applied sciences

MDPI

Article

Model-Based Test Case Prioritization Using an Alternating Variable Method for Regression Testing of a UML-Based Model

Ki-Wook Shin [1,†] and Dong-Jin Lim [2,*,†]

1 Department of Electronic Systems Engineering, Hanyang University, Ansan 15588, Korea; kwshin@hanyang.ac.kr
2 Division of Electrical Engineering, Hanyang University, Ansan 15588, Korea
* Correspondence: limdj@hanyang.ac.kr; Tel.: +82-31-400-5212
† Current address: Hanyang University, 55 Hanyangdaehak-ro Sangnok-gu, Ansan-si, Gyeonggi-do 15588, Korea.

Received: 5 October 2020; Accepted: 19 October 2020; Published: 26 October 2020

Abstract: Many test case prioritization (TCP) studies based on regression testing using a code-based development approach have appeared. However, few studies on model-based mutation testing have explored what kind of fault seeding is appropriate or how much the code-based results differ. In this paper, as automatic seeding for the mutation generation, several mutation operators were employed for the UML statechart. Here, we suggest mutation testing employing the model-based development approach and a new TCP method based on an alternating variable method (AVM). We statistically compare the average percentage of the fault detection (APFD) results of the new method to other TCP methods such as a greedy algorithm for code coverage or fault exposure possibility. Finally, in empirical studies, the model-based TCP results for a power window switch module, a body control module, and a passive entry and start system are derived; these are real industrial challenges in the automotive industry.

Keywords: mutation testing; test case prioritization (TCP); UML model; software model; regression testing

1. Introduction

Model-based development approaches are required because software is becoming increasingly complex. Such approaches design, implement, and verify models using formal techniques. In model-based testing, during model development, the software is verified and validated early using model-based simulation to minimize the effort required. Most code-based regression tests have employed test case prioritization (TCP) employing fault injection. Few reports on TCP during model-based development have appeared. It is essential to optimize the test suite continuously while regenerating new test cases in the regression testing as the software model changes, as in the code-based development approach. During model-based development, the software can be altered to reflect customer requirements or to fix bugs, and a regression testing environment is built in the same manner as that of traditional code-based development. The strategies used to optimize test cases in the regression testing environment include test case minimization, selection, and prioritization. Minimization removes unnecessary or redundant test cases from the test suite; selection identifies a partial suite from the entire test suite with which to test a change in part of the software. Prioritization selects cases that allow early fault detection, thus cost-effectively prioritizing the test order. In such a regression testing environment, various objective functions can be used to optimize test cases. Structural coverages (statement and branch coverage, and an MC/DC) afford

code-level coverage or transitional coverage at the model level. Alternatively, test cases in a suite can be optimized by what they reveal in terms of faults. Here, we use TCP to optimize test cases created via fault injection testing using mutants. To do this, a mutant of the model has to be created via hand seeding or automatic seeding using mutation operators. In a study by Ali et al. [1], hand and automatic seeding were compared; the mutation operators generated many more mutations than did hand seeding. Additionally, these results showed that mutation operators were more efficient to generate test cases based on mutation testing within a given time.

Such TCP techniques have been used to compare algorithms that derive objective functions including the average probability of fault detection (APFD), or average probability of fault detection with cost (APFDc); these are traditional measures. However, previous studies were limited to algorithms or objective functions based on traditional code-based development. During mutation testing, automatic fault seeding uses mutation operators to generate mutants efficiently. By using these mutation operators, mutants with faults can be created in existing programs and test cases can be used to compare the original code and the mutants to reveal faults.

Test cases can be prioritized to reveal several faults at once, or to identify serious faults at an early stage to achieve an objective function. However, the previous TCP was code-based; it is necessary to validate that the same objective functions can be used for model-based mutation testing featuring automatic fault seeding using mutation operators. For example, after creating a control flow graph corresponding to the sequential execution order based on the source code, a mutation operator can create a mutant that changes the graph or the relational operators as a coding mistake. However, unlike the code-based development approach, a software mutant can affect the structural and behavioral diagrams. Here, we compare a TCP method based on model-based development approaches with traditional code-based results for mutation testing.

First, we developed our software model for the automotive electronic components based on the model-based development approach. To obtain an optimized test suite in a regression testing, we wished to employ the mutation testing, but most of the research focused on the code-level. What we found from the research trend was that the model-based mutation testing is not researched well since code-based mutation testing can be employed to the model using automatic code generation. However, the model is one of the important test oracles, which can generate test cases for verification and validation. As our research motivation, we researched what kind of mutation operators should be considered and which algorithm can efficiently achieve the high APFD result for the model-based mutation testing.

The research questions can be summarized as follows:

RQ1. What mutation operators should be used for model-based mutation testing?

RQ2. Which TCP technique is optimal for fault detection? Does this result differ from the code-based result?

RQ3. Is the optimal search algorithm for model-based TCP the same as the code-based algorithm? Does this work for real industrial cases?

To answer RQ1, we define mutation operators active on state diagrams (unlike the operators employed during code-based mutation testing) and use them to create mutants. The operators used in existing code-based or model-based studies are applied to unified modeling language (UML)-based state diagrams. Section 3 explores whether such operators can be used for model-based mutation testing. To answer RQ2, we evaluate whether TCP methods based on code can be used for model-based mutation testing. Test cases are created using the mutants of Section 5, and prioritized. To answer RQ3, test cases are prioritized with the aid of the alternating variable method (AVM) (a representative test case generation method). Search-based software features simulated annealing, the use of genetic algorithms, or particle swarm optimization. Again, we use the traditional AVM. Ali, Muhammad, and Andrea [1] showed that a limited-space search may be superior to the AVM. We employ the AVM because the interface range of any specific module is limited. With our research questions in

mind, we generate test cases to kill mutants generated by the mutation operators and compare the model-based and code-based results of TCP using APFD as the objective function.

Section 2, Background explores the model-based development approach, the related test case prioritization, and mutation testing. Next, Section 3 shows that our model-based mutation testing method is specifically described and applied to the triangle classification example in Section 4. From Section 5, we explain the empirical results on the three automotive electronic modules. Finally, we review the total experimental results and our contribution in Section 6.

2. Background

Our test case generator features a generational structure that combines a custom parser with a satisfiability modulo theories (SMT) solver to generate test cases in a model-based development environment. The test cases are based on a class diagram and a state machine [2]. First, XML path language (XPath) is used to retrieve the information needed to generate a test case from the model via XMI transformation [3]. XPath then parses the state diagrams and the action language or implementation code of the functions using ANTLR, a representative custom parser generator. The parsed text is used to generate test cases automatically based on Yices (an SMT solver [4]).

2.1. The Model-Based Development Approach

The model-based development approach is a recent technique that enables rapid software development and validation. The greatest advantages are that the technique can detect errors in software development early based on analysis of customer requirements and it uses simulation techniques to merge software complexities. The model is a tool for customer specification analysis, so it can be useful for early verification via prototyping, and serves as an oracle when generating test cases using a formal model language or a specific tool-based model such as Simulink. The test cases are created by analyzing the activity and state diagrams of the various models [5,6]. In the UML context, many attempts have been made to generate test cases using state diagrams. Samuel et al. developed a predictive AVM that afforded path-based coverage using fewer test cases than previous methods [7]. However, given the limitations of an AVM, no globally optimal solution was found and coverage was not 100 percent. Gulia presented a path-based test case generation method; a parent solution was used to create a path-based child solution employing a genetic algorithm [8], but the method was not compared to other algorithms. Furthermore, these methods were not tested in real industrial situations. As a different approach, Choi and Lim generated test suites for fault localization from UML models [9]. Several efforts have been made to create test cases based on requirements manifested in use-case tables or diagrams [10]; these are advanced test scenarios reflecting customer demands. However, it is difficult to use a testing tool to create runnable test cases directly. Here, to study test case optimization employing the model-based development approach, we use a software based on the UML 2.1 standard. The Eclipse-based Papyrus open-source tool is a well-known UML-based tool [11]. Enterprise Architect [12] and IBM Rational Rhapsody are representative commercial UML tools. In this paper, Rhapsody is used as a UML model-based tool. Rhapsody uses its own target simulation framework, called the Object Execution Framework (OXF) to run code simulating the behavioral diagrams of the developed software model. However, Rhapsody is incompatible with fUML [13] [the executable UML standard of the Object Management Group (OMG)]. In fUML, action language is used to implement model behavior; a behavioral diagram corresponding to a platform-independent model is derived. The action language may be directly used for software simulation, or translated into platform-specific languages [14]. We use Rhapsody principally to create class, object, and state diagrams. C is employed to implement specific events, transitions, guards, and actions in state diagrams; we do not use an action language. To generate test cases automatically, we employ a concolic execution method in a custom parser-based simulation environment, instead of the OXF framework.

2.2. Test Case Optimization

During software development, it is often necessary to reimplement the software for several reasons such as changing customer requirements or to fix bugs. The three most widely used methods for test suite optimization are test case selection, minimization, and prioritization. We use each of these to optimize the test suite.

2.2.1. Test Case minimization

Test case minimization eliminates obsolete or redundant test cases from test suites. For example, if one test case identifies fault F_1, and another identifies fault F_2, they are combined into a new test case that identifies both faults F_1 and F_2. Such minimization not only reveals faults quickly but also ensures code coverage and reduces testing time.

2.2.2. Test Case Selection

A subset of test cases can be selected from changed parts of the software. The minimal number satisfies a specific objective function of the test suite; the suite is optimized using the selected cases. This is unlike test minimization; existing test cases are not modified, and only the required test cases are selected from the test suite T. The test case set becomes a new test suite T'. For example, the branch coverage of each test case can be measured in T, and test cases selected using an objective function to ensure 100 percent branch coverage; the minimum number of test cases in T required for this constitutes T'.

2.2.3. Test Case Prioritization

Prioritization changes the test case execution order to maximize early fault detection and minimize cost. When creating a new test suite T' from cases in an existing test suite T, the objective function described above is used to optimize the execution order of T. Unlike test case selection, the objective is achieved by changing test case execution priority rather than finding a minimum number of test cases. For example, if one test case finds more faults than any other case, it is run first. We use test case prioritization for regression testing.

2.3. Mutation Testing

If an error is accidentally inserted when modifying the code because of changed customer requirements, this can affect functionality. Mutation testing can be used to solve such problems. A mutant (a context error) caused by a developer is detected in otherwise correct software and a robust test case is selected to kill such mutants. High numbers of many types of mutants are required, optimally generated by a mutant operator. Hand-seeded mutants are closer to the errors made by real programmers, but are slow to create. Do found that automatic seeding was more efficient than hand seeding [15]; we use automatic seeding here. Various objective functions can be used to measure mutant killing. APFD is the most representative technique, measuring how quickly mutants can be killed during test case execution. APFD does not consider execution time or fault severity; APFDc does. Model-based mutation testing is an active field of research; mutants generated automatically or manually using UML or Matlab-based software models are killed [16,17]. As no code (rather a model) is mutated, operators appropriate for model-based development must be used. Aichenig studied model-based mutation testing using a mutation operator of a UML-based model [5]. Automatic test case generation based on code has been more widely studied than model-based mutation. Concolic execution-based test case generators including CREST and KLEE have been employed [18,19]. These methods combine concrete and symbolic approaches that complement their mutual disadvantages. However, such code-based, test case generation tools create cases only in specific languages. Code-based TCP has also been studied. Elbaum used APFD and APFDc when evaluating various TCP techniques in a code-based regression-testing environment [20–22]. Qi, Luo,

and colleagues performed static and dynamic TCP using a Java-based open-source program [23]. McMinn developed an open-source AVM for test data generation. Pradhan performed test case selection using a search-based algorithm featuring a time budget [24]. Arrieta prioritized test cases using search-based algorithms (a greedy algorithm and an AVM [25]).

In typical model-based mutation testing, mutations for the software model or specific diagrams are used to design robust test cases or optimize test cases. Bernhard K. Aichernig, the most active researcher of model-based mutation testing, studied how to perform test case selection on mealy machines using active automata learning techniques for conformance testing [26]. He also proposed a method to generate test cases in a block-box environment based on mutation testing in the model, such as statecharts, class diagrams, and instances diagrams [27]. Fevzi Belli et al. performed a test for fault detection at the system level from event sequence graphs, finite-state machines, and statecharts [28].

2.4. Search Techniques

Test case minimization, selection, and prioritization use different methods to obtain optimal solutions. The TCP that we employ improves the solution by executing the most effective test case first. Thus, a search-based algorithm is used to determine whether the current solution is optimized by a fitness function, and to search continuously for a solution that can be further improved; this is a heuristic approach. For example, if the fitness function measures statement or branch code coverage, a higher code coverage is achieved when the solution is continuously improved. We use search-based algorithms to optimize test cases based on mutation testing; the number of mutants killed by the optimized test suite is the fitness function. In TCP, the best solution kills many mutants early [29]. To measure this, most studies employed APFD. The ordering of test cases in a suite to optimize the objective function is an NP-hard problem; such problems are often solved using a search-based meta-heuristic algorithm such as a genetic algorithm, simulated annealing, and the AVM. The models allow genetically based automatic programming, test case generation, and cost reductions. Here, to find the best solution using an objective function, we optimized an AVM for local searching.

3. Model-Based Mutation Testing

First, test cases must be created. As described in Section 3.1, we generate test cases employing a UML model featuring a custom parser and an SMT solver. The cases are automatically generated from mutants by the model-based mutant operators described in Section 3.2, and the optimal test suite selected via TCP. Section 4 discusses the addition of a fitness function to several TCP methods. Section 5 reviews model-based TCP in the automotive domain; we study a power-window switch module, a body control module, and a passive entry and start system. Section 6 contains the conclusions.

3.1. Model-Based Test Data Generation

Unlike traditional test data generation by legacy codes, software is used for test data generation in the model-based development approach. UML is the most widely used descriptive language for structural and behavioral models. Currently, UML ver. 2.4 is implemented via Papyrus, Enterprise Architect, or IBM Rational Rhapsody; we use Rhapsody, which supports XML Metadata Interchange (XMI) (an OMG standard), and existing model information can be transferred from UML-based tools to other tools [30]. However, when using XMI, UML model information is imported unchanged, but other information must be re-processed. For example, structural diagrams are usually encoded in UML and can be XMI-processed, but model behavioral diagrams containing (perhaps) actions must be redrafted. In the 2000s, a standard action language was lacking; models were implemented using platform-dependent codes. However, commencing in 2010, OMG provided the Foundational Subset for UML for Executable Models (fUML) standard. It is now possible to develop a platform-independent model via a platform-independent action language when building behavioral diagrams of software models. Another OMG model-based testing standard, the UML testing profile (UTP), is also available. These standards create abstract test cases from platform-independent models, and configure

test environments that convert abstract tests to actual tests with real execution targets. However, one problem is that an abstract test is converted to an actual test by an executor that must engage in interpretation within a real test target environment during active run-time. This renders it difficult to use executors that do not meet specific, real-world industrial development standards. For example, in the automotive industry, testing software must conform to the ISO26262 safety standard for passenger vehicles; general UML model-based testing tools are not so certified. Below, we ensure that the automotive verification requirements are met. A custom parser generator has a role to play if the behavioral diagram of a UML model contains unknown elements. For example, if the state chart or activity diagram is described using an action language or C/C++, the behavioral description is parsed using parsers specific to these languages. The parsed code is expressed in SMT-LIB format and used to derive a solution with the aid of a general SMT solver. Yices and Z3 are the most popular SMT solvers supporting SMT-LIB 2.0; we use Yices.

3.2. State Chart-Based Mutation Operators

RQ1 was: What mutation operator should be used for model-based mutation testing? The Krenn model-based mutation-operator-related study is informative [5]. Structural and behavioral diagrams of model-based development techniques require mutation operators distinct from the traditional code-based operators. A model-based mutation operator recognizing a structural diagram (such as a class diagram) is analogous to a class-based mutation operator employed in object-oriented programming, and may engage in method overloading and the creation of inappropriate relationships. State diagrams may feature incorrect transition connections, wrong event calls, and state or transition deletions. Thus, logical, relational, statement deletion, and constant-value-change operators are the model-based mutation operators equivalent to code-based operators [28]. However, in a general model-based development environment, model checking is principally static; it is sought to verify model validity. Rhapsody supports model checking, detecting basic errors. Here, of the various mutation operators on state charts that cannot be detected by model checking, we used those listed on the below Table 1.

Table 1. Mutation Operators for State-Chart-Based Mutation Testing.

ID	Mutation Operator	Mutation Method	Formula
MO1	Replace call trigger	Replaces a trigger of a transition with another trigger.	$\delta : S_i \times T'(N, G, A) \rightarrow S_j$
MO2	Replace guard	Replaces an operand, a logical operator, or a relational operator on a guard.	$\delta : S_i \times T(N, G', A) \rightarrow S_j$
MO3	Replace guard to false	Ensures that a guard is false.	$\delta : S_i \times T(N, False, A) \rightarrow S_j$
MO4	Replace guard to true	Ensures that a guard is true.	$\delta : S_i \times T(N, True, A) \rightarrow S_j$
MO5	Replace action	Replaces an action on a transition or a state with another action.	$\delta : S_i \times T(N, G, A') \rightarrow S_j$
MO6	Replace incoming	Replaces an incoming transition to another state, excluding the default transition.	$\delta : S_k \times T(N, G, A) \rightarrow S_j$
MO7	Replace outgoing	Replaces an outgoing transition to another state, including the default transition.	$\delta : S_i \times T(N, G, A) \rightarrow S_k$
MO8	Remove state	Removes a state and its related transitions on a state machine.	$\delta = \emptyset, S_i = \emptyset$
MO9	Remove transition	Removes a transition on a state machine.	$T(N, G, A) = \emptyset$
MO10	Remove call trigger	Removes a call trigger on a transition.	$\delta : S_i \times T_{\emptyset}(N, G, A) \rightarrow S_j$
MO11	Remove guard	Removes a part of a guard condition.	$\delta : S_i \times T(N, G', A) \rightarrow S_j$
MO12	Remove action	Removes an action on a transition or a state.	$\delta : S_i \times T(N, G, \emptyset) \rightarrow S_j$

These mutation operators are used to create new mutants repeatedly that are not combined. To validate the mutants, the add or replace mutation operator uses the events, actions, inputs, and outputs of the original model. The mutant set is used to generate test cases for fault detection. Specifically, the original state chart and mutants generated by the mutation operator are employed to generate test cases that reveal the faults created by the mutants. If a mutant is selected, a concolic, execution-based, test case generator creates a test case that detects the fault. As validation proceeds, the test cases are added to the test suite. The mutant-based test generation process is performed for a predefined time or for a predefined number of times. Finally, the test cases are added to the test suite and prioritized.

Figure 1 shows an example statechart and its mutants for simple fade-in and -out features by the lamp on and off commands. The mutants generated by mutation operators in a UML-based state chart. The mutant operator falsifies a guard, which thus modifies the guard of any transition to become false in the original state chart (the first and second cases). If this can be expressed as a transition T, a 5-tuple is created. The state transition for any transition function δ can be expressed as a 6-tuple as follows

$$\delta : S_i \times T(N, G, A) \rightarrow S_j$$

where S_i is the source state, S_j is the target state, N the name, T a trigger (that may be the null), and G the guard of the transition. A is the combination of actions of the transition function. The mutation operator **MO3** of Figure 1b can be expressed as

$$\delta : S_0 \times T(N, False, A) \rightarrow \neg\, S_1$$

$$S_0\text{: } Root\ state$$

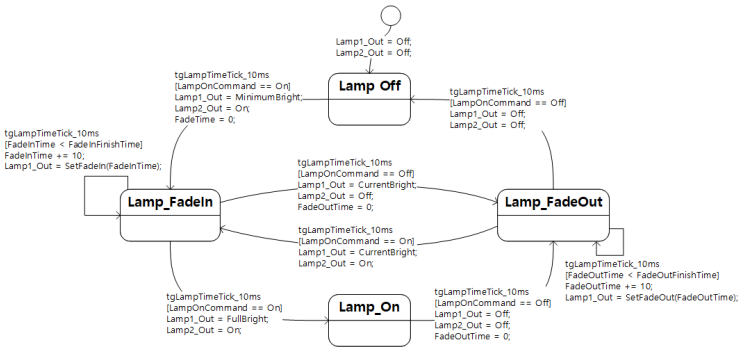
Thus, no transition can proceed to the next state S_1. However, if a transition from the root changes, it is impossible to enter any other state, and this scenario is discovered by the model checker. A mutant by the operator **MO3** of the Figure 1c can be expressed using very similar notation

$$\delta : S_2 \times T(N, False, A) \rightarrow \neg\, S_3$$

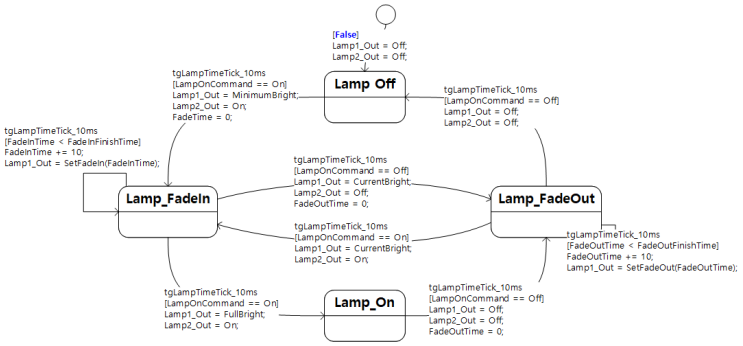
However, this transition may not be detected by the model checker because it is a general transition between states. The mutation operator **MO7** in the Figure 1d shows a mutant, the original state transition of which changes from $S_3 \rightarrow S_1$ to $S_3 \rightarrow S_2$.

$$\delta : S_3 \times T(N, False, A) \rightarrow\ S_2$$

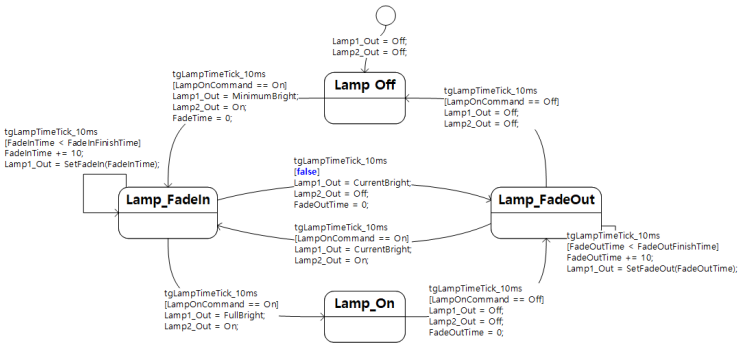
This mutation also may not be detected by the model checker. It is thus useful if the fault is detected early in iterative mutation testing.



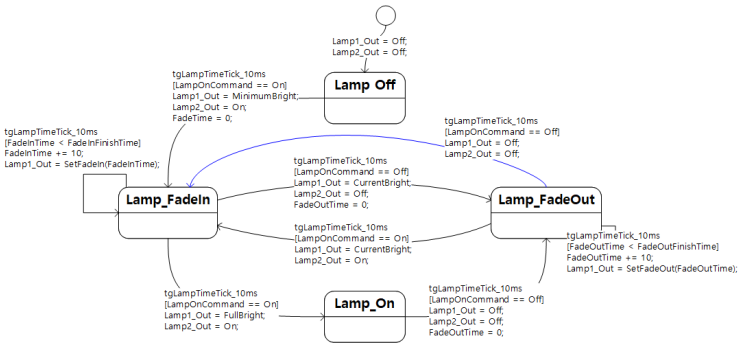
(**a**) The original state diagram for the lamp on and off control. This diagram is used to generate the mutants (b), (c) and (d)



(**b**) This mutated diagram by MO3 cannot be operated from first since the guard condition of the transition from Root to Lamp Off was changed to false.



(**c**) This mutated diagram by MO3 has a false guard on a transition from Lamp FadeIn to Lamp FadeOut.



(**d**) A target of the transition from Lamp FadeOut to Lamp Off was changed to Lamp FadeIn by MO7.

Figure 1. Examples of a model-based mutation operator.

3.3. Test Case Prioritization

Consider software S; the test suite used to test S is T. Term any source code modification in S S', and the test suite used to retest the changed software T'. During regression testing, the test suite T for S must achieve certain objectives. In particular, when the software change $S \rightarrow S'$ occurs, T should be used to examine whether the test cases meet the changed customer requirements. Some test cases in T can be retained without change, but a new test case must be added to T'. However, as software becomes more complex and test suite size increases with each regression cycle, it may not be possible to perform all tests in T' if resources are limited. Optimization techniques can be used to construct a new test suite T'. Coverage is a test exit criterion and an important factor as a metric for monitoring how much testing has been performed. Coverage includes structural coverage of a model or a source code, and fault coverage; both are used as measures of testing. For example, if a certain software must meet 10 requirements, all must be verified. On the other hand, to confirm efficient source code implementation, structural coverage confirms that all necessary tests meet certain conditions or decision-making criteria. Finally, fault coverage should reveal all faults. If the test suite meeting these criteria is optimized early, the resources required during each regression are low. APFD is used to measure how early the test suite reveals faults. For example, for 10 faults, suppose that the first test case reveals both faults 1 and 2. That test case is executed first because it is efficient. The APFD is calculated using the test case execution order for 10 faults:

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{n \times m} + \frac{1}{2n} \tag{1}$$

TF_1 : test case revealing the first fault.
TF_2 : test case revealing the second fault.
...
TF_i : test case revealing the ith fault.
n : number of test cases
m : number of faults

However, APFD assumes that all faults are equally severe and that all tests cost the same. The test order is not prioritized. APFDc prioritizes tests by fault severity and considers costs. To this end, an additional term is needed to reflect the severity and weigh the costs of Equation (1), as follows [21]:

$$APFDc = 1 - \frac{\sum_{j=1}^{m}(f_j \times (\sum_{i=1}^{n} t_i - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{j=1}^{m} f_j} \tag{2}$$

t_i : cost of the ith test case
f_j : severity of the jth fault
TF_i : test case order of test suite T revealing the ith fault

The cost shown above should reflect the time required for test case execution, but fault severities can be difficult to measure precisely. If all fault severities are assumed to be identical, the APFDc becomes more concise [23]:

$$APFDc = 1 - \frac{\sum_{j=1}^{m}(\sum_{i=TF_i}^{n} t_i - \frac{1}{2}t_{TF_i})}{\sum_{i=1}^{n} t_i \times m} \tag{3}$$

When we simulated real-world test case execution times on a PC, they did not differ markedly. Of course, test times in a black box and the real world may differ in terms of variables. However, in a white box, real execution times can be reduced. Thus, the APFD serves as an objective function.

Model-based mutation testing has attracted much attention. Usually, mutants are generated by a software model based on UML or Matlab, as are the test cases [16,17]. Mutation operators are required in model-based environments, but not for code mutation. Bernhard et al. studied model-based mutation testing using mutation operators of a UML-based model [5].

3.4. The Alternating Variable Method (AVM)

The AVM is first applied to search locally for numeric test data, and then to find an optimal local solution in the search space [31]. For example, if three input vectors a to c exist, an arbitrary vector is first selected. Assuming that a is selected, the inputs are created by changing a with b and c held fixed; fitness ranges from 0 to 1, where 1 indicates optimal fitness. When the fitness attains 1, fitting is paused because local optimization has been achieved. This entire process repeats continuously [32]. However, as the AVM method is used principally for local optimization, a global search algorithm is also required. We perform local optimization in this manner; when a fitness of 1 is not obtained in one local optimization, another solution is sought. If a global solution is optimal, it is improved further via local optimization.

As above Algorithm 1, sequential AVM operation is performed on the test suite; if no improvement occurs over three attempts, the algorithm seeks a solution in another space. If a better solution is found, AVM-based prioritization finds a local optimization within that space; this becomes the globally optimal solution. The greatest advantage of this method is that the algorithm seeks a globally optimal solution that prevents underconvergence and also quickly finds local optimal solutions.

Algorithm 1 The search-based test case prioritization using the alternating variable method (AVM)

```
1:  while termination criterion is not matched do
2:    S_AVM=AVMSearch(S,n)
3:    if S is not improved in last 3 loops then
4:      S'=exploratorySearch(S)
5:    end if
6:    if fitness(S') > fitness(S) then
7:      S''=AVMSearch(S')
8:      if fitness(S'') > fitness(S') then
9:        S = S''
10:     else
11:       S = S'
12:     end if
13:   end if
14:   n = n + 1
15: end while
```

4. The Triangle Classification Example

Test prioritization algorithms based on traditional code have been used principally to ensure code coverage [33] of code-based development and model coverage of model-based development. However, the goal of mutation testing is different. During TCP that reveals faults quickly, the mutation operator applicable during model-based development differs from that of the code-based development environment. The mutants and faults differ, as do the TCP results. Here, we generate mutants using the mutation operators of Table 1 and perform TCP, employing an objective function that checks whether mutants are killed early. APFD both ensures coverage and exposes faults in a test suite. APFD is a low-cost multi-objective function, performing prioritization with consideration of time and resource costs. We use APFD for objective prioritization; earlier fault detection (in the absence of severity or likelihood weighting) serves as an indicator of efficiency. APFD objectively evaluates test suites before and after test case optimization and measures whether fault exposure improves after prioritization. RQ2 was: Which prioritization technique allows optimal fault detection? Is this better

than the code-based approach? We first performed TCP using APFD as a fitness function employing the following techniques:

- **Random**: After generating a test suite to kill mutants generated by mutation operators, the suite is subjected to random prioritization. If the algorithm runs only once, the differences between this and other algorithms is large. Thus, the number of random permutations is identical to the number of test cases. This determines whether the randomly generated tests allow fitness evaluation. The computational complexity is $O(n)$;
- **Statement total (StTotal)**: Statement coverage is one of code-based coverage to confirm whether all statements are covered or not. For model coverage, this prioritization is performed using an objective function to cover all possible transitions of the state diagram; the coverage is 100 percent; The computational complexity is $O(n)$.
- **Statement additional (StAddtl)**: This is similar to the above method but, after selecting a reference test case, prioritization (by a greedy algorithm) selects another case that additionally covers another transition. The complexity is $O(n^2)$;
- **Fault exposure probability total (FepTotal)**: Test case execution is prioritized in the order of fault detection probability. The first-selected test suite maximally increases the total probability when the fault is exposed; the numbers of exposed faults are always maximized. The computational complexity is $O(n)$;
- **FEP additional (FepAddtl)**: This is similar to the above method, but basic prioritization employs a greedy algorithm. The fault exposure probability always increases when the next test case is added. The computational complexity is $O(n^2)$;
- **AVM**: A permutation is first prioritized using a fitness function, and the prioritization method, then locally optimized. As only local optimization is possible, an exploratory method is added to find global solutions even when local optimization using the AVM fails after a defined number of trials. The computational complexity is $O(nlogn)$.

We generated 100 different mutants using a simple UML state diagram that selects triangle type by reference to the lengths of the sides, as shown in the Figure 2. During mutation testing, only test cases that killed mutants were included in the test suite. To prevent that number from becoming too large, if the number of cases in the suite exceeded a certain number, only test cases that killed more mutants were added. This prioritized test case execution. We obtained APFD data for each prioritization method based on optimized test suites.

Statistical model: We explored whether data generated via prioritization were normally distributed. The numbers of mutants and test cases in the test suites were determined. One hundred mutants were generated using mutation operators, and test cases created to kill the mutants. Those cases were prioritized 100 times. ANOVA analyses of the average between-group differences rejected the null hypothesis (no between-group difference) at less than 0.005 in Table 2; the differences were, in fact, significant. A total of six methods were compared including five code-based TCP methods and our suggested AVM-based TCP method; the degree of freedom is 5 in between-groups, and the degree of freedom of within-groups is 594, since we tested 100 times.

Table 2. One-way ANOVA results.

Category	Sum of Squares	df	Mean Squares	F	*p*-Value
Between-groups	0.553	5	0.111	4635.015	<0.005
Within-groups	0.014	594	0.000	-	-
Total	0.567	599	-	-	-

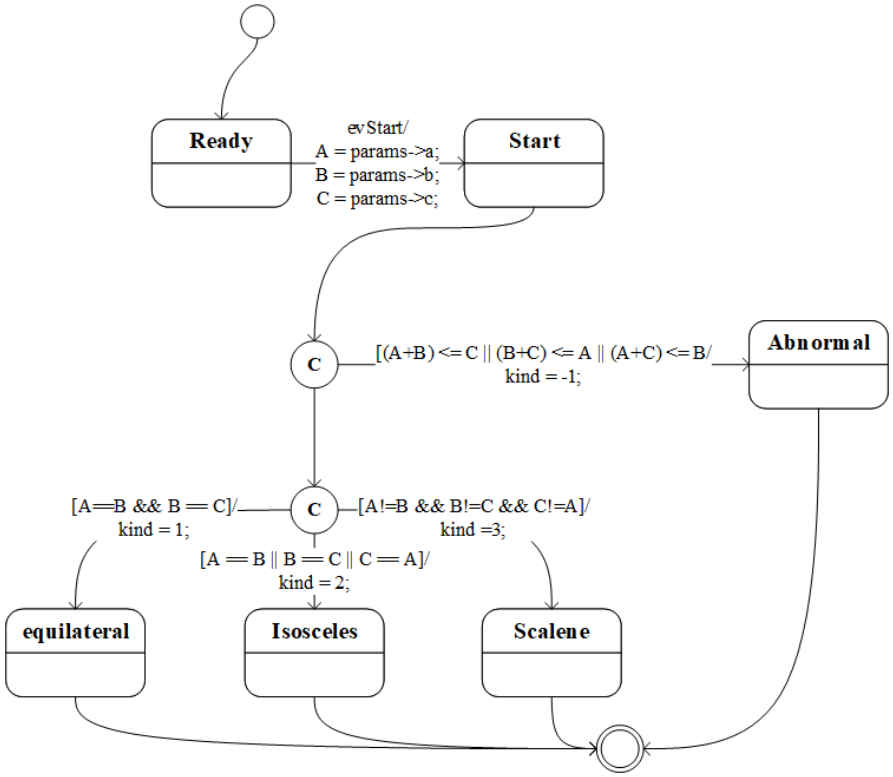


Figure 2. A state diagram of triangle classification.

We used the post-hoc Scheffe method and represented the result in Table 3. In most cases, the mean, post-test between-group difference (p) was <0.005. No clear difference was apparent between FepAddtl and AVM.

As shown in Figure 3, the Random, StTotal, and StAddtl performed less well than FepTotal and FepAddtl. The former methods achieved code or model coverage regardless of fault exposure, and thus exhibited low APFDs. The StAddtl APFD was lower than the Random APFD because Random creates as many test suites as test cases (computational complexity $O(n)$), associated with a higher APFD. However, FepTotal, FepAddtl, and AVM exhibited relatively good TCP. As APFD served as the fitness function, good prioritization reflected good performance. FepAddtl was better than FepTotal because the test suite detected faults with ever-increasing probability, driven by a greedy algorithm. The AVM was significantly better than FepTotal. Usually (compared to AVM), FepTotal found a good solution in a shorter time because the computational complexity of AVM is $O(nlogn)$, but that of FepAddtl is $O(n^2)$. We ran the algorithms on a PC(Intel® Core™ i7-9750H with RAM 24GB) and the average execution times of the algorithms for 100 times were found to be as follows: Random-413 ms, StTotal-13 ms, StAddtl-828 ms, FepTotal-3 ms, FepAddtl-5186 ms, and AVM-3834 ms. As explained in the second paragraph of this section, the time complexities of the algorithms are as follows: Random-$O(n)$, StTotal-$O(n)$, StAddtl-$O(n^2)$, FepTotal-$O(n)$, FepAddtl-$O(n^2)$, AVM-$O(nlogn)$. Since FepAddtl and AVM do the calculations to select the next fault exposure test case in every loop, the more computation power is required than in the code coverage calculation. They require more execution time than StAddtl, but give better APFD results. Finally, AVM's execution time is shorter than FepAddtl, and the APFD result of AVM is the best among other TCP algorithms. However, sometimes, after AVM convergence to a local optimization point, the results no longer improved. This occurred 6–8 times in 100 iterations; it is essential that a solution does not converge only to a local point. Notably, our code-based TCP did not differ from that of Elbaum et al. [20], because some algorithms are not optimized in terms of fault exposure.

Table 3. The outcomes of test case prioritizations.

(I) Method	(J) Method	Average Difference (I − J)	p
Random	StTotal	0.003824646490	<0.05
	StAddtl	0.004647020250	<0.05
	FepTotal	−0.051540050500	<0.05
	FepAddtl	−0.060530909030	<0.05
	AVM	−0.060551111040	<0.05
StTotal	Random	−0.003824646490	<0.05
	StAddtl	0.000822373760	≥0.05
	FepTotal	−0.055364696990	<0.05
	FepAddtl	−0.064355555520	<0.05
	AVM	−0.064375757530	<0.05
StAddtl	Random	−0.004647020250	<0.05
	StTotal	−0.000822373760	≥0.05
	FepTotal	−0.056187070750	<0.05
	FepAddtl	−0.065177929280	<0.05
	AVM	−0.065198131290	<0.05
FepTotal	Random	0.051540050500	<0.05
	StTotal	0.055364696990	<0.05
	StAddtl	0.056187070750	<0.05
	FepAddtl	−0.008990858530	<0.05
	AVM	−0.009011060540	<0.05
FepAddtl	Random	0.060530909030	<0.05
	StTotal	0.064355555520	<0.05
	StAddtl	0.065177929280	<0.05
	FepTotal	0.008990858530	<0.05
	AVM	−0.000020202010	≥0.05
AVM	Random	0.060551111040	<0.05
	StTotal	0.064375757530	<0.05
	StAddtl	0.065198131290	<0.05
	FepTotal	0.009011060540	<0.05
	FepAddtl	0.000020202010	≥0.05

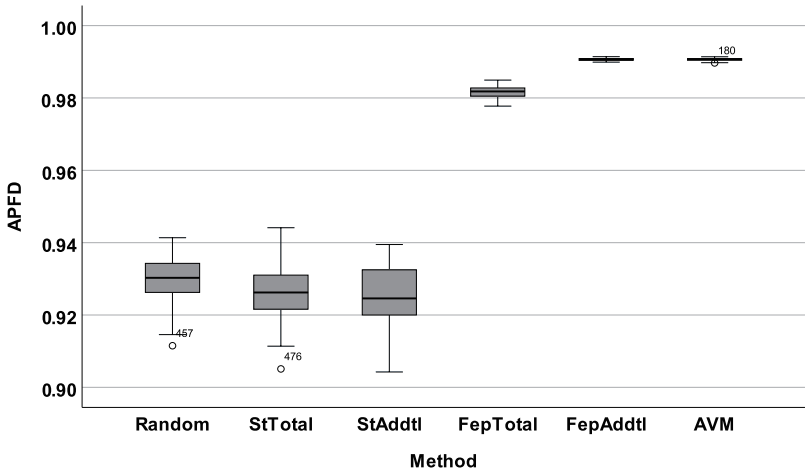


Figure 3. A comparison of test case prioritization methods.

5. Empirical Studies: Industrial Cases

We explored whether our model-based prioritization could be applied to real industrial UML models for the automotive industry (specifically: models of chassis parameters, body electronics, and engine controls). Most prior empirical studies have focused on automotive electronics, many of which are simple, with outputs determined by the combinations and the timings of input signals. In Section 4, we showed how to generate mutations using mutation operators that seeded the state

charts of model-based tests in a manner unlike code-based TCP. This Section seeks answers to RQ2 and RQ3 (Which prioritization technique allows optimal fault detection? Is this better than the code-based approach?; Is the optimal search algorithm for model-based TCP the same as the code-based algorithm? Does this work for real industrial cases?) via empirical studies. Section 5.4 analyzes threats to the validity of such results. We prioritized test cases for body electronic components: the power-window switches on the driver and front passenger doors; the wiper, interior, and exterior lamps; central door controls; seat belt controls; the burglar alarm; and the smart key (passive entry/start) system. A mutant kill table explored whether a new test case killed mutants automatically generated by mutant operators. Using this table, the TCP satisfying a fitness function was statistically analyzed.

5.1. Power-Window Switch Module

Test cases were generated from a UML model for the power-window switch (PWS) module; iterative test execution was optimized via prioritization of the test suite. The module has several submodules controlling door open/closed status, door locking/unlocking, and side mirror control from the driver and passenger seats. The module is connected to the body controller area network (CAN) and receives input from other modules (via the CAN interface) to operate the power-window controls or side-mirror actuators. The module studied is UML-based and is implemented in the Kia Sportage using Rhapsody. The model was loaded into a test case generation program via XMI transformation, and mutants were generated in model state charts using the mutation operators of Section 3.2. The results were compared after 100 test suite prioritizations by each algorithm in the test suite generated to kill 100 mutants.

Table 4. Major functionalities of the power-window switch module.

Module	Number of States	Number of Transitions	Number of State Machines	Number of Test Cases	*p*-Value
Courtesy Lamp	12	26	4	1135	<0.005
OS Mirror	109	349	32	1702	<0.005
Power window	78	126	26	1780	<0.005
Puddle lamp	14	26	5	887	<0.005
Interface	26	39	13	1719	<0.005
Heater	4	6	2	1294	<0.005
IMS	27	38	4	1295	<0.005
Total	270	610	86	9812	

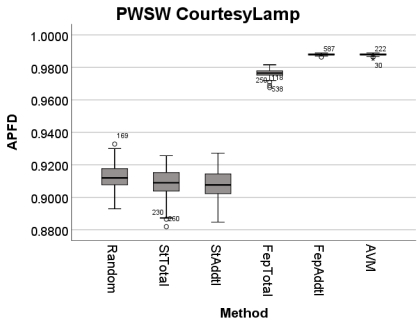
The PWS module featured seven principal submodules in Table 4 and 86 state machines. The test cases generated were proportional to the numbers of mutants, and test cases exhibiting various paths were generated within the state chart even if the module size was small. Thus, similar numbers of test cases were generated for the heater and other more complex functions; the heater submodule is very small. APFD served to measure the fitness of TCP; the outcomes of various prioritization methods were compared. We first confirmed that the data were normally distributed. For each TCP method, the mutation operator created 100 mutants and test cases were then generated automatically. The results (100 for each method) were analyzed as described in Table 5. We confirmed that the APFD values became essentially normally distributed as the test numbers rose; we used ANOVA to explore the statistical significances of APFD differences. We also employed the Duncan post-hoc test; the p-value was <0.005 and was correlated among the groups. When the Scheffe test (which is stricter than the Duncan test) was applied, the results of the Random, StTotal, StAddtl, and FepTotal methods differed significantly. However, the StTotal and StAddtl, and FepAddtl and AVM methods sometimes did not differ significantly (depending on the submodule). A boxplot of the average APFD comparisons follows.

As shown in Figure 4, the TCPs of the seven submodules for the PWS module did not differ significantly. First, test coverage prioritization focusing on statement coverage exhibited relatively low

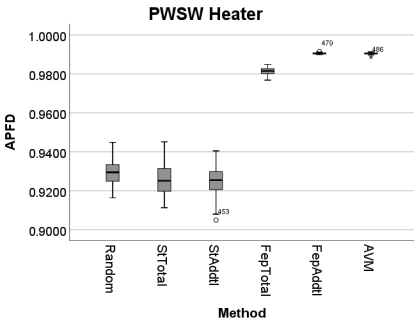
APFDs, as did code-based prioritization, probably reflecting the low correlation between coverage and fault discovery. For the random method, APFD revealed an improved prioritization method based on statement coverage given the iterative operation of $O(n)$, but the APFD remained lower than those of methods based on fault-exposure probability. Compared to the FepAddtl or AVM, the other methods are not locally optimized, delivering locally convergent results. The fault-exposure probability methods exhibited higher APFDs. Of these, FepAddtl uses a greedy algorithm, yielding better results than FepTotal because the algorithm prioritizes the direction in which fault exposure probability maximally improves. The AVM (our preferred choice) was not very different from FepAddtl; the significance levels were within 0.05 percent (depending on the submodule) but AVM was faster. The AVM result was clearly better than that of FepTotal, and equal to or better than that of FepAddtl.

Table 5. Scheffe post-hoc test results for the PWS Puddle Lamp.

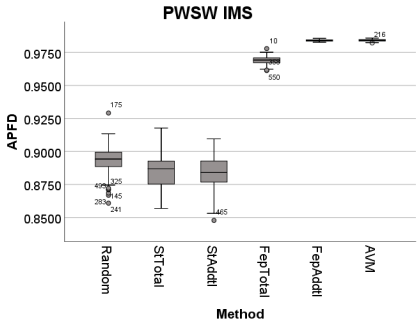
(I) Method	(J) Method	Average difference (I − J)	*p*-Value
Random	StTotal	0.004165355370	<0.005
	StAddtl	0.003424223600	≥0.005
	FepTotal	−0.069128709500	<0.005
	FepAddtl	−0.082282539710	<0.005
	AVM	−0.082333333370	<0.005
StTotal	Random	−0.004165355370	<0.005
	StAddtl	−0.000741131770	≥0.005
	FepTotal	−0.073294064870	<0.005
	FepAddtl	−0.086447895080	<0.005
	AVM	−0.086498688740	<0.005
StAddtl	Random	−0.003424223600	≥0.005
	StTotal	0.000741131770	≥0.005
	FepTotal	−0.072552933100	<0.005
	FepAddtl	−0.085706763310	<0.005
	AVM	−0.085757556970	<0.005
FepTotal	Random	0.069128709500	<0.05
	StTotal	0.073294064870	<0.005
	StAddtl	0.072552933100	<0.005
	FepAddtl	−0.013153830210	<0.005
	AVM	−0.013204623870	<0.005
FepAddtl	Random	0.082282539710	<0.05
	StTotal	0.086447895080	<0.005
	StAddtl	0.085706763310	<0.005
	FepTotal	0.013153830210	<0.005
	AVM	−0.000050793660	≥0.005
AVM	Random	0.082333333370	<0.05
	StTotal	0.086498688740	<0.005
	StAddtl	0.085757556970	<0.005
	FepTotal	0.013204623870	<0.005
	FepAddtl	0.000050793660	≥0.005



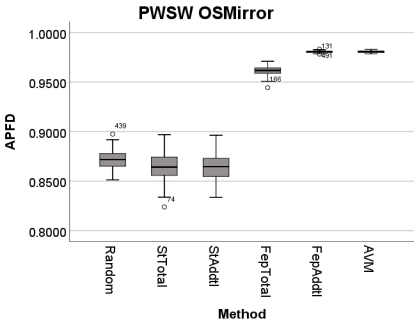
(**a**) Courtesy Lamp module FepAddtl is better than AVM more 0.00002.



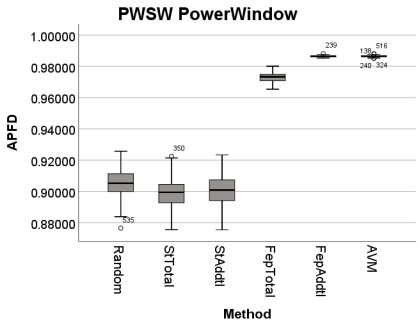
(**b**) Heater: FepAddtl is better than AVM more 0.00002.



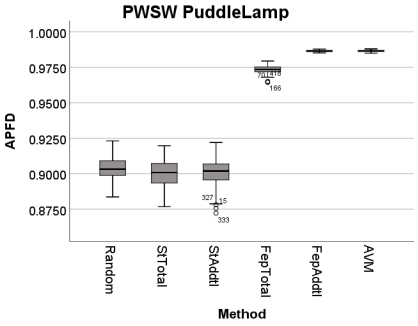
(**c**) IMS: AVM is the best as 0.9842.



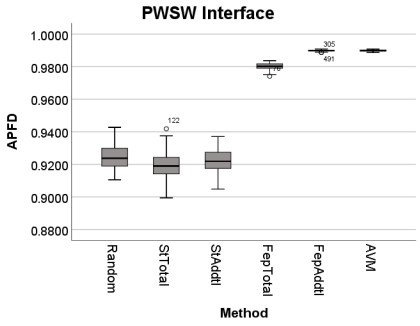
(**d**) OSMirror: AVM is the best.



(**e**) PowerWindow: AVM is better than FepAddtl.



(**f**) PuddleLamp: AVM is the best



(**g**) Interface: AVM is the best

Figure 4. Test case prioritization results of the power-window switch module.

5.2. Body Control Module

The body control module (BCM) core module controls all vehicle electronics via CAN and LIN communication featuring analog and digital inputs and output. The BCM model studied is that of the Kia Ceed. The BCM controls the tailgate lamp, burglar alarm, wipers, washers, side mirrors, power windows, seat belt reminders, central door control, remote key entry, and flasher lamp. The BCM software features 270 states and 610 transitions (Table 6). Each submodule features up to 73 state transitions depending on its complexity. As described in Section 5.1, the UML-based model was limited to 100 mutants for each submodule. Mutation-based testing employed 1000 to 2000 test cases for each submodule depending on submodule complexity and mutant type. However, the number of test cases was limited, but adequate in terms of coverage. The test suites were prioritized using the various methods. As Figure 5 shows, the prioritization test results of the more complex submodules exhibited lower APFDs than the tests of simpler submodules. A low APFD indicates that it is difficult to kill many mutants at once during test case creation. We group the methods as follows: Group 1: Random, StTotal, and StAddtl; Group 2: FepTotal, FepAddtl, and AVM. The APFD values of the first group were 60 percent of those of the second group for complex submodules, and 90 percent for simple submodules. This is because the prioritization methods of group 1 do not adequately expose faults, as mentioned in Section 4. It is essential to prioritize the test suite appropriately because it is difficult to kill multiple mutants simultaneously in more complex modules. Therefore, Random, StTotal, and StAddtl did not yield good APFD values.

Table 6. Functionalities of the body control model (BCM).

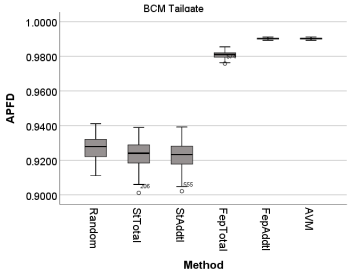
Module	Number of States	Number of Transitions	Number of Test Cases	*p*-Value
Tailgate	8	16	1857	<0.005
OS Mirror	10	73	1583	<0.005
Defroster	7	7	1597	<0.005
Driving	2	3	1329	<0.005
Interface	5	11	1910	<0.005
Interior lamp	2	5	1765	<0.005
Power window	8	9	1785	<0.005
Remote key entry	1	4	1253	<0.005
Flasher	3	37	1017	<0.005
Warning	5	23	1945	<0.005
Wipers/washers	2	5	1657	<0.005
Exterior lamp	2	3	1915	<0.005
Mirror	3	7	1409	<0.005
Door locks	18	48	1528	<0.005
Total	76	251	22,550	-

5.3. Passive Entry Passive Start System

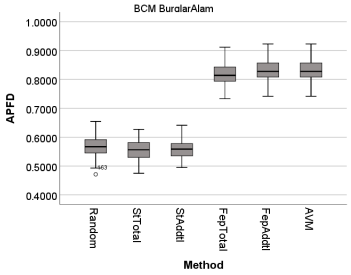
Passive entry passive start (PEPS) systems (smart keys) allow vehicle entry and starting (by pushing a button) without a physical key. The UML-based software used was that of the Accent of Kia Motor Corporation. As for the PWS module, data were XMI-transformed to the UML format and test cases were generated via mutation. Prioritized test suites were created using various methods and the results compared using the APFD. The PEPS system features diagnostic and key learning submodules, and others (see the table below). However, the other submodules are implemented in legacy code rather than UML, and were thus not evaluated.

As shown in Table 7, each module features several state diagrams. Overall, the state diagrams of small units exhibit few states or transitions. However, the Start submodule features complex state diagrams including several substates. The number of submodules is lower than those of the PWS or BCM, but there are more state diagrams per submodule. If the number of mutants for each module is

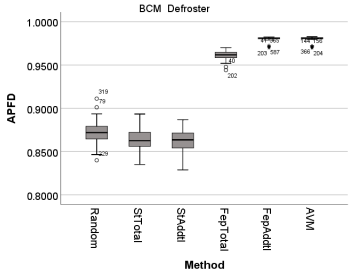
over 100, the time needed for mutation testing will be excessive. We limited the number of mutations to 100 and the number of test cases to 2000; the test suites were prioritized using various methods.



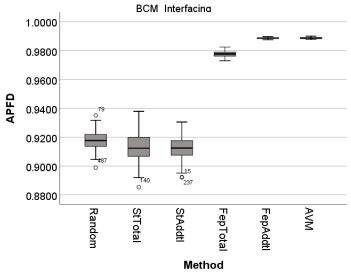
(**a**) Tailgate: AVM is slightly better than FepAddtl.



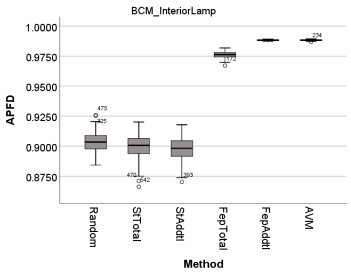
(**b**) BurglarAlarm: AVM is the best.



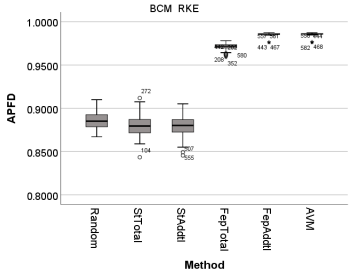
(**c**) Defroster: AVM is better than other algorithms as 0.9801.



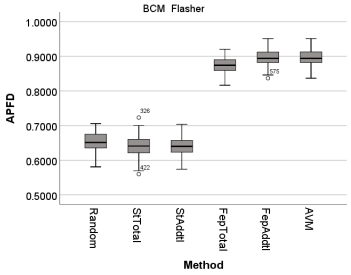
(**d**) Interface: AVM is the best.



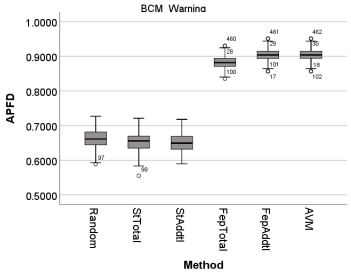
(**e**) InteriorLamp: AVM is the best.



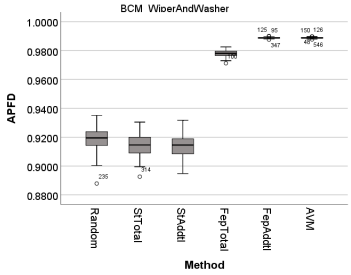
(**f**) RKE: AVM is better than 0.0001 than the result of FepAddtl.



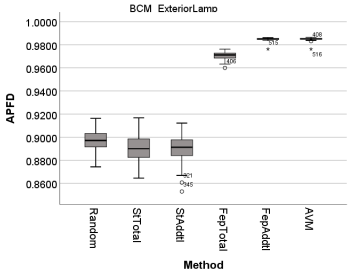
(**g**) Flasher: AVM is very slightly better than FepAddtl.



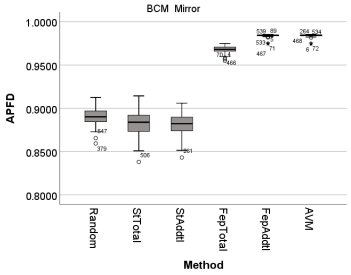
(**h**) Warning: AVM is the best.



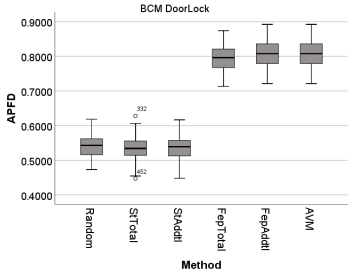
(**i**) WiperAndWasher: AVM is the best.



(**j**) ExteriorLamp: AVM is the best.



(**k**) Mirror: AVM is little better than FepAddtl.



(**l**) DoorLock: AVM is very slightly better than FepAddtl.

Figure 5. APFD results of the test case prioritization methods.

Table 7. ANOVA test results of passive entry start systems (PEPS) submodules.

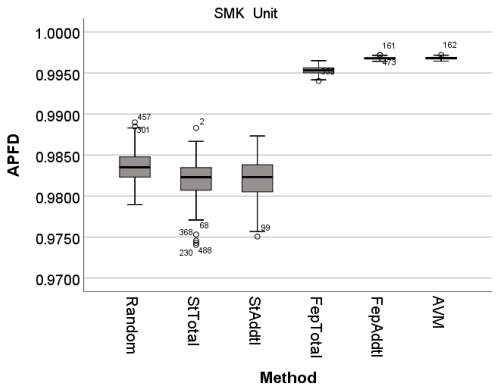
Module	Number of State Diagrams	Number of Test Cases	F	*p*-Value
Unit	81	1387	4389.207	<0.005
Access	13	1993	1631.736	<0.005
RKE	11	1799	4267.017	<0.005
Start	44	1385	3139.672	<0.005
Warning	49	1755	4833.233	<0.005
Total	198	8319	-	-

First, when comparing the FepAddtl and AVM results for the Unit submodule in the upper left panel of Figure 6, it is clear that the AVM result was best (about 0.0002). For the Access module (upper right panel), the AVM and FepAddtl results were 0.94408 and 0.9446. The state diagram of the Start submodule features sub-states and sub-state diagrams, and is complicated; the APFD is lower than that for other submodules. It was difficult to create test cases that killed mutants on mutation testing. Thus, the APFDs of the FepTotal, FepAddtl, and AVM test suites were similar. For the Warning submodule, the average APFD differences between AVM and the other methods were less than 0.00015 and, thus, negligible. In terms of overall prioritization, Random, StTotal, and StAddtl did not yield good results in most cases compared to the other methods. Commonly, the APFD results of AVM were clearly superior to those of FepTotal, and similar to those of FepAddtl. However, as for the PWS (Section 5.1), the Scheffe post-hoc test revealed no significant difference between FepAddtl and AVM. The difference between FepTotal, on the one hand, and FepAddtl and AVM, on the other, was significant when the APFD test suite values exceeded 0.9.

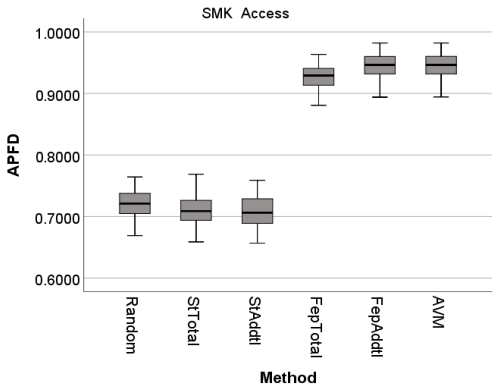
5.4. Results

We used a total of five existing code-based TCP methods for model-based mutation testing. The major reason is that we wanted to compare the model-based TCP results with code-based TCP results. In the three empirical studies, Random, StTotal, and StAddtl afforded APFDs lower than those of FepTotal and FepAddtl regardless of model complexity. In most cases, the model-based TCPs did not differ markedly from those of general code-based methods. In terms of answering RQ2, FepAddtl showed the best results among those five methods excluding AVM, equivalent to those of code-based TCP. The reason that FepAddtl was the best optimal method was because it can achieve high exposure probability for the faults in the early stage of development. However, this does not mean that all test results will be the same; our experimental environment was limited. To answer RQ3, we used a TCP method employing the AVM to create a search-based algorithm. All results were obtained under the same conditions, thus, without optimization; the APFD values ranged from 0.01 to 0.25 depending on software complexity. The APFD figures fell as software complexity increased. The answer to RQ3 is that a search-based algorithm can be used for model-based TCP in the same manner as a code-based technique. In the above industrial scenarios, the AVM APFD results did not differ greatly from those of other algorithms, but the AVM was quicker. The search time, of course, depends on local optimization and the exploration algorithm employed. Fault seeding threatens internal validity. To minimize this, we used mutant operators of model-based state diagrams as seeds. To ensure reliable data collection within a reasonable time, all experiments were 100-fold iterated for each algorithm, yielding statistically robust comparisons. However, this does not mean that the APFD results of TCP methods will always be the same. We constructed an external validity threat and explored the effect thereof on TCP, but this did not verify all software models. To minimize the problem, three empirical cases were considered; we evaluated the software of the PWS, SMK, and PEPS. All models were based on UML, but only the state diagrams were used to draw behavioral diagrams. This affects test case generation and prioritization. For example, if a test suite is generated using a behavioral diagram such as a sequence diagram, this may affect the results even if the operation of both diagrams is identical. In conclusion, most search-based algorithms have the same problems, but the experimental results

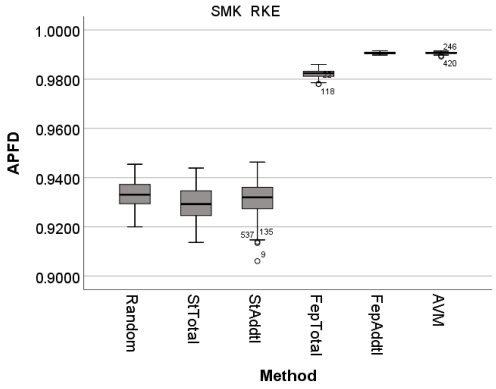
differ greatly if small amounts of data are compared, not always indicating the best algorithm. To deal with this issue, significance was analyzed via post-hoc comparisons of 100 experimental runs.



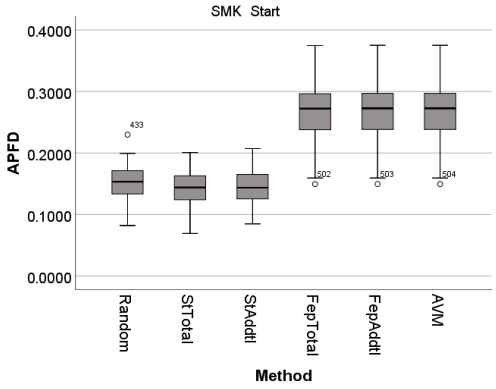
(**a**) Unit: AVM is the best.



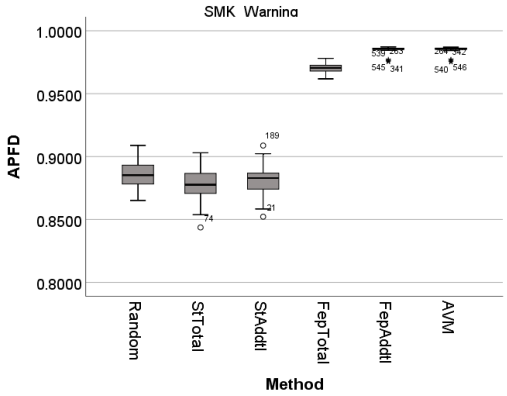
(**b**) Access: AVM is the best.



(**c**) RKE: AVM is higher only 0.000005 than FepAddtl.



(**d**) Start: The average value of AVM is equal to FepAddtl.



(**e**) Warning: AVM is the best.

Figure 6. Test case prioritization for the PEPS System.

6. Conclusions

We present TCP for mutation testing of a UML-based model; we employ a model-based development approach. Unlike the mutation operators used for code-based mutation testing, model-based mutation operators create mutants in the state machine, thus the behavioral diagram. Test cases were created to kill such mutants; our new prioritization method is a search-based algorithm employing the AVM. Code-based TCP methods were compared with our new method; we collected the APFD values of test suites prioritized using various methods. Our AVM-based, search-based TCP method is simpler and yields equivalent or better results than other methods. We empirically studied representative PWS, BCM, and PEPS software of the automotive industry. Mutants were created in UML-based models for mutation testing. Test cases were generated to kill the mutants. The test suites were prioritized using code-based algorithms and our AVM-based search-based algorithms. Our method was better than other methods in all scenarios tested. Our principal contribution is that we explored whether traditional code-based mutation operators could be used for model-based TCP, and the types of such mutation operators. We also developed a new AVM-based prioritization method. The AVM (a local search method) must be combined with an exploratory method to ensure nonconvergence at a local optimum. In future, we will explore whether the results of the various TCP methods change when other fitness values (for example, APFDc) are used. Other search-based algorithms (including a genetic algorithm) will be employed for model-based test case prioritization. Such prioritization must be readily applicable and valuable in the real world.

Author Contributions: Conceptualization, K.-W.S.; methodology, K.-W.S.; software, K.-W.S.; statistical analysis, K.-W.S.; investigation, K.-W.S.; data curation, K.-W.S.; writing—original draft preparation, K.-W.S.; writing—review and editing, D.-J.L.; visualization, K.-W.S.; supervision, D.-J.L.; project administration, D.-J.L.; funding acquisition, D.-J.L. Both authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the World Class 300 project (10050405) of the Ministry of Trade, Industry and Energy (MOTIE) and the Small and Medium Business Administration (SMBA) of South Korea.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ali, S.; Zohaib Iqbal, M.; Arcuri, A.; Briand, L.C. Generating Test Data from OCL Constraints with Search Techniques. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1376–1402. [CrossRef]
2. Shin, K.W.; Lim, D.J. Model-based automatic test case generation for automotive embedded software testing. *Int. J. Automot. Technol.* **2018**, *19*, 107–119. [CrossRef]
3. Shin, K.W.; Kim, S.; Park, S.; Lim, D.J. Automated Test Case Generation for Automotive Embedded Software Testing Using XMI-Based UML Model Transformations. *SAE Tech. Pap.* **2014**, *1*. [CrossRef]
4. Shin, K.W.; Lee, J.H.; Kim, S.S.; Park, J.U.; Lim, D.J. Automatic Test Case Generation Based on the UML Model of the Automotive-Embedded Software using a Custom Parser and SMT Solver. In Proceedings of the JSAE 2016 Annual Congress, Yokohama, Japan, 25–27 May 2016; pp. 1198–1202.
5. Aichernig, B.K.; Auer, J.; Jöbstl, E.; Korošec, R.; Krenn, W.; Schlick, R.; Schmidt, B.V. Model-Based Mutation Testing of an Industrial Measurement Device. In *Tests and Proofs*; Seidl, M., Tillmann, N., Eds.; Springer International Publishing: Cham, Switzerland, 2014; pp. 1–19. [CrossRef]
6. Utting, M.; Pretschner, A.; Legeard, B. A Taxonomy of Model-Based Testing Approaches. *Softw. Test. Verif. Reliab.* **2012**, *22*, 297–312. [CrossRef]
7. Samuel, P.; Mall, R.; Bothra, A.K. Automatic test case generation using unified modeling language (UML) state diagrams. *IET Softw.* **2008**, *2*, 79–93.:20060061. [CrossRef]
8. Gulia, P.; Chillar, R. A new approach to generate and optimize test cases for UML state diagram using genetic algorithm. *ACM Sigsoft Softw. Eng. Notes* **2012**, 1–5. [CrossRef]
9. Choi, Y.M.; Lim, D.J. Model-Based Test Suite Generation Using Mutation Analysis for Fault Localization. *Appl. Sci.* **2019**, *9*, 3492. [CrossRef]

10. Heumann, J. Generating Test Cases From Use Cases. Technical Report, Rational Software. 2001. Available online: https://www.ibm.com/developerworks/rational/library/content/RationalEdge/jun01/GeneratingTestCasesFromUseCasesJune01.pdf (accessed on 15 June 2001).
11. Eclipse Papyrus. Available online: https://www.eclipse.org/papyrus/ (accessed on 15 June 2020).
12. Enterprise Architect 15.2. Available online: https://sparxsystems.com/products/ea/ (accessed on 27 August 2020).
13. Vörös, A.; Micskei, Z.; Konnerth, R.; Horváth, B.; Semerath, O.; Varro, D. On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf. In Proceedings of the 1st Workshop on Open Source Software for Model Driven Engineering, Valencia, Spain, 28 September 2014.
14. Ciccozzi, F.; Malavolta, I.; Selic, B. Execution of UML models: A systematic review of research and practice. *Softw. Syst. Model.* **2018**, *18*, 2313–2360. [CrossRef]
15. Hyunsook D.; Rothermel, G. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Trans. Softw. Eng.* **2006**, *32*, 733–752. [CrossRef]
16. Aichernig, B.; Jöbstl, E.; Tiran, S. Model-based mutation testing via symbolic refinement checking. *Sci. Comput. Program.* **2015**, *97*. [CrossRef]
17. Zhan, Y.; Clark, J.A. Search-Based Mutation Testing for Simulink Models. In Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, GECCO '05, Washington, DC, USA, 25–29 June 2005; ACM: New York, NY, USA, 2005; pp. 1061–1068. [CrossRef]
18. Kim, M.; Kim, Y.; Jang, Y. Industrial Application of Concolic Testing on Embedded Software: Case Studies. In Proceedings of the IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, 17–21 April 2012. [CrossRef]
19. Kim, Y.; Kim, M.; Kim, Y.; Jung, U.J. Comparison of Search Strategies of KLEE Concolic Testing Tool. *J. KIISE Comput. Pract. Lett.* **2012**, *18*, 321–325.
20. Elbaum, S.; Malishevsky, A.G.; Rothermel, G. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.* **2002**, *28*, 159–182. [CrossRef]
21. Elbaum, S.; Malishevsky, A.; Rothermel, G. Incorporating varying test costs and fault severities into test case prioritization. In Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Toronto, ON, Canada, 12–19 May 2001; pp. 329–338. [CrossRef]
22. Elbaum, S.; Rothermel, G.; Kanduri, S.; Malishevsky, A.G. Selecting a Cost-Effective Test Case Prioritization Technique. *Softw. Qual. J.* **2004**, *12*, 185–210. [CrossRef]
23. Luo, Q.; Moran, K.; Zhang, L.; Poshyvanyk, D. How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects. *IEEE Trans. Softw. Eng.* **2019**, *45*, 1054–1080. [CrossRef]
24. Pradhan, D.; Wang, S.; Ali, S.; Yue, T. Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study. In Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, Denver, CO, USA, 20–24 July 2016; ACM: New York, NY, USA, 2016; pp. 1085–1092. [CrossRef]
25. Arrieta, A.; Wang, S.; Sagardui, G.; Etxeberria, L. Test Case Prioritization of Configurable Cyber-Physical Systems with Weight-Based Search Algorithms. In Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, Denver, CO, USA, 20–24 July 2016; ACM: New York, NY, USA, 2016; pp. 1053–1060. [CrossRef]
26. Aichernig, B.K.; Tappler, M. Efficient Active Automata Learning via Mutation Testing. *J. Autom. Reason.* **2019**, *63*, 1103–1134. [CrossRef]
27. Krenn, W.; Schlick, R.; Tiran, S.; Aichernig, B.; Jobstl, E.; Brandl, H. MoMut:: UML Model-Based Mutation Testing for UML. In Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, 13–17 April 2015; pp. 1–8. [CrossRef]
28. Belli, F.; Budnik, C.J.; Hollmann, A.; Tuglular, T.; Wong, W.E. Model-based mutation testing—Approach and case studies. *Sci. Comput. Program.* **2016**, *120*, 25–48. [CrossRef]
29. Eghbali, S.; Tahvildari, L. Test Case Prioritization Using Lexicographical Ordering. *IEEE Trans. Softw. Eng.* **2016**, *42*, 1178–1195. [CrossRef]
30. XML Metadata Interchange (XMI) Specification. Available onlilne: https://www.omg.org/spec/XMI/2.0/PDF (accessed on 15 May 2003).

31. McMinn, P.; Kapfhammer, G. AVMf: An Open-Source Framework and Implementation of the Alternating Variable Method. In Proceedings of the International Symposium on Search Based Software Engineering: 8th, Raleigh, NC, USA, 8–10 October 2016; pp. 259–266. [CrossRef]
32. Kempka, J.; McMinn, P.; Sudholt, D. Design and analysis of different alternating variable searches for search-based software testing. *Theor. Comput. Sci.* **2015**, *605*, 1–20. [CrossRef]
33. Catal, C.; Mishra, D. Test case prioritization: A systematic mapping study. *Softw. Qual. J.* **2013**, *21*. [CrossRef]