Article

Test Case Generation Method for Increasing Software Reliability in Safety-Critical Embedded Systems

Bongjoo Koo 1,2, Jungho Bae 1, Seogbong Kim 1, Kangmin Park 1 and Hyungshin Kim 2,*

1 Agency for Defense Development, Daejeon 34186, Korea; tgkoo@add.re.kr (B.K.); jhbae@add.re.kr (J.B.); fieldplant@add.re.kr (S.K.); kmp@add.re.kr (K.P.)
2 Computer Engineering Department, Chungnam National University, Daejeon 34134, Korea
* Correspondence: hyungshin@cnu.ac.kr

Received: 14 April 2020; Accepted: 8 May 2020; Published: 12 May 2020

Abstract: Finite-state machines (FSMs) and the W method have been widely used in software testing. However, the W method fails to detect post-processing errors in the implementation under test (IUT) because it ends testing when it encounters a previously visited state. To alleviate this issue, we propose an enhanced fault-detection W method. The proposed method does not stop the test, even if it has reached a previously visited state; it continues to test and check the points that the W method misses. Through various case studies, we demonstrated software testing using the W method and the proposed method. From the results, it can be inferred that the proposed method can more explicitly determine the consistency between design and implementation, and it is a better option for testing larger software. Unfortunately, the testing time of the proposed method is approximately 1.4 times longer than that of the W method because of the added paths. However, our method is more appropriate than the W method for software testing in safety-critical systems, even if this method is time consuming. This is because the error-free characteristics of a safety-critical system are more important than anything else. As a result, our method can be used to increase software reliability in safety-critical embedded systems.

Keywords: finite-state machine; W method; post-processing errors; implementation under test; enhanced fault-detection W method; software reliability; safety-critical embedded system

1. Introduction

In recent decades, because of the groundbreaking development of computer engineering technology, automatic control systems have gradually substituted for humans. Initially, this change depended on the capability of hardware systems. However, with the growth of the semiconductor industry, better central processing units (CPUs) for computers were developed, and software became a more important part of such systems.

Accordingly, detecting a malfunction in software before its release has also become an increasingly important part of its development. This is particularly true for safety-critical systems, such as those in airplanes, trains, nuclear power plants, autonomous driving cars, and defense systems. This has led to the increasing need for software testing.

Among numerous testing methods, finite-state machine (FSM) has been widely applied in various fields. D. Lee and M. Yannakakis defined five fundamental problems in using FSM [1]. These are homing/synchronizing sequence, state identification, state verification, machine verification/fault detection/conformance testing, and machine identification. Conformance testing checks the equivalence between the FSM specification and the implementation under test (IUT). In other words, the state diagram of the FSM specification can be compared with the behavior of an implementation to verify whether the IUT is equivalent to the FSM specification in the conformance testing.
This conformance testing is often used to test software. Numerous studies have been conducted in this field. Some studies involved reducing the testing size (length) [2-6], modeling FSM in various ways [7-17], and establishing a test criterion [18].

In this paper, we aim to reinforce the checking of the equivalence between the FSM specification and the IUT. We propose an enhanced fault-detection W method to generate test suites (sequences), analyze the testing time, and apply the method to case studies. Through testing time analysis and case studies, we demonstrate that system checking through the W method is not up to par with our method, even if our method is time consuming. This indicates that the consistency between design and implementation can be verified explicitly by our method, and our method is more appropriate than the W method in safety-critical systems where the error-free characteristic is more important than anything else. Moreover, we show that the difference in the number of errors captured by the two methods increases as the software becomes larger and more complex. This result indicates that our method is a better option for testing larger software. As a result, our method can be used to increase software reliability in safety-critical embedded systems.

2. Background

FSM and W method are widely used to test software. In this section, we briefly discuss the UML (Unified Modeling Language) [19,20], FSM, and W method as background techniques.

2.1. UML State Machine

An FSM can be represented by a state transition diagram. State-based languages, such as Statecharts [21], SDL (Specification and Description Language) [22], and UML are used for state transition diagrams. In this study, the UML state machine is used to represent FSM.

Definition:
The dynamic behavior of a component is represented by a state machine \( SM = (S, s_0, S_F, I, O, \sigma) \)

- \( S \): Non-empty set
- \( s_0 \): Initial state, the element of the set “S”
- \( S_F \): Final state, the element of the set “S”
- \( I \): The set of input event
- \( O \): The set of output event
- \( \sigma: S \times I \rightarrow O \times S' \); transition function between two states

An example of a state machine is shown in Figure 1. This state machine has two states and a transition function. The set of the states is \( S = \{ \text{IDLE}, \text{PWR ON} \} \) and the transition function is \( \sigma = \{(\text{IDLE}, \text{PwrOn}, \text{Waiting} (20)), \text{PWR ON} \} \). The initial state is \( s_0 = \text{IDLE} \). The final state is \( S_F = \text{PWR ON} \). The input is \( I = \{ \text{PwrOn} \} \), and the output is \( O = \{ \text{Waiting} (20) \} \). The input in IDLE state is \( \text{PwrOn} \). If \( \text{PwrOn} \) is entered, the state becomes PWR ON state after 20 s.

![Figure 1. An example of state machine.](image-url)

Figure 1. An example of state machine.

2.2. FSM and W Method

Here, we briefly describe the construction of an FSM from a specification and the generation of a test suite using the W method. For an equipment with the following requirement specification, the corresponding FSM is illustrated in Figure 2.
1. Set the root of “T” with the initial state of the FSM, “IDLE”.
2. Suppose we have already built “T” to a level “k”. The “(k + 1)”th level is built by examining nodes transferable from the “k”th-level nodes. A node at the “k”th level is terminated if it is the same as a non-terminal at some level “j”, j ≤ k. Otherwise, we attach a successor node to the “k”th-level node.

In the process of making a tree in Figure 3, if one state was already visited or a final state was encountered, the state will be a terminal node.

This FSM consists of six inputs, I = {Pwr On, Pwr Off, Start Cnt Dn, Inhibit, Fired, Failed}, and five states, S = {IDLE, PWR ON, COUNT DOWN, LIFTOFF, COOLDOWN}.

From the above FSM, the W method derives a tree “T” via breadth-first search or depth-first search as follows:

1. Set the root of “T” with the initial state of the FSM, “IDLE”. This is level 1 of “T”.
2. Suppose we have already built “T” to a level “k”. The “(k + 1)”th level is built by examining nodes transferable from the “k”th-level nodes. A node at the “k”th level is terminated if it is the same as a non-terminal at some level “j”, j ≤ k. Otherwise, we attach a successor node to the “k”th-level node.

In the process of making a tree in Figure 3, if one state was already visited or a final state was encountered, the state will be a terminal node.

Five test sequences in Table 1 can be generated from the tree.
Table 1. Test suite by W method.

| No. of Test Sequence | Test Sequence | Covered Requirements |
|----------------------|---------------|----------------------|
| 1                    | Idle → Pwr on → Idle | Req. 1, 4          |
| 2                    | Idle → Pwr on → Count down → Pwr on | Req. 2, 5 |
| 3                    | Idle → Pwr on → Count down → Lift off | Req. 2, 3, 5 |
| 4                    | Idle → Pwr on → Cool down | Req. 1, 4       |
| 5                    | Idle → Pwr on → Count down → Cool down | Req. 2, 3, 5 |

3. Proposed Method

In this section, we propose a modified method to supplement the W method. Moreover, we theoretically analyze the testing time of the two methods to determine the feasibility of the proposed method in real testing.

The terminologies used in this paper are defined as follows.

- **Node**: It is used as the corresponding expression of the “state” in the tree, i.e., a “node” is used in the tree as a “state” is used in the FSM.
- **Terminal node**: This is present in W method. If one state has already been visited, the state becomes a “terminal node”. The final state is also a “terminal node”.
- **Initial entry status**: This is the status of the state when it has been entered for the first time.
- **Re-entry status**: This is the status of the state when it is re-entered after entering another state.
- **Post-processing**: This is the processing that makes the re-entry status equal to the initial entry status.
- **Post-processing error(s)**: These error(s) are occurred by the reasons why post-processing is incorrect.
- **Test case**: This has the same meaning as “test sequence”.
- **Test set**: This is the collection of test cases (sequences). It has the same meaning as “test suite”.

3.1. Algorithm

The W method ceases to make a test sequence when it encounters a previously visited state. This mechanism fails to detect post-processing errors in the IUT. To illustrate, given a state A, let $A_i$ be the initial entry status of A and $A_r$ be the re-entry status of A. When a software is implemented, there are many cases in which $A_i$ is not equal to $A_r$. This is because of the mistakes in post-processing such as memory allocation faults and faults for initializing variables. The mechanism of the W method does not test this point, which means that it fails to check the equivalence between an FSM specification and an IUT.

Based on the foregoing, we propose a modified method to improve the W method. The proposed method adds test paths from a terminal node to a specified final state. The added paths check whether the behavior of the software is the same in the initial entry and the re-entry, i.e., the deficiency of the W method. The algorithm of the proposed method is presented below.
Algorithm of the proposed method

1: input:
2: \( SM: \text{state machine}, s': \text{a specified final state} \)
3: output:
4: \( TS: \text{Test set} \)
5: begin
6: \( TS = \{\}, \text{Tree } Tr = \{\text{init node } n_0, \text{ node set } N = \{n_0\}, \text{ edge set } E = \{\}\} \)
7: \( n_0 \) is a node corresponding to the initial state \( s_0 \in SM \)
8: for each non-terminal leaf node \( n \in Tr \)
9: begin
10: for each outbound transition \( t \) of the state corresponding to \( n \)
11: begin
12: \( st = \text{a target state of } t \)
13: for each input \( i \in I \in t \)
14: begin
15: \( n' = \text{a node corresponding to } st, N = N \cup \{n'\} \)
16: \( e = \text{an edge } [n \times \{i\} \to O \in t \times n'], E = E \cup \{e\} \)
17: end for
18: if \( st \) is already represented by another node then
19: \( Tr = Tr \cup \text{shortest path from the node } n' \text{ corresponding to } st \) to a node corresponding to \( s' \), \( st = s' \)
20: end if
21: if \( st \) is a final state then
22: set \( n' \) is terminal node
23: end if
24: end for
25: end for
26: for each event sequence \( TC \) from \( n_0 \in Tr \) to a terminal node \( n \in Tr \)
27: begin
28: \( TS = TS \cup TC \)
29: end for
///end///

In the above algorithm, lines 6 and 7 initialize the test set and the tree. The tree, \( Tr \), consists of the root(init) node “\( n_0 \)”, node set “\( N' \)” corresponding to the state, and edge set “\( E' \)” corresponding to the transition.

Lines 8 to 25 are the main body of the algorithm. Among these, lines 10 to 24 add child nodes and edges for transitions. Lines 18 to 20 are the novel addition which enhances the algorithm. These lines add the shortest path from the terminal node in the W method to the specified final state. Lines 21 to 23 set the node as the terminal node in the proposed method if the added child node corresponds to a specified final node. Lines 26 to 29 collect all the sequences from the initial node to the terminal node in the proposed method and make a test set (suite).

Using the new algorithm, Figure 3 becomes the tree in Figure 4.
The modified test sequences are shown in Table 2.

| No. of Test Sequence | Test Sequence                                                                 | Covered Requirements |
|----------------------|-----------------------------------------------------------------------------|----------------------|
| 1                    | Idle → Pwr on → Idle → Pwr on → Count down → Lift off                       | Req. 1, 2, 3, 4, 5   |
| 2                    | Idle → Pwr on → Count down → Lift off                                       | Req. 1, 2, 3, 4, 5   |
| 3                    | Idle → Pwr on → Count down → Lift off                                       | Req. 2, 3, 5         |
| 4                    | Idle → Pwr on → Count down → Lift off                                       | Req. 1, 4            |
| 5                    | Idle → Pwr on → Count down → Lift off                                       | Req. 2, 3, 5         |

3.2. Testing Time Analysis

The proposed method returns a longer test suite (sequences) than the W method. To compare the testing time of the two methods, theoretical analysis is performed.

Suppose we have an FSM as in Figure 5.

The tree derived using W method is illustrated in Figure 6.
In Figure 6, each state is denoted as “An” where “n” is varied from 1 to N. The level of the tree is denoted as “l” and is varied from 1 to N + 1.

To determine the testing time, we sum the tested nodes. In W method, the number of tested nodes, $Tw$, is given by the following equation.

$$Tw = 0 \times 1 + 1 \times 2 + 2(N-1) \times 3 + 3(N-1)(N-2) \times 4 + \ldots + (l-1)(N-1)!/(N-(l-1))! \times l$$

$$= \sum_{i=2}^{N+1} i(l-1)(N-1)!/(N-l+1)!$$

(1)

The tree derived using the proposed method is shown in Figure 7.

In the proposed method, testing time is calculated using a different approach.

In case of “An” terminal nodes:

1. The number of test nodes for a “An” in each level, $Tpn(An, l)$, is the sum of the test nodes in W method and the added nodes in the proposed method.
2. If the number of “An”s in each level is $Nt(An, l)$, the number of test nodes for “An”s in each level, $Tpn(An, l)$, is $Tpn(An, l) \times Nt(An, l)$.
3. The number of test nodes for “An”s in the tree, $Tp(An)$, is the sum of $Tp(An, l)$ where $l$ is from 1 to $N + 1$. 

Figure 6. Tree using W method.

Figure 7. Tree using the proposed method.
As a result, the testing time of the proposed method, $T_p$, is the sum of $T_p(A_n)$ and is given by the following equation.

$$T_p = \sum_{n=1}^{N} T_p(A_n)$$  \hspace{1cm} (2)

To obtain $T_p(A_n)$ from Figure 7, we start with “A1” terminal nodes as follows.

1. Level 1: $Tpn(A1, 1) = 1$, $Nt(A1, 1) = 0$, $Tp(A1,1) = 0$
2. Level 2: $Tpn(A1, 2) = N + 1$, $Nt(A1, 2) = 1$, $Tp(A1,2) = N + 1$
3. Level 3: $Tpn(A1, 3) = N + 2$, $Nt(A1, 3) = N - 1$, $Tp(A1,3) = (N + 2) \times (N - 1)$
4. Level 4: $Tpn(A1, 4) = N + 3$, $Nt(A1, 4) = (N - 1)(N - 2)$, $Tp(A1,4) = (N + 3)(N - 2)$

\[ \cdots \cdots \]

5. Level $N + 1$: $Tpn(A1, N + 1) = 2N$, $Nt(A1, N + 1) = (N - 1)!$, $Tp(A1,N + 1) = 2N \times (N - 1)!$

According to the above results,

$$T_p(A1) = \sum_{l=2}^{N+1} (N + l - 1)(N - 1)! / (N - l + 1)!$$  \hspace{1cm} (3)

In the same manner,

$$T_p(A2) = \sum_{l=2}^{N+1} (N + l - 2)(l - 2)(N - 2)! / (N - l + 1)!$$  \hspace{1cm} (4)

$$T_p(AN) = \sum_{l=2}^{N+1} l(l - 2)(N - 2)! / (N - l + 1)!$$  \hspace{1cm} (5)

Therefore,

$$T_p = \sum_{n=1}^{N} T_p(A_n) = \sum_{l=2}^{N+1} \left[ \frac{(N - 1)(2l^2 + (N - 4)l + 2)}{2(N - l + 1)!} \right]$$  \hspace{1cm} (6)

Using Equations (1) and (6), we get the following results in Table 3 and Figure 8.

| No. of States (N) | $T_w$ | $T_p$ | $T_p/T_w$ |
|-------------------|-------|-------|-----------|
| 3                 | 38    | 51    | 1.34      |
| 4                 | 212   | 293   | 1.38      |
| 5                 | 1370  | 1924  | 1.40      |
| 6                 | 10,112| 14,352| 1.42      |
| 7                 | 84,158| 120,365| 1.43     |
| 8                 | 780,908| 1,123,411| 1.44 |
| 9                 | 8,000,882| 11,562,918| 1.44 |

Table 3. Results of $T_w$ and $T_p$.

![Figure 8](image.png)  
*Figure 8. Testing time of two methods and testing time increasing rate of $T_p$ by number of states.*
From the above, we know that the proposed method takes about 1.4 times longer than the W method. Nevertheless, the proposed method remains acceptable considering the enhanced effectiveness of the testing, especially in the safety-critical system.

4. Case Study

Four case studies are performed with the software of the safety-critical system. These are console software of the army missile defense system, a console software of the navy missile defense system, a satellite operation software, and a radar operation software. Among these, only the first case study is described in detail. The other case studies are briefly summarized.

4.1. Methodology

Case studies were performed through the following methodology.

1. Construct an FSM specification from the requirement specification.
2. Derive a tree and generate test suites using the W method and the proposed method.
3. Test the implementation of the two test suites and compare the results. This step is repeated to check the reproducibility of the test.

The basic information of the case studies is shown in Table 4.

| No. of Modules | Army Missile | Navy Missile | Satellite | Radar |
|----------------|--------------|--------------|-----------|-------|
| Lines of Code  | 74,122       | 220,153      | 5207      | 60,389|
| McCabe’s Cyclomatic Complexity | 7881 | 25,701 | 512 | 6358 |

4.2. Constructing FSM

In this section, we show the FSM specification from the requirements specification. The requirements of the army missile console software are as follows.

- Req 1: Mission Assign, Mission assignment, or cancel
- Req 2: Password Control, Input password, and confirm
- Req 3: Power Control, Power on, and off control
- Req 4: Mission Process Control, Mission Processing after Power on
- Req 5: Countdown Control, Countdown start, and lift-off detection
- Req 6: Fire Aborted (Cool down), Emergency cool down sequence execution
- Req 7: Ready Monitoring, Firing Ready status monitoring
- Req 8: Countdown Monitoring, Countdown status monitoring

The constructed FSM is shown in Figure 9.
4.3. Deriving Tree and Generating Test Suite

Two trees were derived from the above FSM. Figure 10 shows the tree using W method.

Figure 9. FSM of the army missile console software.

Figure 10. Tree derived using W method.

Figure 11 shows the tree using the proposed method.
Two test suites were generated from the above trees as shown in Tables 5 and 6.

Table 5. Test suite using W method.

| No. of Test Sequence | Test Sequence |
|----------------------|---------------|
| 1                    | Idle→Assigned→Idle |
| 2                    | Idle→Assigned→Weapon Initializing→Assigned |
| 3                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Ready Complete |
| 4                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Assigned |
| 5                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Assigned |
| 6                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Lift off |
| 7                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Fire Aborted |
| 8                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Fire Aborted |
| 9                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Fire Aborted |
| 10                   | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Aborted |
| 11                   | Idle→Assigned→Weapon Initializing→Fire Aborted |

Figure 11. Tree derived using the propose method.
Table 6. Test suite using the proposed method.

| No. of Test Sequence | Test Sequence                                      |
|----------------------|----------------------------------------------------|
| 1                    | Idle→Assigned→Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Lift off |
| 2                    | Idle→Assigned→Weapon Initializing→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Lift off |
| 3                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Lift off |
| 4                    | Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Lift off |
| 5                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Lift off |
| 6                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Lift off |
| 7                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Irreversible→Fire Aborted |
| 8                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Reversible→Fire Aborted |
| 9                    | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Enable→Fire Aborted |
| 10                   | Idle→Assigned→Weapon Initializing→Fire Ready Complete→Fire Aborted |
| 11                   | Idle→Assigned→Weapon Initializing→Fire Aborted |

Table 7 shows the requirements covered by the two test suites.

Table 7. Covered Requirements.

| No. of Test Sequence | W Method | Proposed Method |
|----------------------|----------|-----------------|
| 1                    | Req. 1   | Req. 1, 2, 3, 4, 5, 6, 7, 8 |
| 2                    | Req. 1, 3, 4 | Req. 1, 2, 3, 4, 5, 6, 7, 8 |
| 3                    | Req. 1, 2, 3, 4, 7 | Req. 1, 2, 3, 4, 5, 6, 7, 8 |
| 4                    | Req. 1, 3, 4, 7 | Req. 1, 2, 3, 4, 5, 6, 7, 8 |
| 5                    | Req. 1, 2, 3, 4, 7 | Req. 1, 2, 3, 4, 5, 6, 7, 8 |
| 6                    | Req. 1, 2, 3, 4, 5, 7, 8 | Req. 1, 2, 3, 4, 5, 7, 8 |
| 7                    | Req. 1, 2, 3, 4, 5, 6, 7, 8 | Req. 1, 2, 3, 4, 5, 6, 7, 8 |
| 8                    | Req. 1, 2, 3, 4, 5, 6, 7, 8 | Req. 1, 2, 3, 4, 5, 6, 7, 8 |
| 9                    | Req. 1, 2, 3, 4, 6, 7 | Req. 1, 2, 3, 4, 6, 7 |
| 10                   | Req. 1, 3, 4, 6, 7 | Req. 1, 3, 4, 6, 7 |
| 11                   | Req. 1, 3, 4, 6 | Req. 1, 3, 4, 6 |

4.4. Test and Results

The configuration of the missile defense system is shown in Figure 12, which was used as the test configuration. The equipment used for testing were the real equipment, except for the launcher (with weapon), which was substituted with a simulator. While the test was processed, we checked the results through the graphic user interface(GUI) screen of the console.
The test results from the two suites are shown in Table 8.

| Test Suite   | Total Length (Test Nodes) | No. of Errors (Founded) |
|--------------|----------------------------|-------------------------|
| W method     | 61                         | 5                       |
| Proposed method | 90                     | 7                       |

The proposed method detected two errors more than that of the W method.

One error was detected in test sequence no. 1 [Idle → Assigned → Idle → Assigned → … ]. The original sequence of W method is [Idle → Assigned → Idle]. The error is that the weapon (missile) was not assigned in the re-entry to the “Assigned” state. In the initial entry, missile assignment was performed successfully, but in the re-entry after cancelling the assignment, missile reassignment failed. This error is a fatal error resulting in the mission not being performed.

The other error was detected in test sequences no. 5 [ … → Assigned→ … →Fire Enable→ Assigned→ … → Fire Enable→ … → Lift off]. The original sequence of W method is [ … → Assigned→ Weapon Initializing→ Fire Ready Complete→ … ]. The error is that the firing authority was permitted without checking for a password. In the initial entry to the “Fire Enable” state, password checking was performed successfully, but in the re-entry, the firing authority was given without checking for a password. This error is also a fatal error in that a missile can be fired by a person who does not have the authority. These errors appeared repeatedly when the same software is used without debugging the errors.

This case study indicates that our proposed method can more properly check the equivalence between the FSM specification and the IUT.

Other case studies were performed in the same manner. The test summary is shown in Table 9, Figures 13 and 14.

| Case Study No. | System    | Total Testing Length (W/Proposed) | No. of Errors (W/Proposed) |
|---------------|-----------|-----------------------------------|----------------------------|
| 1             | Army Missile | 61/90                            | 5/7                        |
| 2             | Navy Missile | 226/308                          | 23/40                      |
| 3             | Satellite      | 18/32                            | 2/2                        |
| 4             | Radar        | 41/77                            | 8/9                        |
Among numerous testing methods, FSM and the W method have been widely used. However, the W method fails to detect post-processing errors in the IUT. To correct this deficiency, we proposed an enhanced fault-detection W method to generate test suites (sequences), analyzed the testing time, and applied the method to case studies. Through testing time analysis and case studies, we demonstrated that system checking through the W method is not up to par with our method, even if our method is time consuming. This indicates that the consistency between design and implementation is a deficiency, and our method becomes a better option for testing as the software becomes longer and more complex.

5. Conclusions

Testing is the most important part of software development, especially in safety-critical systems. Among numerous testing methods, FSM and the W method have been widely used. Furthermore, our method is more appropriate than the W method for software testing in safety-critical systems where the error-free characteristic is more important than anything else.
Moreover, we showed that the difference in the number of errors captured by the two methods increases as the software becomes larger and more complex. This result inferred that our method is a better option for testing larger software. Thus, our method can be used to increase software reliability in safety-critical embedded systems.

In future studies, the proposed method will be applied to more case studies for further modification and generalization.

**Author Contributions:** Conceptualization, B.K. and J.B.; methodology, B.K.; software, S.K. and K.P.; validation, B.K. and J.B.; formal analysis, B.K.; investigation, J.B.; writing—original draft preparation, B.K.; writing—review and editing, H.K. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

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

**References**

1. Lee, D.; Yannakakis, M. Principles and Methods of Testing Finite State Machines—A survey. *Proc. IEEE* 1996, 84, 1090–1123. [CrossRef]
2. Chow, T.S. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.* 1978, 4, 178–187. [CrossRef]
3. Dorofeeva, M.; Koufareva, I. Novel modification of the W-method. *Jt. NCC IIS Bull. Comp. Sci.* 2002, 18, 69–80.
4. Fujiwara, S.; Bochmann, G.; Khendek, F.; Amalou, M.; Ghedamsi, A. Test selection based on finite state models. *IEEE Trans. Softw. Eng.* 1991, 6, 591–603. [CrossRef]
5. Petrenko, A.; Yevtushenko, N.; Bochmann, G.V. Testing determinisitic implementations from their nondeterministic specifications. In Proceedings of the Testing of Communicating Systems: Proceedings IFIP 9th Intern Workshop of Protocol Test Systems, Darmstadt, Germany, 9–11 September 1996.
6. Cutigi, J.F.; Simao, A.; Souza, S.R.S. Reducing FSM-Based Test Suites with Guaranteed Fault Coverage. *Comput. J.* 2016, 5, 1129–1143. [CrossRef]
7. Florentin, I.; Logica, B. W-method for Hierarchical and Communicating Finite State Machines. In Proceedings of the 2007 5th IEEE International Conference on Industrial Informatics, Vienna, Austria, 23–27 June 2007.
8. Liang, H.; Dingel, J.; Diskin, Z. A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. In Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, Shanghai, China, 27 May 2006.
9. Whittle, J.; Schumann, J. Generating Statechart Designs from Scenarios. In Proceedings of the International Conference on Software Engineering, Limerick, Ireland, 4–11 June 2000.
10. Sun, J.; Dong, J.S. Extracting FSMs from Object-Z Specifications with History Invariants. In Proceedings of the International Conference on Engineering of Complex Computer Systems, Shanghai, China, 16–20 June 2005.
11. Dick, J.; Faivre, A. Automating the Generation and Sequencing of Test Cases from Model Based Specifications. In Proceedings of the FME’93: Industrial-Strength Formal Methods, Fifth International Symposium Formal Methods Europe, Odense, Denmark, 19–23 April 1993.
12. Carrington, D.; MacColl, I.; McDonald, J.; Murray, L.; Strooper, P. From Object-Z Specifications to ClassBench Test Suites. *Softw. Test. Verif. Reliab.* 2000, 10, 111–137. [CrossRef]
13. Hierons, R.M. Testing from a Z Specification. *Softw. Test. Verif. Reliab.* 1997, 7, 19–33. [CrossRef]
14. Weißleder, S. Simulated Satisfaction of Coverage Criteria on UML State Machines. In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation, Paris, France, 6–9 April 2010.
15. Friske, M.; Schelingloff, B.H. Improving Test Coverage for UML State Machines using Transition Instrumentation. In Proceedings of the International Conference on Computer Safety, Reliability, and Security, Nuremberg, Germany, 18–21 September 2007; pp. 301–314.
16. Rajan, A.; Whalen, M.W.; Heimdahl, M.P.E. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In Proceedings of the International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008.
17. Aleksandr, S.T.; Maxim, L.G.; Nina, V.Y. Testing systems of Interacting Timed Finite State Machines with the Guaranteed Fault Coverage. In Proceedings of the 17th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM), Erlagol, Russia, 30 June–4 July 2016.

18. Hoda, K. Finite State Machine Testing Complete Round-trip Versus Transition Trees. In Proceedings of the IEEE 28th International Symposium on Software Reliability Engineering Workshops, Toulouse, France, 23–26 October 2017.

19. Booch, G.; Rumbaugh, J.; Jacobson, I. Unified Modelling Language User Guide, 2nd ed.; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2005.

20. UML. Available online: http://www.omg.org/spec/UML (accessed on 18 December 2019).

21. Harel, D.; Politi, M. Modeling Reactive Systems with Statecharts: The STATEMATE Approach; McGraw-Hill: New York, NY, USA, 1998.

22. ITU-T. Recommendation Z.100 Specification and Description Language (SDL); International Telecommunications Union: Geneva, Switzerland, 1999.