1 Repairability:** To what extent can FLOWREPAIR repair state-flow faults? This RQ assesses the effectiveness of FLOWREPAIR to repair the Stateflow models in terms of generating plausible and valid patches.
- RQ2 Comparison with other algorithms: How does FLOWREPAIR compare to a baseline algorithm? We assess whether FLOWREPAIR is better than a (1+1)EA version of it, which is a common algorithm for APR in the context of other CPS studies.

4.2 Dataset

Our dataset involves a total of 9 faulty models of 3 different case study systems. It is important to note that, while the number of faulty models is not large, the number of datasets involving real faults in Simulink models in general, and in Stateflow models in particular, is scarce. To the best of our knowledge, the only existing dataset involving faulty Stateflow models is provided by Ayesh et al. [25], providing a total of 2 faulty models. We extend this dataset by including 7 new faulty models. These faulty models were extracted from students' projects where they had to develop a Stateflow controller. Note that the practice of using students' projects is common in other APR studies [32]. Although the number of bugs appears not to be large, it is important to note that (1) it is often difficult to experiment with a large set of faulty models in the context of CPSs due to their high computation [19, 20]; and (2) this is the study of APR for CPSs with the largest number of bugs (i.e., 8 more bugs than Valle et al. [19], 7 more bugs than Abdessalem et al. [20], 5 more bugs than Jung et al. [33] and 3 more bugs than Lyu et al. [29]). We also considered to extend this dataset by including mutants. However, the mutation operators would be the same as the repair operators discussed in Section 3.4, thus, significant bias would be included in the evaluation, increasing the risk towards having flawed results.

Therefore, in total, our dataset includes three different case study systems. The first one relates to two models of a pacemaker. Each model has one independent bug. Information about these buggy simulink models can be found on [25]. The second case study system involves the control of a fridge. We had three faulty models, but the second one had two faults in the model. Therefore, fridge_1, fridge_2 and fridge_3 represent the buggy models. Fridge_2a represents the second model with one of the bugs fixed and Fridge_2b represents the second model with the other bug fixed. This would simulate how the APR tools behave when one of the bugs is manually fixed but the other one is not fixed yet. Lastly, the third case study system involves the model of an automated door that automatically opens and closes when detecting people in the building. The details (e.g., requirements) of these two case study systems can be found in the replication package.

As for the generated test cases, for each faulty model, we asked an independent developer to generate a failing and a passing test case. As for the oracles, we employed a regression oracle that provides the expected value of the system throughout the test execution. We opted for this kind of oracle because it is a widely adopted method by industrial CPS developers (e.g., [34]).

4.3 Baseline Algorithm

We implemented a (1+1)EA version of our algorithm, which is in line with the algorithms used by Abdessalem et al. [20] and another CPS misconfiguration repair approach [19]. The algorithm is basically a version of our algorithm without the local search routine. It also maintains only a single partial archive, i.e., if a partial patch is found, the previous patch is removed from the archive. It is noteworthy, that the baseline algorithm employs the same strategy to select the component to be repaired as FLOWREPAIR. The algorithm is also stronger than the commonly employed Random Search baseline in other search-based studies.

4.4 FLOWREPAIR Configuration

FLOWREPAIR has two configurable parameters. Based on a set of preliminary experiments, we decided on the following values for each of them: (1) time budget: 1 hour; (2) the number of local tries: 30. The time budget can be increased in future studies depending on how much time the test execution takes for a specific CPS. As indicative for future replication studies, with this time budget and our case study systems, our algorithm generated between 150 and 1060 patches in each run, depending on the case study system.

As explained during the previous section, we also used the *Tarantula* metric to obtain the suspiciousness score of our test cases, which is one of the most employed suspiciousness metrics [28].

4.5 Evaluation Metrics

For each run of the algorithm, we extract two metrics: (1) the number of plausible patches and (2) the number of valid patches. The former relates to the number of patches provided by FLOWREPAIR that passes the test suite, while the latter refers to the number of patches that are semantically equivalent to the patch proposed by the developer.

4.6 Experiment Runs

Given the stochastic nature of search-based approaches, including FLOWREPAIR, we run the algorithm 5 times, both, for FLOWREPAIR and the baseline algorithm. As we employed 9 faulty models, and the given search budget was 1 hour, in total, the experiments lasted 90 hours (i.e., 5 (repetitions) \times 9 (models) \times 2 (algorithms) = 90 hours). We could not afford a higher number of repetitions, not only because of the long execution time of the algorithms, but also due to the significant manual effort for validating each of the plausible patches (i.e., with the current setup, we had to manually validate 686 plausible patches to check its semantic equivalence with the patch provided by the developer).

4.7 Execution Environment

We used MATLAB 2022b version for the models and FLOWREPAIR. The employed operating system was Windows 10, with 16GB RAM memory and an AMD Ryzen 7 5800HS processor with 8 cores and 16 threads.

5 Analysis of the Results and Discussion

5.1 RQ1 – Repairability of FLOWREPAIR

Figure 4 depicts the obtained results in terms of the number of plausible patches found by FLOWREPAIR over time. Further, Table 1 summarizes the global results of the entire algorithm executions. As can be seen, for all faulty models except for the pacemaker_fault2, FLOWREPAIR was able to provide at least one plausible patch. It can also be appreciated from Figure 4 that the number of found plausible patches increased over time. In addition, it can be appreciated the standard deviation across the 5 runs. The example from fridge_3 between 2600 and 3300 seconds shows no standard deviation since in all runs the number of plausible patches was the same (i.e., 1 plausible patch). The example from fridge_2 is interesting, as this faulty model included two faults. Note, however, that we consider a plausible patch when the test suite is passed, i.e., both bugs need their corresponding fixes. In that example, it is seen that plausible patches were found after 750 seconds of execution, i.e., roughly after 12 minutes. As expected, more time

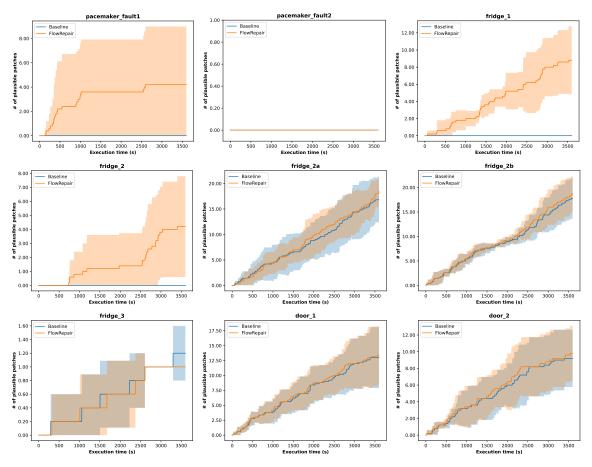


Figure 4: Average number of plausible patches found by FLOWREPAIR and baseline over 5 different runs for one hour each

was needed in this example because FLOWREPAIR focused on generating patches for one bug, and once this was fixed, focused on the second one.

Table 1: Average, minimum and maximum values obtained for the 5 different runs. PP stands for Plausible Patch and VP stands for Valid Patch

	FLOWREPAIR						Baseline					
Models	Plausible Patches			Validated Patches			Plausible Patches			Validated Patches		
	mean	min	max									
Pacemaker_1	4.2	0	11	0.4	0	1	0.0	0	0	0.0	0	0
Pacemaker_2	0.0	0	0	0.0	0	0	0.0	0	0	0.0	0	0
Fridge_1	8.8	3	14	8.8	3	14	0.0	0	0	0.0	0	0
Fridge_2	4.2	0	8	0.4	0	2	0.0	0	0	0.0	0	0
Fridge_2a	18.4	15	24	13.4	9	17	16.8	12	25	13.4	10	17
Fridge_2b	18.8	15	25	7.4	4	9	17.8	13	25	6.6	4	9
Fridge_3	1.0	1	1	0.0	0	0	1.2	1	2	0.0	0	0
door_1	13.4	8	21	0.0	0	0	13.0	8	21	0.0	0	0
door_2	9.8	6	15	2.0	2	6	9.0	5	15	2.0	2	6

FLOWREPAIR was not able to generate any plausible patch for pacemaker_fault2 model. When having a closer look at potential reasons, we saw that the bug involved an incorrect value given to a variable. The bug involved the following assignation: "VENT_CMP_REF

_PWM = 125;", whereas the patch submitted by the developer was "VENT_CMP_REF_PWM = VENT_Sensitivity/5*100;". Given that the "VENT_Sensitivity" variable was 3.5, the bug could eventually be fixed by applying the Numerical Replacement Operator mutation that changed the number 125 for 70, having this way a semantically equivalent fix to that proposed by the developer. Moreover, the tool could also apply

several mutation operators to generate the exact same fix as the one proposed by the developer. However, although FLOWREPAIR had the capacity for generating such a fix, unfortunately, the buggy state in which this assignment was carried out involved a total of other 14 variable assignments. Therefore, even if a perfect fault location was assumed, the search space to fix this bug in such a model would be extremely large. That is, the way the model is developed restricts efficient APR. We believe that a better design of the Stateflow model could have helped repair the bug by FLOWREPAIR, although a more in-depth evaluation is required to confirm this hypothesis.

All these plausible patches were manually verified to see whether they were valid. For the five different runs, in six out of the eight buggy models FLOWREPAIR found at least a valid patch. In four of them, in all runs, there was at least one valid patch. We believe these results are quite competent, and, therefore, can summarize the first RQ as follows:

RQ1: FLOWREPAIR was able to find plausible patches for eight out of nine buggy models, and valid patches for six out of the nine buggy models. In four of the models, FLOWREPAIR consistently suggested valid patches.

5.2 RQ2 – Comparison with Baseline

FLOWREPAIR was able to produce plausible patches for three out of nine buggy models, (i.e., fridge_1, fridge_2 and pacemaker_fault1) in which the selected baseline was not able to produce any plausible patch in any of the five runs. For the buggy model pacemaker_fault1, FLOWREPAIR significantly surpassed the baseline algorithm, which found on average 0.4 plausible patches per run, whereas FLOWREPAIR generated, 4.2 plausible patches per run on average. We further confirmed there was statistical significance for this particular case by applying the Wilcoxon-rank sum test. When having a closer look at buggy models, we found that for many of the generated plausible patches, FLOWREPAIR made significant use of the local search routine. This might mean that to fix more complex bugs the strategy that combines global and local search makes the repair process to be significantly more efficient, something paramount in the context of CPSs. For the remaining buggy models in which plausible patches were found, when considering average values, FLOWREPAIR slightly surpassed the baseline algorithm in all buggy models except for Fridge_3. However, the differences in all these cases were not statistically significant according to the Wilcoxon signed rank test we applied. In these buggy models, we found that, although for some models the patch was generated through the local search routine, in many other models, this was not necessary. Indeed, in a large portion of the models, the plausible patch was directly generated by applying a single edit on the initial buggy model. In those models, the fix could be provided by a pure random search version of FLOWREPAIR. We conjecture that the main reason is that the nature of such bugs is less complex than the ones in which the baseline algorithm failed to repair them.

Furthermore, the baseline was able to generate valid patches only in three of the buggy models, unlike FLOWREPAIR, which was able to generate valid patches in six of them. When comparing these three models, we found no statistical significance between FLOWREPAIR and the baseline.

RQ2: FLOWREPAIR is able to fix more bugs than the selected baseline. Given five tries, FLOWREPAIR found plausible patches in eight out of nine buggy models, whereas the baseline found plausible patches only in six models. FLOWREPAIR found valid patches in six buggy models, whereas the baseline found only in three of them.

6 Threats to Validity

We now discuss which were the threats of the evaluation and how we tried to mitigate them:

External Validity: We applied FLOWREPAIR to only nine faulty models in three case studies, which may not be large enough to generalize our findings. These faults, however, were real faults and the only ones we found available for Stateflow models. The only dataset we found was from Ayesh et al. [25]. We complemented these with additional faulty models, extending it to nine faulty models. We note that in the context of CPSs, it is not common to have a large number of real faults available, as discussed in Section 4.2. Nevertheless, we managed to have the largest dataset for APR in the context of CPSs.

Internal Validity: The configuration of various parameters of our algorithms is a concern that could affect the results. To mitigate this threat, we run preliminary experiments to find the optimal parameters of our algorithm, including the number of local tries.

Conclusion Validity: We dealt with the randomness in our algorithm by repeating each algorithm run five times. Furthermore, to compare it with the baseline algorithm, we apply a relevant guide [35] to select the suitable statistical tests (i.e., the Wilcoxon signed-rank test).

Construct Validity: We chose two appropriate measures that are commonly used in other APR studies [36, 37, 38, 16], i.e., the number of plausible valid patches and the number of valid patches. Moreover, we used the same stopping criterion for our algorithm and the baseline algorithm for a fair comparison, i.e., time budget.

7 Related Work

Testing CPSs has been a widely investigated topic [39]. This has been targeted from multiple perspectives, including test generation [10, 2, 3, 40, 9, 41, 42, 24], regression test optimization [43, 4] and the test oracle problem [11, 44, 45]. In addition, testing CPSs has also been tackled focusing on different kinds of systems, e.g., autonomous vehicles [46, 47, 48, 49], automated warehouses [50, 51], healthcare applications [52]. We focus on Stateflow models, which is a notation for modeling the high-level control logic of Simulink models. Testing has been tackled from all these different perspectives in the context of Simulink models. For instance, Matinnejad et al. [2, 3] focused on test generation based on search algorithms that aim at increasing certain anti-patterns. Menghi et al. [10] focused on test generation following a surrogate-assisted falsification-based approach. Nejati et al. [24] compared search-based techniques with model checking. Besides test generation, other existing approaches include oracle generation [11], test system generation [53], and test case prioritization [4]. However, all these approaches focus on testing. Conversely, our work focuses on repairing bugs specifically in the context of Stateflow models, which, to the best of our knowledge, has not been targeted yet.

Another line of research relates to debugging Simulink models. Some works focused on fault localization [54, 55, 56, 57, 58]. Deshmukh et al. [59] proposed a technique to localize misconfigurations. Other debugging works focused on other aspects, e.g., CPSDebug [60] tests, debugs, and explains failures in Simulink/Stateflow models. Boufaied et al. [61] focused on diagnosing CPS failures based on trace-checking input and output signals. Valle and Arrieta [62] focused on reducing the failure-inducing test inputs of Simulink models. While all these studies focused on localizing bugs to ease the repairing process of buggy models, none of them targeted the automated repair task, which remains a manual activity in the context of Stateflow/Simulink models.

In the context of classical software systems, automated repair has been an intensive area of research aiming at reducing the laborious manual task of repairing software with an automated approach to generate patches for the software [63]. To this end, many approaches have been proposed applying different techniques, such as search algorithms [64, 65, 12, 13], semantic analysis [14], and machine-learning [66, 67, 68, 18, 15, 16, 17]. All these seminal works on APR are applied to general purpose software, but, as explained in the introduction, they cannot be directly applied to CPSs due to scalability issues, and would require, at least, several modifications.

Despite this challenge, some works have focused on repairing CPSs. Abdessalem et al. [20] proposed Ariel, a tool that repairs interaction failures in automated driving systems. Jung et al. [33] proposed Swarmbug, a tool to repair misconfigurations of swarm aerial vehicles. Valle et al. [19] proposed an algorithm for repairing CPS misconfigurations applied to the context of elevator systems. Our works differ from these in two key perspectives. Firstly, these approaches do not focus on repairing the program itself, but either misconfigurations [19, 33] or feature interactions [20]. Secondly, although they could eventually be adapted to other domains, these works focus on specific CPS domains and adapt their tools to target specific challenges from those domains, i.e., automotive [20], robotics [33] and elevators [19]. Our work, in contrast, focuses on programs modeled in Simulink/Stateflow, which are common in many CPS domains [2, 3], and does not make any assumptions nor implementation that is restricted to a specific system or domain.

To the best of our knowledge, similar to our approach, only two works target repair on Simulink models. Singh and Saha [69] focus on debugging, fault localization, and repairing Simulink models. However, their repair is restricted solely to parameters, unable to repair bugs due to, e.g., faults in relational, conditional, and mathematical operators, faults due to the incorrect destination of a transition, etc. Therefore, their approach would fail to repair the most common bugs in Simulink models (e.g., it would not be able to repair any of the faults of our benchmark). Another relevant work is AutoRepair [29], which focuses on repairing neural networks embedded in CPSs. Specifically, AutoRepair isolates failure-inducing segments on a test case and uses a search algorithm to produce new data that satisfies an oracle. When this new data is found by the search algorithm, that data is used to extend the training data and retrain the neural network. Unlike AutoRepair, which focuses on repairing neural networks, FLOWREPAIR focuses on Stateflow models. AutoRepair and FLOWREPAIR are complementary: the former focuses on repairing bugs in neural networks, whereas the latter focuses on repairing Stateflow models.

8 Conclusion and Future Work

Simulink is a well-known modeling and simulation tool employed in industry for the development of CPSs. As part of such modeling, Stateflow models are often used to model the control logic of CPSs. While significant effort has

been devoted to advance on testing tasks of CPSs, the repair process has remained manual in such models. To this end, we presented FLOWREPAIR, an automated search-based tool to automatically repair Stateflow models. An empirical evaluation with 9 buggy models suggested that FLOWREPAIR is able to propose both valid and plausible patches in an affordable time budget for most buggy models, surpassing a baseline technique. However, among the plausible patches, we found that there were some over-fitting patches, which is a well-recognized problem in this context [23, 70, 71].

We identified the following research avenues that we will tackle in the near future. Firstly, we would like to analyze techniques to prioritize plausible patches for manual validation, reducing the time for finding valid patches. Secondly, we would also like to explore the potential of LLMs in proposing patches, for which we foresee applying different strategies (e.g., adding LLMs as mutators). Thirdly, we would like to explore and adapt techniques to alleviate the patch overfitting problem in the context of CPSs and Simulink models. Lastly, we would like to integrate this approach with other approaches (e.g., AutoRepair [29]), and propose other techniques to repair other Simulink parts (e.g., other blocks), offering a suite of APR techniques for Simulink models to industrial practitioners.

Replication Package

The tool and its evolution can be followed on GitHub: https://github.com/aitorarrietamarcos/StateflowRepairTool

The replication package and specific version used in our evaluation is available on Zenodo: https://zenodo.org/records/10936238

Acknowledgments

Aitor Arrieta and Pablo Valle are part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT1519-22), supported by the Department of Education, Universities and Research of the Basque Country. Shaukat Ali is also supported by the Co-evolver project (Project #286898), funded by the Research Council of Norway.

References

- [1] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2011.
- [2] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Automated test suite generation for time-continuous simulink models. In *proceedings of the 38th International Conference on Software Engineering*, pages 595–606, 2016.
- [3] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, 45(9):919–944, 2018.
- [4] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. *Journal of Systems and Software*, 149:1–34, 2019.
- [5] Sajad Khatiri, Sebastiano Panichella, and Paolo Tonella. Simulation-based test case generation for unmanned aerial vehicles in the neighborhood of real flights. In 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), pages 281–292. IEEE, 2023.
- [6] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 63–74, 2016.
- [7] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1016–1026, 2018.
- [8] Claudio Mandrioli, Seung Yeob Shin, Martina Maggio, Domenico Bianculli, and Lionel Briand. Stress testing control loops in cyber-physical systems. *ACM Transactions on Software Engineering and Methodology*, 2023.
- [9] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Employing multi-objective search to enhance reactive test case generation and prioritization for testing industrial cyber-physical systems. *IEEE Transactions on Industrial Informatics*, 14(3):1055–1066, 2017.

- [10] Claudio Menghi, Shiva Nejati, Lionel Briand, and Yago Isasi Parache. Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 372–384, 2020.
- [11] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C Briand. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 27–38, 2019.
- [12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.
- [13] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on software engineering*, 46(10):1040–1067, 2018.
- [14] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In 2013 35th International Conference on Software Engineering (ICSE), pages 772–781. IEEE, 2013.
- [15] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837, 2018.
- [16] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery, 2023.
- [17] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.
- [18] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt. In *IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2023, Melbourne, Australia, May 16, 2023*, pages 23–30. IEEE, 2023.
- [19] Pablo Valle, Aitor Arrieta, and Maite Arratibel. Automated misconfiguration repair of configurable cyber-physical systems with search: an industrial case study on elevator dispatching algorithms. In 45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023, pages 396–408. IEEE, 2023.
- [20] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Automated repair of feature interaction failures in automated driving systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 88–100, 2020.
- [21] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel C Briand. Comparing offline and online testing of deep neural networks: An autonomous car case study. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pages 85–95. IEEE, 2020.
- [22] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel Briand. Can offline testing of deep neural networks replace their online testing? a case study of automated driving systems. *Empirical Software Engineering*, 26(5):90, 2021.
- [23] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.
- [24] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C Briand, Stephen Foster, and David Wolfe. Evaluating model testing and model checking for finding requirements violations in simulink models. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1015–1025, 2019.
- [25] Mostafa Ayesh, Namya Mehan, Ethan Dhanraj, Abdul El-Rahwan, Simon Emil Opalka, Tony Fan, Akil Hamilton, Akshay Mathews Jacob, Rahul Anthony Sundarrajan, Bryan Widjaja, et al. Two simulink models with requirements for a simple controller of a pacemaker device. In *Proceedings of 9th International Workshop on Applied*, volume 90, pages 18–25, 2022.
- [26] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.
- [27] MathWorks. Stateflow, 2023.

- [28] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.
- [29] Deyun Lyu, Jiayang Song, Zhenya Zhang, Zhijie Wang, Tianyi Zhang, Lei Ma, and Jianjun Zhao. Autorepair: Automated repair for ai-enabled cyber-physical systems under safety-critical conditions. *arXiv preprint arXiv:2304.05617*, 2023.
- [30] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87, 2020.
- [31] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. Evaluating and improving unified debugging. *IEEE Transactions on Software Engineering*, 48(11):4692–4716, 2021.
- [32] Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. Propr: property-based automatic program repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1768–1780, 2022.
- [33] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian Elbaum, and Yonghwi Kwon. Swarmbug: debugging configuration bugs in swarm robotics. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 868–880, 2021.
- [34] Goiuria Sagardui, Leire Etxeberria, Joseba A Agirre, Aitor Arrieta, Carlos Fernando Nicolas, and Jose Maria Martin. A configurable validation environment for refactored embedded software: An application to the vertical transport domain. In 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pages 16–19. IEEE, 2017.
- [35] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery.
- [36] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 1161–1173. IEEE, 2021.
- [37] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.
- [38] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pages 1469–1481. IEEE, 2023.
- [39] Zahra Sadri-Moshkenani, Justin Bradley, and Gregg Rothermel. Survey on test case generation, selection and prioritization for cyber-physical systems. *Software Testing, Verification and Reliability*, 32(1):e1794, 2022.
- [40] Vincenzo Riccio and Paolo Tonella. Model-based exploration of the frontier of behaviours for deep learning system testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 876–888, 2020.
- [41] Georgios E Fainekos, Sriram Sankaranarayanan, Koichi Ueda, and Hakan Yazarel. Verification of automotive control applications using s-taliro. In 2012 American Control Conference (ACC), pages 3567–3572. IEEE, 2012.
- [42] Dmytro Humeniuk, Foutse Khomh, and Giuliano Antoniol. A search-based framework for automatic generation of testing environments for cyber–physical systems. *Information and Software Technology*, 149:106936, 2022.
- [43] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology*, 114:137–154, 2019.
- [44] Jon Ayerdi, Pablo Valle, Sergio Segura, Aitor Arrieta, Goiuria Sagardui, and Maite Arratibel. Performance-driven metamorphic testing of cyber-physical systems. *IEEE Transactions on Reliability*, 2022.
- [45] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1264–1274, 2021.
- [46] Chengjie Lu, Yize Shi, Huihui Zhang, Man Zhang, Tiexin Wang, Tao Yue, and Shaukat Ali. Learning configurations of operating environment of autonomous vehicles to maximize their collisions. *IEEE Transactions on Software Engineering*, 49(1):384–402, 2023.

- [47] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *Proceedings of the ACM/IEEE* 42nd International Conference on Software Engineering, pages 347–358, 2020.
- [48] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 132–142, 2018.
- [49] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.
- [50] Man Zhang, Shaukat Ali, Tao Yue, Roland Norgren, and Oscar Okariz. Uncertainty-wise cyber-physical system test modeling. *Software & Systems Modeling*, 18(2):1379–1418, 2019.
- [51] Man Zhang, Shaukat Ali, and Tao Yue. Uncertainty-wise test case generation and minimization for cyber-physical systems. *Journal of Systems and Software*, 153:1–21, 2019.
- [52] Hassan Sartaj, Shaukat Ali, Tao Yue, and Kjetil Moberg. Hita: An architecture for system-level testing of healthcare iot applications. In *Proceedings of the 17th European Conference on Software Architecture: Companion Proceedings*. Springer, 2023. To Appear.
- [53] Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria, and Justyna Zander. Automatic generation of test system instances for configurable cyber-physical systems. *Software Quality Journal*, 25(3):1041–1083, 2017.
- [54] Bing Liu, Shiva Nejati, Lionel C Briand, et al. Improving fault localization for simulink models using search-based testing and prediction models. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 359–370. IEEE, 2017.
- [55] Bing Liu, Shiva Nejati, Lionel Briand, Thomas Bruckmann, et al. Localizing multiple faults in simulink models. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 146–156. IEEE, 2016.
- [56] Bing Liu, Lucia, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability*, 26(6):431–459, 2016.
- [57] Bing Liu, Shiva Nejati, Lionel C Briand, et al. Effective fault localization of automotive simulink models: achieving the trade-off between test oracle effort and fault localization accuracy. *Empirical Software Engineering*, pages 1–47, 2018.
- [58] Ezio Bartocci, Leonardo Mariani, Dejan Ničković, and Drishti Yadav. Search-based testing for accurate fault localization in cps. In 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE), pages 145–156. IEEE, 2022.
- [59] Jyotirmoy Deshmukh, Xiaoqing Jin, Rupak Majumdar, and Vinayak Prabhu. Parameter optimization in control software using statistical fault localization techniques. In 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), pages 220–231. IEEE, 2018.
- [60] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. Cpsdebug: Automatic failure explanation in cps models. *Int. J. Softw. Tools Technol. Transf.*, 23(5):783–796, oct 2021.
- [61] Chaima Boufaied, Claudio Menghi, Domenico Bianculli, and Lionel C Briand. Trace diagnostics for signal-based temporal properties. *IEEE Transactions on Software Engineering*, 2023.
- [62] Pablo Valle and Aitor Arrieta. Towards the isolation of failure-inducing inputs in cyber-physical systems: is delta debugging enough? In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 549–553. IEEE, 2022.
- [63] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.
- [64] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), pages 162–168, 2008.
- [65] Andrea Arcuri and Xin Yao. Coevolving programs and unit tests from their specification. In Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE '07, page 397–400, New York, NY, USA, 2007. Association for Computing Machinery.
- [66] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In 2013 35th International Conference on Software Engineering (ICSE), pages 802–811, 2013.

- [67] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 511–523, New York, NY, USA, 2022. Association for Computing Machinery.
- [68] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pages 1169–1180, 2022.
- [69] Nikhil Kumar Singh and Indranil Saha. Specification-guided automated debugging of cps models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4142–4153, 2020.
- [70] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543, 2015.
- [71] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24:33–67, 2019.