Received: 30 September 2020 | Revised: 22 March 2022 | Accepted: 3 April 2022

DOI: 10.1002/stvr.1815

SPECIAL ISSUE PAPER

WILEY

MuFBDTester: A mutation-based test sequence generator for FBD programs implementing nuclear power plant software

Lingjun Liu | **Eunkyoung Jee** | **Doo-Hwan Bae**

School of Computing, KAIST, Daejeon, Republic of Korea

Correspondence
Eunkyoung Jee, School of Computing, KAIST, Daejeon, Republic of Korea.
Email: ekjee@se.kaist.ac.kr

Funding information
Ministry of Education, Grant/Award Number: NRF-2019R1I1A1A01062946; Nuclear Safety and Security Commission (NSSC) of the Republic of Korea, Grant/Award Number: 2105030; Institute of Information & Communications Technology Planning & Evaluation, Grant/Award Number: 2015-0-00250; MSIT (Ministry of Science and ICT), Korea, Grant/Award Number: IITP-2022-2020-0-01795

Summary
Function block diagram (FBD) is a standard programming language for programmable logic controllers (PLCs). PLCs have been widely used to develop safety-critical systems such as nuclear reactor protection systems. It is crucial to test FBD programs for such systems effectively. This paper presents an automated test sequence generation approach using mutation testing techniques for FBD programs and the developed tool, MuFBDTester. Given an FBD program, MuFBDTester analyses the program and generates mutated programs based on mutation operators. MuFBDTester translates the given program and mutants into the input language of a satisfiability modulo theories (SMT) solver to derive a set of test sequences. The primary objective is to find the test data that can distinguish between the results of the given program and mutants. We conducted experiments with several examples including real industrial cases to evaluate the effectiveness and efficiency of our approach. With the control of test size, the results indicated that the mutation-based test suites were statistically more effective at revealing artificial faults than structural coverage-based test suites. Furthermore, the mutation-based test suites detected more reproduced faults, found in industrial programs, than structural coverage-based test suites. Compared to structural coverage-based test generation time, the time required by MuFBDTester to generate one test sequence from industrial programs is approximately 1.3 times longer; however, it is considered to be worth paying the price for high effectiveness. Using MuFBDTester, the manual effort of creating test suites was significantly reduced from days to minutes due to automated test generation. MuFBDTester can provide highly effective test suites for FBD engineers.

KEYWORDS
automated test generation, function block diagram, mutation adequacy, mutation testing, SMT solver

1 | INTRODUCTION

Function block diagram (FBD) is one of the programming languages for programmable logic controllers (PLCs) defined in the IEC 61131-3 standard [1]. As FBD programs are used to implement safety-critical systems such as reactor protection systems, their testing is very important. For nuclear power plant (NPP) software, the test coverage should be specified, and comprehensive testing should be provided. The coverage of requirements and module structure are both important for the unit testing of the NPP software [2]. However, there are no specific structural coverage requirements in the regulatory guide and safety standard for NPP software testing [2,3]. Thus, it is essential to provide strong testing criteria and automated testing tools for the NPP software.

Research has been ongoing to automatically generate test sets for FBD programs. Most studies are focused on generating tests based on the structural coverage of the FBD program's test models [4-8]. By converting an FBD

program into a binary decision diagram [4], a timed automata [5,6], or using the FBD program itself as a test model [7,8], a test set that meets structural coverage is generated. However, Inozemtseva and Holmes [9] showed that generating test data to satisfy structural coverage has the limitation that achieving high structural coverage is not strongly correlated to high fault detection effectiveness. Thus, test data with a high structural coverage level are not guaranteed to detect certain types of errors. In addition, Shin et al. [10] experimentally showed that the FBD structural coverage criteria are relatively weak at detecting some frequently occurring types of faults in FBD programs, such as wrong constant values and incorrect relational operators.

Mutation testing is an error-based testing technique that provides a testing criterion that can be used to generate mutation-adequate test suites. The mutants are generated by applying mutation operators that denote the types of changes or errors that mimic programming errors. Test suites are crafted to distinguish between the mutants and the program. Thus, mutation-adequate test suites are highly effective at revealing specific types of faults that are expressed by mutation operators. Furthermore, empirical studies on other procedural languages have shown that mutation-based tests provide higher fault revelation than tests guided by structural coverage criteria [11-15]. The FBD programs have different characteristics, such as cyclic execution and data flow representation, from procedural languages; thus, the fault types that frequently occur in FBD programs are also different from those that occur in procedural languages.

In this paper, we proposed a test data generation method that satisfies mutation adequacy for the FBD programs to guarantee the detection of various types of faults in the NPP software. Given a target FBD program and the types of mutation operators, mutants are generated by inserting a fault into the target program. Test data are generated for the purpose of killing these mutants. The target program and the mutants are input to a satisfiability modulo theories (SMT) solver, and the SMT solver determines whether there are input data that can distinguish between the output values of the mutant and those of the target program.

To evaluate our approach, we statistically compared the proposed and structural coverage-based approaches. We created artificial faults using the mutation operator set, different from that utilized for test generation. With a fixed test set size, mutation-based test sets were significantly more effective at fault detection than structural coverage-based test sets. When the test size increased from 10 to 50 test sequences, the fault detection of mutation-based test suites increased from 91% to 97%. In addition, we replicated the experiment from [8] to investigate the mutation and structural coverages of mutation- and structural coverage-based tests. The test sets generated using our approach achieved 100% mutation coverage and could detect the mutants that structural coverage-based test sets failed to identify. On average, the mutation-based test suites achieved an 81% level of strong structural coverage. For all subjects, the mutation-adequate test suites could kill at least 99.9% of the generated higher-order mutants while excluding equivalent mutants. Our approach was also able to identify all the real faults collected from industrial programs. For the complex subject program, mutation-based test suite generation was over 100 times faster than test suite generation based on a strong structural coverage criterion in terms of execution time. In addition, our approach reduced at least 40% of the generation time and generated more effective mutation-adequate test suites compared to the mutation-based approach using model checking [16].

The remainder of this paper is organized as follows. Section 2 explains the background knowledge, and Section 3 describes related studies. In Section 4, we describe the mutation-adequate test sequence generation. In Section 5, we apply our approach to the subject programs and describe experiments intended to answer the research questions we have proposed. We conclude this paper in Section 6.

2 | BACKGROUND

2.1 | FBD programs

The main characteristic of PLCs is indefinite cyclic execution [17]. FBD is a graphical language for PLCs and represents data flows through blocks. An FBD program consists of functions and/or function blocks. The difference between functions and function blocks is related to the internal memory of the PLC. Functions deliver the same output values when the same input parameters are given as functions only consider the input values in the current cycle. Function blocks can yield different output values even when the same input parameters are given. This is because they consider not only the input parameters in the current cycle but also the values stored in the internal memory.

As defined in IEC 61131-3 [1], functions can be categorized into 11 groups: data type conversion, numerical, arithmetic, bit shift, bitwise Boolean, selection, comparison, character string, date and duration, endianness conversion and validate. Function blocks can be categorized into four groups: bistable, edge detection, counter and timer. There are 21 different data types, including Boolean, integer, real numbers and duration.

Figure 1 shows an example of an FBD program called simTRIP, which is composed of a Greater than or Equal to (GE) function, an AND function and one ON-delay-Timer (TON) function block. In the TON function block, the

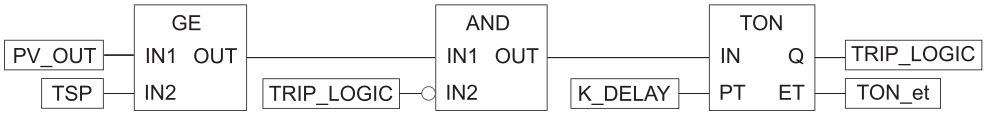

FIGURE 1 Example FBD program called simTRIP

TABLE 1 An example of mutation

Original program *p*	Mutant program *m*
…	…
if (a > 0) a = a + b	if (a > 0) a = a * b
…	…

output Q is assigned true if the input IN has been true for the preset time (PT). The output ET captures the elapsed time that the input IN has been true. In Figure 1, TRIP_LOGIC is assigned true if the output of the AND block has been true for a certain time, which is K_DELAY. TON_et denotes the duration that the output of the AND block has been true. simTRIP is a simplified trip module implemented in a reactor protection system. The output TRIP_LOGIC is a trip signal that would shut down a nuclear reactor safely when TRIP_LOGIC is set to true. TRIP_LOGIC is also an input that represents whether simTRIP sent the shutdown signal in the previous cycle (the output TRIP_LOGIC was set to true) or not. As the FBD program is executed cyclically, TRIP_LOGIC here forms a feedback path. The Processing Value (PV_OUT) represents pressure, temperature and so forth. If the PV_out has been greater than or equal to Trip Set Point (TSP) and simTRIP has not sent a shutdown signal for K_DELAY time, simTRIP sends a trip signal to shut down the nuclear reactor for protection.

2.2 | Mutation testing

Mutation testing is a fault-based testing technique. It is based on the assumption that a competent programmer develops almost correct programs [18]. Therefore, the faults used in mutation testing represent the simple mistakes that programmers often make. It is assumed that these simple faults can be corrected by a few syntactical changes. Moreover, the coupling effect hypothesis states that the test set, which can detect simple faults, is so sensitive that it can also detect a high percentage of more complex errors [19].

The mutation testing method is described as follows. A program p needs to be tested. A set of faulty programs, also called mutants M, is generated by seeding some syntactical changes in the original program p. For example, in Table 1, the mutant $m \in M$ is generated by replacing the addition operator (+) of the original program p with the multiplication operator (*). Then, the original program and mutants are executed on a test set T. If the result of a mutant m differs from the result of the program p on any test case in T, the mutant m is said to be killed or dead. Otherwise, the mutant m is said to be live. After executing all test cases in T, there might be some live mutants. One possible reason is that the test set T is not effective enough to detect the faults seeded in live mutants. The tester needs to provide additional test cases to improve the test set quality. The other reason is that the live mutant is an equivalent mutant. An equivalent mutant is syntactically different from the original program; however, it is functionally the same as the original program (it produces the same result as the original program every time). The goal of mutation testing is to find test cases that kill all non-equivalent mutants [12]. If the test set T killed all non-equivalent mutants, T is clearly adequate with regard to the set of mutants [20]. Mutation testing can be used to assess the quality of test sets, and then it is called a mutation analysis. Mutation analysis involves a testing criterion called the 'mutation score', which is calculated as the number of killed mutants divided by the total number of non-equivalent mutants.

2.3 | Mutation operator set for FBD programs

The mutation operator denotes the rule as to how to change or mutate the program. It can also be considered as the type of changes or faults seeded in the program. For FBD testing, Shin et al. [10] conducted a mutation analysis to evaluate the effectiveness of test suites that met three structural coverage criteria. They defined five mutation operators: constant value replacement (CVR), inverter insertion or deletion (IID), arithmetic block replacement (ABR), logic block replacement (LBR) and comparison block replacement (CBR). Enoiu et al. [16] utilized a mutation testing

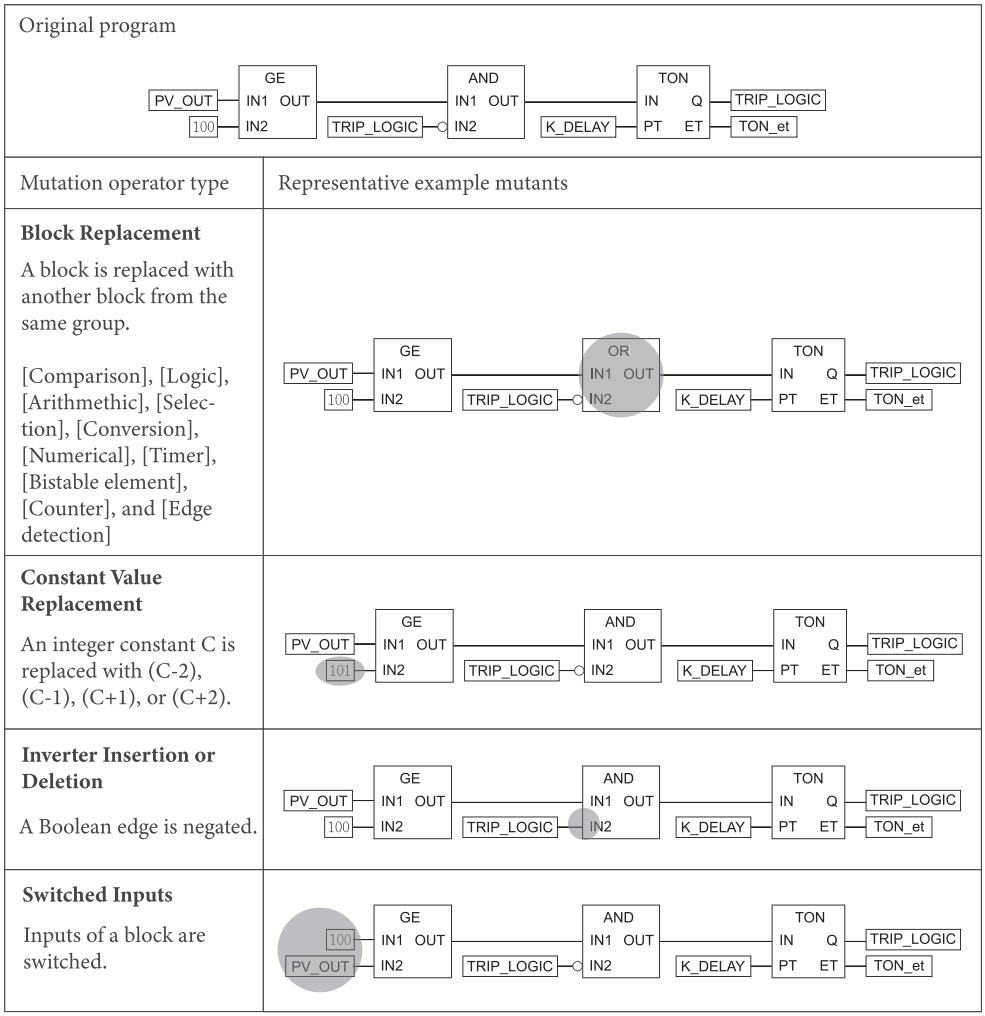

FIGURE 2 Definition of mutation operator set [21]

method in their study and defined six mutation operators: LBR operator, CBR operator, ABR operator, Negation Insertion operator, Value Replacement operator and Timer Block Replacement operator (TRO). The difference between these two mutation operator sets is the additional TRO defined in Enoiu et al.'s work.

Subsequently, Jee et al. [21] extended the work of Shin et al. by adding eight more mutation operators. Figure 2 shows the definition of the mutation operator set of Jee et al. [21] and representative examples. The extended mutation operator set can be categorized into four types: 10 Block Replacement operators, CVR operator, IID operator and Switched Inputs (SWI) operator. The block replacement operators are as follows: CBR, LBR, ABR, Selection Block Replacement, Conversion Block Replacement, Numerical Block Replacement, Timer Block Replacement, Bistable element Block Replacement, Counter Block Replacement and Edge-detection Block Replacement. This mutation operator set covers all the functions and function blocks in our subject programs, so it is practical for our use.

3 | RELATED WORK

3.1 | Test generation for FBD programs

There are several tools for performing unit testing of FBD programs. A simulation-based validation tool named SIVAT [22] generates ANSI C code from an FBD program and performs functional testing based on the code. An integrated tool environment called PLCTOOLS [23] supports the entire development process of PLC programs, including specification, transformation and simulation. PLCTOOLS transforms FBD programs into high-level timed Petri net (HLTPN) models and performs simulation-based functional testing based on the HLTPN models. Lee et al. [24]

proposed a simulation-based testing environment for software programmed with the FBD/LD language. Their testbed was developed by emulating the microprocessor architecture of the PLC used in NPP applications and capturing its behaviour at each machine instruction. The software test cases were manually developed considering the digital characteristics of the target system and the plant dynamics. However, these approaches do not support automated test data generation for FBD programs.

Test data generation for programs implemented in procedural languages is a mature research topic [25-29], and the use of symbolic execution in test data generation has been extensively studied [30-37]. FBD, a data flow language, has different characteristics from those of procedural languages. The types of errors that can occur in FBD programs are also different from those that can occur in programs written in procedural language. For example, FBD programs do not have the concept of branching, which often plays an important role in the testing of procedural language programs. Some core techniques for generating test data for programs written in procedural languages can be similarly used, but the techniques and tools developed for procedural languages cannot be directly applied to FBD programs.

As model-driven development is popular in the industrial automation domain [38,39], several model-based test case generation approaches for PLC programs have been proposed [4-6,40-45]. Enoiu et al. [6,40] proposed a model-based test suite generation for FBD programs. They transformed FBD programs into timed automata models and automatically generated test suites from these timed automata models using the UPPAAL model checker. Test suites can be generated according to structural coverage criteria such as decision coverage, condition coverage and MC/DC coverage on a timed automata model. Their test specifications should be provided as a closed network of timed automata, and a set of test requirements should be manually formulated using CTL formulas. Lahtinen [4] utilized binary decision diagrams to generate test sequences for FBD programs. Wu and Fan [5] generated test cases from timed automata models, which were automatically transformed from FBD programs, using the UPPAAL model checker. They utilized a structural coverage criterion called FPCC in their test generation.

Da Silva et al. [41] generated timed automata models from FBD programs to test whether the execution traces of the implementation conform to the traces of the specification model by using the UPPAAL TRON tool. Kormann and Vogel-Heuser [42] proposed an automated test case generation approach for PLC control software exception handling using fault injection. They injected faults into Unified Modelling Language (UML) state charts and generated test cases that could detect exception handling errors of the target PLC programs. Hametner et al. [43] presented an automated test case generation approach for industrial automation applications described by UML diagrams. They could generate IEC 61499 basic function blocks including the generated test suite; test cases are generated only when test specifications are manually described. Magnus et al. [44] introduced a model-based procedure of automatic test generation for fieldbus profile specification. Their tool transformed a UML state machine, capturing dynamic specifications of function blocks, into a special Petri net dialect and generated test cases with the round-trip path coverage criterion. Hussain and Frey [45] addressed an IEC 61499 compliant UML-based development process with automatic test case generation. UML state diagrams and activity diagrams were used to generate test cases. The round-trip path coverage criterion was used for the state diagrams.

Unlike other works [4-6,40-45], Jee et al. [7] and Song et al. [8] developed test generation techniques that work directly on FBD programs without relying on intermediate models. FBDTester (from Jee et al. [7]) and FBDTester 2.0 (from Song et al. [8]) use FBD-specific structural coverage criteria—specifically, Basic Coverage (BC), Input Condition Coverage (ICC) and Complex Condition Coverage (CCC)—to automatically generate test suites. Our approach is partially motivated by FBDTester 2.0, including the use of an SMT solver and the unwinding loop bound. However, whereas FBDTester 2.0 primarily focuses on structural coverage criteria, our work aims to achieve mutation coverage. In addition, the test generation strategy used by FBDTester 2.0 is to compress the test set size by taking more search resources. We prefer to generate more test data to guarantee test effectiveness.

Research on model-based test generation for other data flow language programs such as Lustre and Simulink has also been conducted [46-49]. Raymond et al. [46] proposed a method to generate test sequences from an observer model written in the Lustre language. Seljimi and Parissis [47] proposed a test sequence generation tool for Lustre programs called Lutess. It is a black box testing tool that uses constraint logic programming and generates input sequences from environmental descriptions. Mohalik et al. [48] generated test cases for Simulink/Stateflow models using model checking with respect to structural coverage criteria such as decision, condition and MC/DC. Tekaya et al. [49] presented an aspect-oriented approach that generates test cases for Simulink models according to blocks, decision coverage and masking MC/DC structural coverage.

All the above-mentioned model-based techniques generate test data from models for specific goals such as coverage achievement. Generating test data to achieve structural coverage, however, has the limitation that it cannot provide specific guarantees for fault detection effectiveness [9]. In contrast, mutation-based test generation can be used to create tests that can detect errors of the type reproduced by the mutation operators. In this study, we propose a technique that ensures fault detection effectiveness for FBD programs by generating test sets that can detect the possible errors expressed by the mutation operators.

3.2 | Mutation-based test generation

Mutation-based test generation techniques for procedural languages have been actively studied [50-53], and they have inspired research on mutation-based test generation for FBD programs. However, the characteristics of the errors that occur in a program written in a data flow language and those that occur in a program written in a procedural language are not the same; thus, it is necessary to define and apply appropriate mutation operators for the data flow language.

The work most relevant to our approach is mutation-based test generation for FBD programs using the UPPAAL model checker [16]. Enoiu et al. [16] generated mutants for the given FBD program by using six mutation operators and used the UPPAAL model checker to generate test suites satisfying the detection of mutants, which is based on a combined timed automata model that contains all the mutants and the original program. In their evaluation, they used manually seeded faults provided by industrial engineers. Their evaluation results showed that even if mutation-based test generation achieved better fault detection than automated decision coverage-based test generation, the mutation-based test suites were not better at detecting faults than manual test suites. Enoiu et al.'s work is valuable as it was the first study to propose the generation of a mutation-based test set for the FBD programs. However, our approach has several advantages over their work. Their approach required additional effort to create and annotate timed automata models from the FBD program, while our approach does not require additional model generation beyond the FBD program. Their approach cannot detect equivalent mutants beforehand because the model checkers decide whether there is no solution only after every state has been explored. We used SMT solvers, which can efficiently determine if the problem is unsatisfiable; thus, our approach does not waste much effort on equivalent mutants. Compared to their work, our developed tool can support more diverse standard block groups and fault types.

Brillout et al. [54] applied the bounded model checking engine to generate test suites for Simulink models with high mutation coverage. Several heuristics were discussed to achieve the desired coverage as well as to keep the number of redundant test cases small to reduce the time required to execute the test suite. Their study is meaningful in that it presents a mutation-based test methodology for the Simulink model. It is similar to ours in that it uses the SAT solver to generate mutation-based tests for a data flow language. However, in their study, there is no consideration of the mutation operators considering the characteristics of the Simulink model, and experimental evaluation of the effectiveness of the methodology they developed is not presented. In contrast, we carefully considered the mutation operator set suitable for the FBD program when generating the mutation-based test for the FBD program, and we experimentally demonstrated the effectiveness of our proposed method through case studies on various target models. He et al. [55] pointed out the computational cost of mutation-based test case generation for Simulink models. To overcome the high computational cost, they proposed an algorithm that combines white-box testing with formal concept analysis. The algorithm measures the similarity among mutants and generates a small-sized test suite with a high coverage. When applying the MuFBDTester proposed in this paper to target systems in our experiments, the optimization issues were not considered because the computational cost was not deemed a major problem. It is worth considering whether the heuristics proposed in Brillout et al.'s study and the technique used in He et al.'s study for the Simulink models can be applied to FBD programs to further improve the scalability of our technique.

Kushigian et al. [56] focused on mutant equivalence detection using SMT solvers for Java programs. Because many procedural language features cannot be defined as first-order logic, they cannot be modelled with SMT solvers. To overcome the applicability challenges, Kushigian et al. translated JVM bytecode instead of source code into SMT constraints. They transformed the JVM bytecode into a control flow graph (CFG) and then rendered the SMT constraints from the CFG structure. In this study, we translate data flow language into SMT constraints. Furthermore, our approach is tailored to FBD programs, and our objective is to generate mutation-adequate tests.

4 | MUTATION-ADEQUATE TEST SEQUENCE GENERATION FOR FBD PROGRAMS

In this section, we describe our automated test sequence generation approach for FBD programs based on mutation testing. Our approach utilizes an SMT solver to find test data that can distinguish between the output results of the target program and those of the mutants. We implemented a tool called MuFBDTester based on this approach with the help of the Yices 1 SMT solver [57]. Figure 3 gives an overview of MuFBDTester, which comprises four main steps: mutant generation, FBD program analysis, Yices input generation and test suite generation. The formal algorithm for Figure 3 is depicted in Algorithm 1. Users are allowed to select variables and assign constant values to them, if necessary. Users also need to select the desired or needed mutation operators. First, MuFBDTester generates mutated programs or mutants based on the users' selection. It then starts analysing the target program and uses the extracted information to subsequently generate Yices input files. For each mutant, the structural information is extracted through analysis, and MuFBDTester constructs a structured Yices input file by combining the information of the target program and the mutant. The Yices input files are executed in succession to generate the test suite. In brief, MuFBDTester

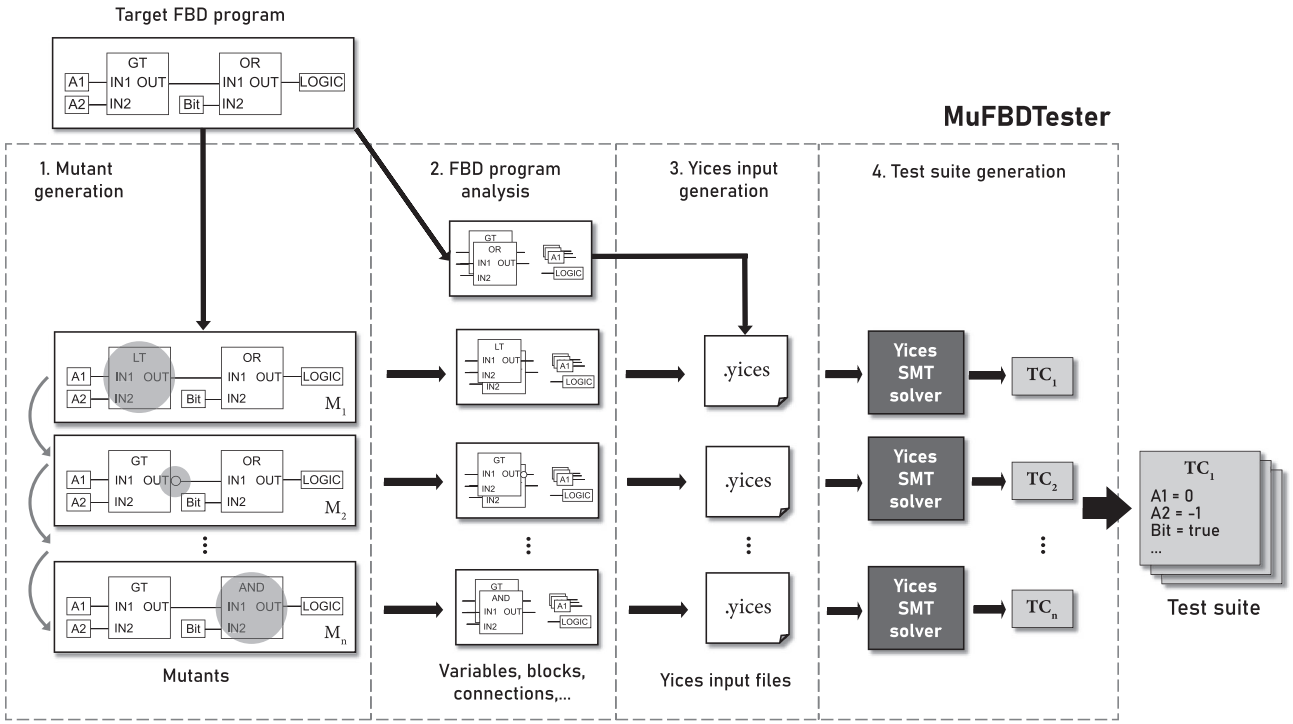

FIGURE 3 Overview of *MuFBDTester*

generates a Yices input file for each mutant to obtain the test sequence to kill it. MuFBDTester is available on GitHub.[1]

Algorithm 1: GenerateTestSequenceSet

Input : an FBD program *p*, a selected mutation operator set *muOpSet*, a library of block operations and information *blockLib*

Output: a set of test sequences *TestSequenceSet*

Local : structure information of program *p pStruc*, a set of generated mutants *mSet*, structure information of a mutant *mStruc*, a yices input file *yicesIN*, a test sequence *TS*

```
1 mSet ← GenerateMutantSet(p, muOpSet);
2 pStruc ← AnalyzeFBD(p);
3 for each m ∈ mSet do
4 |   mStruc ← AnalyzeFBD(m);
5 |   yicesIN ← GenerateYicesInput(pStruc, mStruc, blockLib);
6 |   TS ← ExecuteYices(yicesIN);
7 |   TestSequenceSet ← TestSequenceSet ∪ TS;
8 end for
9 return TestSequenceSet;
```

4.1 | Mutant generation

Target FBD programs in PLCOpen XML format [58] and selected mutation operators are used as input. Mutation testing is effective in detecting faults, but it often involves expensive computing and human resources because of the large number of mutants [59]. Therefore, MuFBDTester allows the users to choose the mutation operators that they need to generate mutants. In addition to mutation operator selection, MuFBDTester also enables users to set the percentages

[1] https://github.com/Rienshaliu/MuFBDTester.

of generated mutants to be used to generate test data. Mutation operator selection and percentage setting are the additional functions for our tool.[2]

Algorithm 2 shows the mutant set generation procedure for an FBD program. The mutant generator parses the target program XML file and then extracts the information regarding all the blocks and constants in the program. Here, constants are directly assigned in FBD programs, for example, 100 in the original program of Figure 2. For each block, the mutant generator applies the corresponding block replacement operator with the number of inputs depending on whether the block replacement operator is selected; the mutant generator applies the SWI operator with equivalent mutant issues depending on whether the SWI operator is selected; the mutant generator applies the IID operator if the IID operator is selected. For each constant, the mutant generator applies the CVR operator when the operator is selected. The following two paragraphs explain the equivalent mutant issues of the SWI operator and why we need to consider the number of inputs while applying the block replacement operators. After mutant generation, MuFBDTester randomly selects candidate mutants based on the user-defined percentage. For instance, if the user sets 50% in the CBR operator, 50% of mutants are selected as candidate mutants from all the mutants generated by the CBR operator. Only those selected candidate mutants will be passed to the next step and used to generate test data.

Algorithm 2: GenerateMutantSet

Input : an FBD program *p*, a selected mutation operator set *muOpSet*
Output: a set of generated mutants *mSet*

```
1  commutativeFunc ← {"ADD", "MUL", "AND", "OR", "EQ", "NE"};
2  blockEleList, inputVarList ← ParseXML(p);
3  for each block ∈ blockEleList do
4      blockGroup ← DetermineBlockGroup(block);
5      if blockGroup in muOpSet then
6          // Apply the corresponding Block Replacement operator
7          brSet ← BR(block, replacibleBlockList[ blockGroup] );
8          mSet ← mSet ∪ brSet;
9      end if
10     if "SWI" in muOpSet then
11         if block is not in commutativeFunc then
12             // Apply Switched Inputs operator
13             swiSet ← SWI(block);
14             mSet ← mSet ∪ swiSet;
15         end if
16     end if
17     if "IID" in muOpSet then
18         // Apply Inverter Insertion or Deletion operator
19         iidSet ← IID(block);
20         mSet ← mSet ∪ iidSet;
21     end if
22 end for
23 /* Constant Value Replacement operator                                    */
24 if "CVR" in muOpSet then
25     for each inVar ∈ inputVarList do
26         if inVar.constant is true then
27             numbers ← {inVar.value − 1, inVar.value − 2, inVar.value + 1, inVar.value + 2};
28             cvrSet ← CVR(inVar, numbers);
29             mSet ← mSet ∪ cvrSet;
30         end if
31     end for
32 end if
33 return mSet;
```

[2]These additional functions are not used for experiments; therefore, all mutation operators are used to generate all possible mutants in our experiments.

Equivalent mutants are functionally equivalent to the given program; thus, obtaining the test data for equivalent mutants is not possible. Thus, the chance of generating equivalent mutants should be limited. We found that switching inputs in commutative functions can generate equivalent mutants. For instance, if we apply the SWI operator to the AND block, there is no impact on the logic (behaviour). The order of input connections has no effect on the AND block operation. When we apply the SWI operator on blocks, we carefully exclude all commutative functions that yield equivalent mutants, including the addition (ADD), multiplication (MUL), AND, OR, equal to (EQ) and not equal to (NE) functions.

In IEC61131-3 [1], there are some extensible functions in which the number of inputs can be increased. In non-extensible functions, the number of inputs is fixed and cannot be extended. In the same block group, there might be some extensible blocks and some non-extensible blocks. For instance, in the arithmetic block group, the addition (ADD) block is an extensible block while the subtraction (SUB) block is a non-extensible block. When applying the ABR operator to an ADD block, if the number of inputs is three, for example, replacement of the ADD function by the SUB function is restricted. Thus, when we apply the block replacement mutation operators, we consider whether the block is extensible or not. In addition, there can be different numbers of inputs or outputs between blocks in the same block group. For instance, the MOVE function only has one input, but the other functions in the arithmetic group take at least two inputs. Even though the MOVE function is in the arithmetic group, we do not consider the MOVE function when applying the ABR operator.

4.2 | FBD program analysis

The FBD program analyser extracts the whole structure of the input FBD program, including variables, blocks and connection elements. For instance, in the simTRIP example program, shown in Figure 1, the variables are PV_OUT, TSP, TRIP_LOGIC, K_DELAY and TON_et; the blocks are GE function, AND function and TON function block; the connection elements are the lines that connect between a variable and a block (e.g., the connection between PV_OUT and the GE function), or between two blocks (e.g., the connection between the GE function and AND function).

In addition to input variables and output variables, the FBD program analyser is also required to identify inout variables and constants. The inout variables are inputs to and outputs from the program, for example, the variable TRIP_LOGIC in simTRIP. Because of cyclic execution, the output values of inout variables are stored in the PLC's internal memory and are delivered as the input of the program in the next cycle.

If a variable is defined as a variable but has a fixed constant value throughout the program execution, MuFBDTester allows the user to select those variables and assign constant values. Variables with constant values are not considered when generating test data.

An FBD program runs periodically with a scan time (e.g., 50 ms). When an FBD program includes function blocks, a test sequence, which is a series of test inputs, is required to handle various internal states of the program. Periodic execution of an FBD program can be viewed as a program with an external loop. When an FBD program contains function blocks with internal states, we explicitly unwind the FBD program for the necessary number of cycles to generate test sequences over several cycles. The number of times the subject program has to be unwound is determined in the FBD program analysis. Song et al. [8] suggested a minimum iteration number, which is the number of cycles by which the FBD program is unwound. We utilize the concept of the minimum iteration number in our approach.

The minimum iteration number of the timer group is specified as $(Presettime/Scantime)+1$; the minimum iteration number of the bistable group and edge-detection group is set to two; the minimum iteration number of the counter group is set to the counter limit (PV) for the counter. The minimum iteration number of an FBD program is the maximum of all the function blocks' minimum iteration numbers. While analysing FBD programs, MuFBDTester calculates the minimum iteration number of the program and determines the iteration number for which the program needs to be unwound in the Yices file. We set the iteration number to the minimum iteration number.

4.3 | Yices input generation

Listing 1 depicts an example Yices input file generated from simTRIP and the mutant produced by applying the LBR operator. The Yices file is composed of the header (Lines 1–3), the definition of the target FBD program (Lines 4–33) and the mutant (Lines 35–54), the test requirement (Line 56) and the command (Line 59). The main idea is to use an SMT solver to search for test data so that the results of mutants are different from the results of the target program. The following subsections describe how we translate FBD programs into the Yices input language, distinguish between the target program and mutants and design the test requirement and command.

```
;; Environment setting
(set-evidence! true)
(set-verbosity! 1)
;; Rule 1-1. Define constant and variables
;; constant variables
(define SCAN_TIME::int 50)
(define K_DELAY::int 100)

(define TON_et_t3::int 0)
;; 2 cycles ago
(define TSP_t2::int )
(define TRIP_LOGIC_t2::bool )
(define PV_OUT_t2::int )
(define GE_out_t2::bool (>= PV_OUT_t2 TSP_t2))
(define AND_out_t2::bool (and GE_out_t2 (not TRIP_LOGIC_t2)))
(define TON_et_t2::int (if AND_out_t2 (if (< TON_et_t3 K_DELAY) (+ TON_et_t3
    SCAN_TIME) K_DELAY) 0))
(define TRIP_LOGIC_out_t2::bool (and AND_out_t2 (>= TON_et_t3 K_DELAY)))
;; 1 cycles ago
```

```
;; present cycle
(define TSP::int )
(define TRIP_LOGIC::bool TRIP_LOGIC_out_t1)
(define PV_OUT::int )
(define GE_out::bool (>= PV_OUT TSP))
(define AND_out::bool (and GE_out (not TRIP_LOGIC)))
(define TON_et::int (if AND_out (if (< TON_et_t1 K_DELAY) (+ TON_et_t1
    SCAN_TIME) K_DELAY) 0))
(define TRIP_LOGIC_out::bool (and AND_out (>= TON_et_t1 K_DELAY)))

;;Mutant
;; Rule 1-1. Define constant and variables

(define M_TON_et_t3::int TON_et_t3)
;; 2 cycles ago
(define M_GE_out_t2::bool (>= PV_OUT_t2 TSP_t2))
(define M_OR_out_t2::bool (or M_GE_out_t2 (not TRIP_LOGIC_t2)))
(define M_TON_et_t2::int (if M_OR_out_t2 (if (< M_TON_et_t3 K_DELAY) (+
    M_TON_et_t3 SCAN_TIME) K_DELAY) 0))
(define M_TRIP_LOGIC_out_t2::bool (and M_OR_out_t2 (>= M_TON_et_t3 K_DELAY)))
;; 1 cycles ago
```

```
;; present cycle
(define M_TRIP_LOGIC::bool M_TRIP_LOGIC_out_t1)
(define M_GE_out::bool (>= PV_OUT TSP))
(define M_OR_out::bool (or M_GE_out (not M_TRIP_LOGIC)))
(define M_TON_et::int (if M_OR_out (if (< M_TON_et_t1 K_DELAY) (+ M_TON_et_t1
    SCAN_TIME) K_DELAY) 0))
(define M_TRIP_LOGIC_out::bool (and M_OR_out (>= M_TON_et_t1 K_DELAY)))

(assert (or (/= TRIP_LOGIC_out M_TRIP_LOGIC_out) (/= TON_et M_TON_et) ))

;; Rule 3. Execute check
(check)
```

LISTING 1 Example Yices file generated from simTRIP

4.3.1 | Translation of FBD programs

In the FBD program analysis, we capture the structural information of FBD programs, which includes variables, blocks and connections. With these essential elements in FBD programs, we can translate FBD programs into the Yices input language. As shown in Figure 1, MuFBDTester defines the scan time and constant variable in Lines 6 and 7. It is necessary to include scan time information to translate the behaviour of the FBD programs because they are executed cyclically. The constant variable K_DELAY is the preset time of the TON function in simTRIP. We define a set of SMT constraints for executing an FBD program for one cycle. The set of constraints includes input variables, intermediate variables and output variables. MuFBDTester defines input variables (TSP, TRIP_LOGIC and PV_OUT) in Lines 27–29. To translate functions and function blocks, we use the block operation library file that includes the operational definitions of standard functions and function blocks. Take the GE function as an example. Based on connection information, IN1 (input 1) of GE function is connected to PV_OUT, and IN2 (input 2) of GE function is connected to TSP. In Line 30, this GE function is converted into '>= PV_OUT TSP' ('operation IN1 IN2'). The output of the GE function will be captured as a new variable 'GE_out'. MuFBDTester defines output variables (TON_et and TRIP_LOGIC) in Lines 32 and 33. TON_et and TRIP_LOGIC will capture the output values of ET and Q in the TON function, respectively.

The example Yices file represents the execution of the example program simTRIP for three cycles. Thus, the set of constraints is repeated for three times in Lines 9–33. To distinguish different cycles, we give the variables different variable names with the suffix '_t#'. For instance, in Lines 11–17, these variables are in two cycles ago from the current cycle, so the variables' suffixes are '_t2'. The number in the suffix denotes how many cycles ago from the current cycle. We do not give the suffix to the variables in the current cycle, and Lines 27–33 represent the current cycle. The TON function block considers the input parameters in the current cycle and the value of ET (the elapsed time that the value of IN has been true) in the previous cycle, when the TON function block delivers the output results. In the IEC61131-3 standard [1], the data type of the ET variable is specified as time. The value of the time variable is either reset to zero or increasing. Thus, in Line 9, we initialize the value of TON_et at three cycles ago to the starting state, which is zero.

4.3.2 | Differences between target program and mutant

In Figure 1, the target FBD program is defined in Lines 9–33, and the mutant is defined in Lines 38–54. To distinguish between the target FBD program and the mutant, all the intermediate results and output variables of the mutant have the prefix 'M_' attached to their names. MuFBDTester does not define input variables in the definition of the mutant. The key concept is to find the input data that can generate results of the mutant that are different from those of the target program. Thus, the input variables are defined only once so that the input data can be delivered to both the target program and the mutant. For instance, in Line 40, the output of the GE function in the mutant uses the variables PV_OUT_t2 and TSP_t2 as inputs.

4.3.3 | Test requirement and command

The test requirement (Line 56) is to find a solution to distinguish the outputs of the target FBD program from those of the mutant. There should be at least one output value of the target FBD program that is different from the mutant. The check command (Line 59) is to check if there is a solution to fulfil the test requirement. The SMT solver can efficiently determine the satisfiability of the test requirements. The Yices SMT solver reports three main types of results: sat, unsat and unknown. If the problem is satisfiable, the Yices SMT solver reports 'sat' and prints the solution. If the problem is not satisfiable, it returns 'unsat'. If the solver cannot find a solution within a fixed number of iterations, it returns 'unknown'. If the test requirement is not satisfiable, the mutant is an equivalent mutant because there is no solution to differentiate between the target program and the equivalent mutant. Our approach can detect equivalent mutants automatically. It does not suffer from the burden of identifying equivalent mutants manually. In addition, no effort is wasted to search for the infeasible test data of the equivalent mutants. We also found that some mutants of ABR operators would introduce nonlinear problem. The Yices SMT solver cannot resolve nonlinear problems; apart for equivalent mutants, these are the only cases for which our tool cannot find the test sequence.

4.4 | Test suite generation

To generate mutation-based test suites for FBD programs, this process includes the execution of the Yices SMT solver, parsing the outputs of Yices execution and outputting test data. For each mutant, a test sequence was obtained using

Yices execution. After collecting all the test sequences, we filtered out duplicates. Thus, the test suite generated by MuFBDTester is not just an aggregation of test data generated by multiple executions of Yices. Instead, the test suite is a union set of all the test data generated.

5 | EVALUATION

To demonstrate the effectiveness and efficiency of our proposed approach, we conducted experiments on several subject programs. We compared our developed tool, MuFBDTester, with related testing tools, including the most advanced structural coverage-based test generation tool FBDTester 2.0 [8] and other mutation-based test generation work proposed by Enoiu et al. [16]. We executed FBDTester 2.0, which is available on GitHub,[3] and re-implemented Enoiu et al.'s approach by utilizing their proposed test generation tool CompleteTest.[4] The experiments were performed on a Windows machine with Intel Core i7-7700 Processor and 16 GB memory. We designed our experiments to answer the following four questions.

- **RQ1.** Does the mutation-adequate test suite exhibit better fault detection effectiveness than that exhibited by the test suites conforming to structural coverage criteria?
  RQ1-1. Does it exhibit better fault detection effectiveness on artificial faults?
  RQ1-2. Does it exhibit better fault detection effectiveness on real faults?
- **RQ2.** Does MuFBDTester exhibit better efficiency than that exhibited by other related tools?

5.1 | Subject programs

We chose our subject programs from the Korean Nuclear Instrumentation and Control System (KNICS) project's Bistable Processor (BP) system [60] implemented in FBD. The BP system, a part of the reactor protection system, is used to determine whether a nuclear reactor must be stopped or not. Hence, the BP system is considered as a safety-critical system. The BP system consists of more than 1000 blocks and more than 1000 input variables. For the BP system, two experienced FBD engineers required 6 weeks for two experienced FBD engineers to perform its unit testing, which included functional test creation, execution and composition of the unit test report. The unit test report comprises 139 pages and over 300 test cases [61].

Approximately 20 modules included in the BP system can be categorized into six types: fixed-falling trip decision (FFTD), fixed-rising trip decision (FRTD), variable-rate-falling trip decision (VFTD), variable-rate-rising trip decision (VRTD), manual-reset-falling trip decision (MFTD) and heartbeat (HB) monitoring. One module for each type of module in the BP system is selected. Furthermore, the combinedTD module is developed by combining several modules in the BP system. As some function block groups defined in IEC61131-3 [1] are not used in the BP system, we include three more subject programs, simTRIP, simGRAVEL and LAUNCHER, in our experiments. simTRIP, shown in Figure 1, includes the TON function block from the timer group; simGRAVEL, shown in Figure 4, includes the CTU function block from the counter group and the TON function block; LAUNCHER, shown in Figure 5, includes the R_TRIG function block from the edge-detection group and the SR function block from the bistable group. simGRAVEL is used to control the amount of gravel transferred from a silo to a bin, and LAUNCHER is a module to launch a system. Table 2 shows the information of each subject program.

5.2 | Experimental setup and design

5.2.1 | RQ1-1. Fault detection effectiveness on artificial faults

Statistical comparisons

We performed statistical tests to compare the fault detection effectiveness of mutation-based and structural coverage-based test suites on artificial faults. To obtain artificial faults, we used six mutation operators, different from those used to generate mutation-based test suites. Among these six mutation operators, five were discussed by Enoiu et al. [16], which they plan to further develop in future work, and we have implemented them based on definitions in this study for experiments. The last operator is the variable replacement (VARR) operator. The VARR operator was a candidate mutation operator when Jee et al. defined 13 mutation operators [21] because there were several misused

[3] https://github.com/sarahsong7/FBDTester_v2.0.
[4] https://www.completetest.org/.

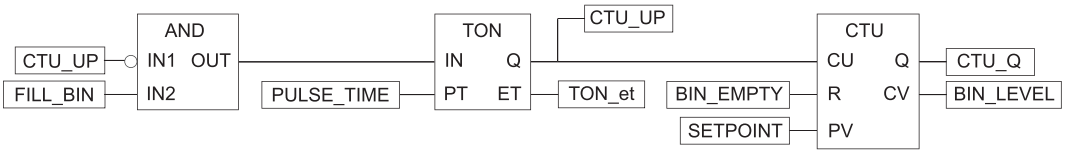

FIGURE 4 Subject program called simGRAVEL

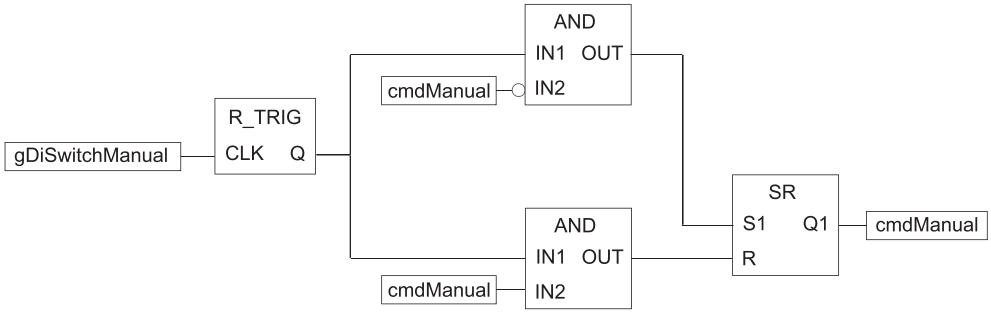

FIGURE 5 Subject program called LAUNCHER

TABLE 2 Subject program information

Subject	#blocks	#function blocks	#inputs	#outputs
simTRIP	3	Timer: 1	3	2
simGRAVEL	3	Timer: 1, counter: 1	3	4
LAUNCHER	4	Edge detection: 1, bistable element: 1	2	1
FFTD	29	Timer: 2	12	8
FRTD	29	Timer: 2	12	8
VFTD	44	Timer: 2	16	8
VRTD	44	Timer: 2	17	8
MFTD	47	Timer: 2	21	8
HB	19	-	6	1
combinedTD	437	Timer: 26	221	92

variable errors found in real faults. However, the variable replacement operator was not included in the final 13 mutation operators due to its high cost. We defined and implemented the VARR operator in this study for experiments. Mutation operators used in statistical comparisons are defined as follows.

- Feedback Loop Insertion Operator (FIO) [16]: A connection is inserted between an output variable and any block that is connected with the input variables.
- Logical Block Insertion Operator (LIO) [16]: A logical block is inserted between any two other logical blocks in the program.
- Logical Block Deletion Operator (LDO) [16]: A logical block is deleted, and the inputs of this block are connected to the next logical block in the program.
- Value Replacement Operator-Improved (VRO-I) [16]: A constant C is replaced not only with its boundary values but also with non-boundary values including 0, 1 and −1.
- Logical Block Replacement Operator-Improved (LRO-I) [16]: A logical block is replaced with logical blocks from the same group and blocks with Boolean inputs (e.g., an AND block can be replaced with an SR block).
- Variable Replacement Operator (VARR): A variable is replaced with any other variable that has the same data type.

We generated all possible first-order mutants using these mutation operators as faulty programs. It should be noted that a fault is said to be revealed by a test suite if it can distinguish the output values of the faulty program from those of the subject program.

In this experiment, the subject program set comprised BP system modules, including FFTD, FRTD, VFTD, VRTD and MFTD modules. For each BP system module, we generated mutation- and structural coverage-based test suites 10 times with random seed settings in the Yices SMT solver. Thus, we generated a different test suite on each run. We used MuFBDTester to generate mutation-adequate test suites and FBDTester 2.0 [8] to generate test suites based on the structural coverage criteria: BC, ICC and CCC defined in [62]. We set four cycles of test inputs for each test sequence, as the minimum iteration number for all subjects was four. Additionally, FBDTester 2.0 has the enhanced mechanism to cover not-yet-satisfied test requirements by unwinding more scan cycles of the given FBD program. However, we did not use this mechanism in this experiment so that all test sequences could share the same number of cycles in a test suite. Furthermore, MuFBDTester has a tendency to generate more test sequences than FBDTester 2.0 owing to the large number of mutants. To control the number of test sequences in a test suite, we utilized the splitting factor setting of FBDTester 2.0. The splitting factor that can split the test requirements into slices then attempts to find test data to satisfy each slice of the test requirements. Thus, with different splitting factors, FBDTester 2.0 generates test suites that have various numbers of test sequences. For each mutation-based test suite, we attempted to find the structural coverage-based test suites with the closest number of test sequences. The size information of each generated test suite can be found in Table A1. The test sets guided by the BC, ICC and CCC criteria are denoted as BC-suite, ICC-suite and CCC-suite, respectively. The mutation-based test suite generated by MuFBDTester is denoted as M-suite. With comparable numbers of test sequences, MuFBDTester and FBDTester 2.0 attempted to reach the maximum level of mutation and structural coverages, respectively.

For each test suite, we created five sample test sets of 10, 20, 30, 40 and 50 test sequences, resulting in a total of 5000 sample test sets across the five subject programs (i.e., 5 subject programs $\times$ 4 test generation methods $\times$ 10 runs $\times$ 5 test sizes $\times$ 5 sample test sets). For each sample test set, we calculated the fault detection score by executing the subject program and its faulty program on the sample test set. If the execution results of the subject program varied from those of a faulty program, the fault in the faulty program was considered to be detected. The fault detection score was calculated by dividing the number of detected faults with the number of faulty programs. For each test size, we conducted the Wilcoxon matched-pairs signed-rank test for two reasons: (1) the collected fault detection scores have matched pairs as we generated M-, BC-, ICC- and CCC-suites from the same subject program, and (2) the distributions of fault detection scores and score differences of any two approaches did not follow a normal distribution, so we needed a non-parametric test. Thus, given that we had a large number of sample test sets, paired design and no assumptions of data distribution, we selected the Wilcoxon matched-pairs signed-rank test.

Replicated experiment of the previous study: Mutation analysis

Song et al. [8] conducted mutation analysis to investigate the effectiveness of BC-, ICC- and CCC-suites. We extended Song et al.'s experiment to measure the effectiveness of the generated test data and to verify that our approach functions as we expected. We used the same set of subject programs used in Song et al.'s experiment: simTRIP, simGRAVEL, LAUNCHER, FFTD, FRTD, VFTD, VRTD, MFTD and combinedTD modules.

Based on the defined mutation operator set [21], we generated all possible first- and second-order mutants by combining two first-order mutants. The mutation operator set [21] employed in this experiment is thus an extended version of the original operator set from Song et al.'s experiment. Hence, generated mutants still included those used in the mutation analysis of Song et al.'s study.

Test suites in this experiment were generated when MuFBDTester and FBDTester 2.0 were set to their default settings. Furthermore, we used the enhanced mechanism of FBDTester 2.0 for this experiment. Thus, FBDTester 2.0 unwinds more cycles of target program when there remains unsatisfied test requirements. Hence, the test suites generated by FBDTester 2.0 had test sequences containing different numbers of cycles of test inputs. Moreover, we executed MuFBDTester and FBDTester 2.0 without random seed settings. Thus, the Yices SMT solver produced the same results by default in each run. In this experiment, we ran MuFBDTester and FBDTester 2.0 once for each subject program. For the test suites generated by FBDTester 2.0 and MuFBDTester, we separately measured the mutation adequacy scores on the generated first- and second-order mutant set.

Test oracle

We prepared test oracles semi-automatically because the testing tools only produced the test data. Fortunately, requirement specifications for the target industrial programs were well prepared. We implemented java code following the requirement specification and utilized it to calculate the expected outputs for the given input data. Moreover, the implemented java code also performs verdict computation in our experiment to determine the liveness of mutants. Because the test suite is composed of test sequences, we reset all the internal memory states before feeding a test sequence to the java code.

5.2.2 | RQ1-2. Fault detection effectiveness on real faults

In the KNICS project, software design for the BP system in FBD programs was formally verified. Several potential faults were found for the preliminary version of the BP system [63] and the developers revised the BP program by correcting the faults. A total of 47 errors were found, and we reproduced only the distinct errors because a distinct module is repeatedly used in the BP system. By using the information on these faults, we created nine faulty FBD programs, each of which included a distinct real fault. Table 3 shows the information regarding the reproduced faulty programs.

Mutation- and structural coverage-based test suites were generated from the original faulty programs instead of the corrected programs, because an engineer had introduced an error in the program and actually tested the faulty program. In this experiment, MuFBDTester and FBDTester 2.0 were set to their default settings, so both test generators were run once for each faulty program. We observed the outputs obtained from the corrected program and the faulty program with the test suite that we generated. If the outputs of the corrected program and the faulty programs are not identical, the test suite can detect faults that the faulty programs contained.

Among the real faults, some were related to transfer definitions. A transfer definition means that an output variable is connected to an input variable, as shown in Figure 6. In other words, the input value of one variable is assigned by the output value of the other. To reproduce the real faults, the program parts of transfer definitions, which were not included in Song et al.'s [8] subject programs, needed to be included in the subject programs. The transfer definitions were included in the VFTD, VRTD and MFTD modules. The star sign in Table 3 denotes the original subject programs including transfer definitions.

In these subject programs, some variables indicate the values of a certain variable in 19 cycles ago. Thus, to capture the values of previous 19 cycles, the transfer definitions were used several times to flush values. Figure 6 shows a part of the transfer definitions in the VFTD module. At the end of a cycle, the value of the TSP variable in the current cycle was stored in TSP_t1, the value of TSP_t1 was stored in TSP_t2 and so on. In this case, the program would be able to capture the values of the TSP variable for 20 cycles. One of the real faults was found in the transfer definition parts in the VFTD module. Owing to the similarity of the variable names, PTSP_t17 in Figure 6 is misplaced. TSP_t16 should be connected to TSP_t17 instead of PTSP_t17.

Some faults occurred while the results of multiple decision modules were combined by the OR function; thus, we included two more subject programs—FRTD+MFTD and VFTD+VRTD—as shown in Table 3.

TABLE 3 Faulty program information

Program	#blocks	#inputs	#outputs	Error cause
FRTD-1	29	12	8	Incorrect operator
FRTD-2	29	12	8	Incorrect operator
VFTD	44	56*	48*	Misused variable name
VRTD	44	57*	48*	Incorrect operator
MFTD	47	40*	28*	Incorrect operator
HB-1	18	6	1	Incorrect logic
HB-2	19	6	1	Remaining test code
FRTD+MFTD	76	53	37	Misused variable name
VFTD+VRTD	89	114	97	Misused variable name

*The original version of subject program including transfer definitions.

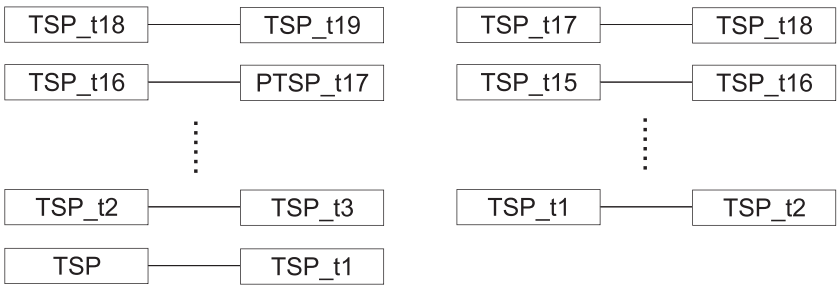

FIGURE 6 Transfer definitions in the VFTD module

5.2.3 | RQ2. Efficiency

We compared the test generation time of the test sequences guided by BC, ICC and CCC in FBDTester 2.0 and those of the mutation-based test sequences generated by MuFBDTester and Enoiu et al.'s approach [16] for the subject FBD programs. The tool CompleteTest [40] transforms FBD programs into timed automata and generates structural coverage-based test sequences using the UPPAAL model checker. Based on the tool CompleteTest, we utilized the transformation of FBD programs into the UPPAAL model to re-implement Enoiu et al.'s [16] mutation-based test generation approach. Enoiu et al. proposed six mutation operators; therefore, we also re-implemented the mutant generation method according to the proposed mutation operator set.

5.3 | Experimental results

5.3.1 | RQ1-1. Fault detection effectiveness on artificial faults

Statistical comparisons

For each of the three pairs of test generation approaches, we performed a two-tailed Wilcoxon matched-pairs signed-rank test comparing the fault detection scores of the mutation- and structural coverage-based approaches for artificial faults. The information of artificial faults is shown in Table 4. Table 5 lists the statistical test results of paired comparisons (M-suite against BC-, ICC- and CCC-suites). All the p-values in the comparison results are under 0.01. Thus, we can reject the null hypothesis that there is no significant difference between the two data groups. The paired comparison results indicate that M-suites were significantly stronger in detecting artificial faults than BC-, ICC- and CCC-suites with restricted test sizes. In addition, the agreement of statistical comparisons stayed the same for different test sizes. The p-value represents the probability that the data could have occurred under the null hypothesis. The lower the p-value, the greater the statistical significance of the difference between the two data groups. Table 5 shows that as the test size increased, p-values decreased for all the paired comparisons. Thus, when we expanded the test size, stronger evidences against null hypothesis were observed. A confidence interval represents a range of estimated values for the median of fault detection score differences between approaches. The greater the values in confidence interval, the greater the difference between the two data groups. When M-suite is compared to CCC-suite, the estimated median of the score differences constantly increases with the test size. The effect size represents practical significance. The effect size r is calculated as the test statistic z divided by the squared root of the number of pairs. According to Cohen's effect size level [64], the effect sizes are either moderate (0.3–0.5) or large ($\geq$ 0.5). Therefore, the results of paired comparisons have not only statistical significance but also practical significance. These results indicate that M-suites are statistically and practically superior to BC-, ICC- and CCC-suites in view of fault detection effectiveness on artificial faults.

Figure 7 shows the distributions of fault detection scores for all test suites. When test sizes were 30–50 test sequences, the distributions of fault detection scores for M-suite mainly fell at the range of 92.5–100%. For BC-, ICC- and CCC-suites, the distributions of fault detection scores are more scattered. In addition, BC-, ICC- and CCC-suites have lower densities on the range of 92.5–100% fault detection score compared to M-suite. M-suites have a tendency to have higher fault detection scores than BC-, ICC- and CCC-suites. Thus, mutation-based test suites perform better on detecting artificial faults than structural coverage-based test suites.

Replicated experiment of the previous study: Mutation analysis

Table 6 shows the mutation scores of the mutation-based test sets and the test sets that were guided by the BC, ICC and CCC criteria running on first-order mutants. Detailed information of first-order mutants are shown in Table 7. The first-order equivalent mutants were detected by MuFBDTester and then manually verified. In this experiment, we

TABLE 4 Artificial fault information

Mutation operator	FFTD	FRTD	VFTD	VRTD	MFTD
FIO	30	30	9	9	76
LIO	24	24	24	24	24
LDO	3	3	3	3	4
VRO-I	0	0	6	6	0
LRO-I	14	14	16	16	16
VARR	78	78	237	237	315
Total	149	149	295	295	435

TABLE 5 Results of statistical comparisons of M-suite, BC-, ICC- and CCC-suites

Test size = 10				
Paired comparisons (winner > loser)	**Agree?**	***p*-value**	**95% confidence interval**	**Effect size**
M-suite > BC-suite	Yes.	4.73e-13	[2.01, 3.36]	0.45
M-suite > ICC-suite	Yes.	3.01e-12	[1.64, 2.71]	0.44
M-suite > CCC-suite	Yes.	8.22e-09	[1.02, 2.12]	0.36
Test size = 20				
Paired comparisons (winner > loser)	**Agree?**	***p*-value**	**95% confidence interval**	**Effect size**
M-suite > BC-suite	Yes.	8.54e-22	[1.93, 3.01]	0.60
M-suite > ICC-suite	Yes.	3.21e-20	[1.52, 2.34]	0.58
M-suite > CCC-suite	Yes.	7.70e-30	[2.44, 3.27]	0.71
Test size = 30				
Paired comparisons (winner > loser)	**Agree?**	***p*-value**	**95% confidence interval**	**Effect size**
M-suite > BC-suite	Yes.	1.00e-25	[1.88, 3.02]	0.66
M-suite > ICC-suite	Yes.	3.04e-24	[1.51, 2.34]	0.64
M-suite > CCC-suite	Yes.	2.16e-37	[2.85, 3.52]	0.80
Test size = 40				
Paired comparisons (winner > loser)	**Agree?**	***p*-value**	**95% confidence interval**	**Effect size**
M-suite > BC-suite	Yes.	1.97e-30	[2.01, 2.85]	0.72
M-suite > ICC-suite	Yes.	5.23e-29	[1.67, 2.35]	0.70
M-suite > CCC-suite	Yes.	6.42e-39	[3.18, 3.82]	0.82
Test size = 50				
Paired comparisons (winner > loser)	**Agree?**	***p*-value**	**95% confidence interval**	**Effect size**
M-suite > BC-suite	Yes.	7.82e-36	[2.10, 2.80]	0.79
M-suite > ICC-suite	Yes.	7.27e-34	[1.55, 2.34]	0.76
M-suite > CCC-suite	Yes.	4.36e-40	[3.25, 3.93]	0.83

Note: Test size: number of test sequences.

set the splitting factor to one for FBDTester 2.0. However, we could not obtain test sets from FBDTester 2.0 for the combinedTD module when the splitting factor was set to one. Thus, in Table 6, the mutation scores of BC-suite, ICC-suite and CCC-suite are 'unknown' in the case of combinedTD. FBDTester 2.0 utilized the 'max-sat' command to search for the test data to satisfy the maximum test requirements. For the combinedTD module, FBDTester 2.0 yielded over 10,000 test requirements; consequently, the Yices SMT solver had difficulty finding the solution to satisfy the maximum test requirements within a fixed number of iterations. This is heavily dependent on the performance of the SMT solver. For all the subject programs, the mutation-based test sets had the highest mutation scores among all the approaches. On average, BC-suite, ICC-suite and CCC-suite achieved mutation scores of 93.7%, 97.7% and 98.1%, respectively. The mutation-based test sets succeeded in killing all the mutants except for equivalent mutants; that is, it achieved a 100% mutation score for all the subject programs. These results demonstrate that the mutation-based test suites generated by MuFBDTester are highly effective in detecting all potential errors expressed by the mutation operators.

Table 8 shows the coverage results of the test suites generated by FBDTester 2.0 and MuFBDTester, with respect to the three test coverage criteria for the FBD programs. On average, the mutation-based test suites achieved 91% BC, 87% ICC and 81% CCC. Although the structural coverage achieved by the mutant-based test suites is less than those generated maximizing the structural coverage, it can be confirmed that the mutant-based test suites also achieved a fairly high level of structural coverage. The fact that the coverage level of the mutation test suites is not always superior to BC-suite, ICC-suite and CCC-suite might be caused by many factors, such as the test set size and the subsuming relationship between mutant and coverage. The analysis of possible factors is being considered for future work.

Furthermore, while there are some cases for which BC-suite, ICC-suite and CCC-suite reached 100% coverage level, those test suites failed to detect 100% of mutants. For instance, the BC-suite, ICC-suite and CCC-suite generated for simTRIP achieved 100% coverage level, but they failed to detect some mutants. Conversely, the mutation-based

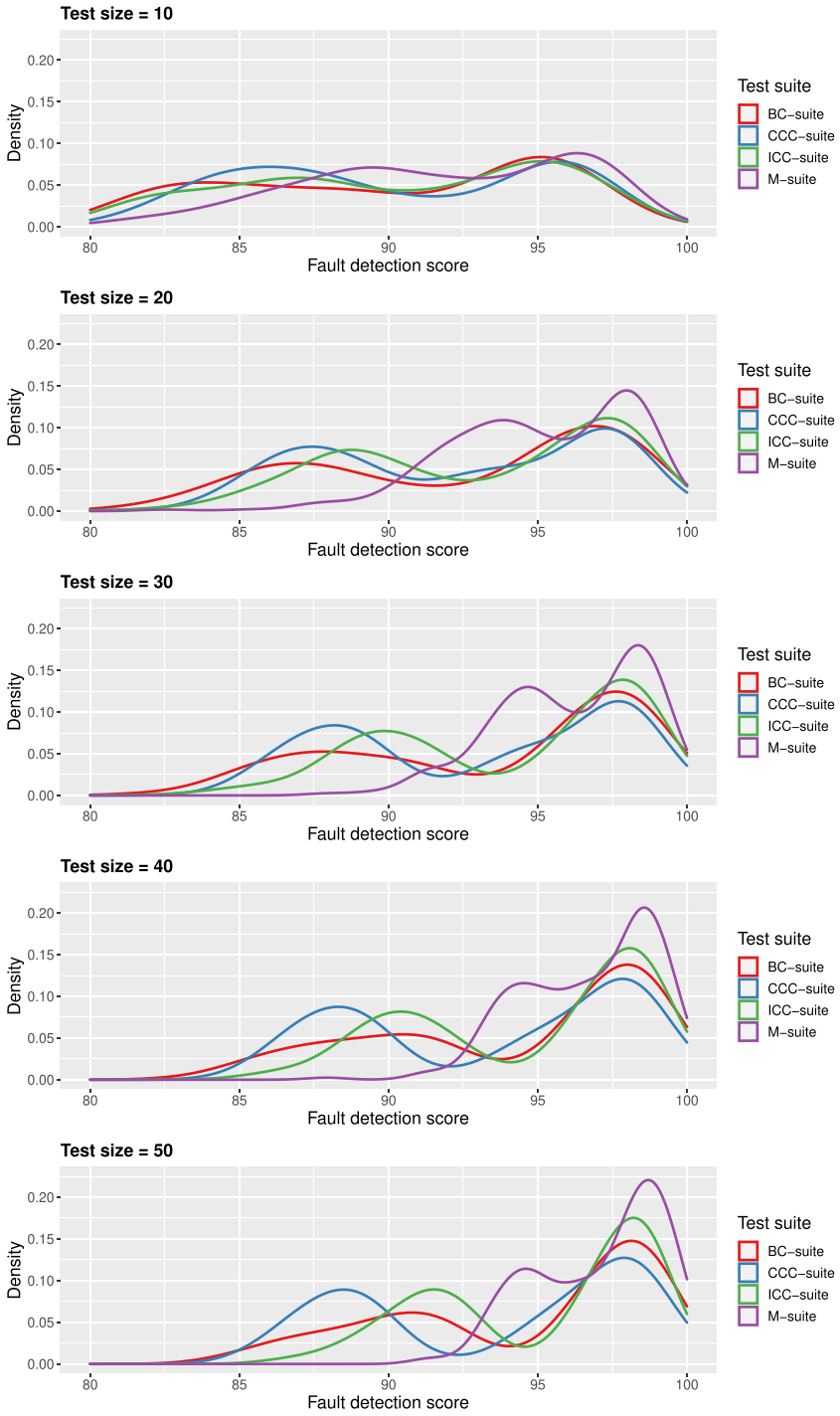

FIGURE 7 Distributions of fault detection scores for all test suites

test suite generated for simTRIP achieved 100% coverage of the three coverage criteria and also detected 100% of mutants.

Figure 8 shows the mutants that were not detected by BC-suite, ICC-suite and CCC-suite. Four types of mutants were not detected by the test suites guided by the structural coverage criteria. These four types of mutants are generated by the CBR, IID, SWI and LBR mutation operators. As shown in Figure 8, it was found that most of the non-killed mutants were generated by the CBR mutation operator. This is probably because the structural coverage criteria that FBDTester 2.0 is based on are not strong enough to guarantee detection for these mutants.

As the mutation-based test set size can be larger than the FBDTester 2.0 test set size, we investigated whether the test set size would have an impact on the mutation score. When we conducted mutation analysis, the splitting factor of

TABLE 6 Mutation score on first-order mutants

Subject	#mutants	#equivalent mutants	M-suite	BC-suite	ICC-suite	CCC-suite
simTRIP	16	0	100	75.0	93.7	93.7
simGRAVEL	13	0	100	100	100	100
LAUNCHER	18	1	100	88.2	100	100
FFTD	133	0	100	95.4	96.2	97.7
FRTD	133	0	100	96.9	96.9	97.7
VFTD	202	3	100	98.4	98.9	99.4
VRTD	203	3	100	99.0	99.0	100
MFTD	215	3	100	97.1	97.6	97.1
combinedTD	1948	9	100	Unknown	Unknown	Unknown
Average	320.1	-	100	93.7*	97.8*	98.2*

*Average scores are calculated by only known scores.

TABLE 7 Number of first-order mutants per mutation operator

Mutation operator	simTRIP	simGRAVEL	LAUNCHER	FFTD	FRTD	VFTD	VRTD	MFTD	combinedTD
ABR	-	-	-	16	16	40	40	32	280
CBR	5	-	-	30	30	45	45	55	435
LBR	2	2	4	18	18	20	20	24	260
TBR	2	2	-	4	4	4	4	4	52
BBR	-	-	1	-	-	-	-	-	-
EBR	-	-	1	-	-	-	-	-	-
CouBR	-	1	-	-	-	-	-	-	-
CVR	-	-	-	-	-	8	8	-	24
IID	6	8	11	49	49	59	59	71	421
SWI	1	-	1	16	16	26	27	29	476

TABLE 8 Coverage of test suites

	BC			ICC			CCC		
Subject	#TR	BC-suite	M-suite	#TR	ICC-suite	M-suite	#TR	CCC-suite	M-suite
simTRIP	4	100	100	6	100	100	24	100	100
simGRAVEL	13	92.3	92.3	29	86.2	75.8	69	85.5	68.1
LAUNCHER	4	100	100	12	83.3	83.3	32	78.1	78.1
FFTD	126	100	100	186	100	98.9	902	95.1	94.6
FRTD	126	100	100	186	100	98.9	902	95.1	94.6
VFTD	2,090	85.6	85.2	2,292	85.5	85.4	23,826	84.4	76.9
VRTD	2,090	85.8	85.7	2,292	85.9	85.9	23,826	84.6	77.2
MFTD	4,256	86.1	71.4	6,282	82.9	67.8	55,138	82.0	58.2
Average	1,088	93	91	1,410	90	87	13,089	88	81

Note: #TR: number of test requirements.

FBDTester 2.0 was set to one. However, if the splitting factor increases, the test set size might also increase. Thus, testers can choose different splitting factors to generate different numbers of test data. Tables 9 and 10 show the CCC-suite size and mutation scores with different splitting factors on three small programs and the BP system modules, respectively. For the three small programs and the FFTD, FRTD and VRTD modules, the mutation scores of CCC-suite remained the same with different test set sizes. For the VFTD and MFTD modules, the mutation scores of CCC-suite differed for different splitting factors. We checked the number of killed mutants due to the differences. The differences were either one or two mutants. Furthermore, in the case of the FFTD and FRTD modules, when the splitting factor was four, the test set size of CCC-suite was similar to that of the mutation-based test suite. However,

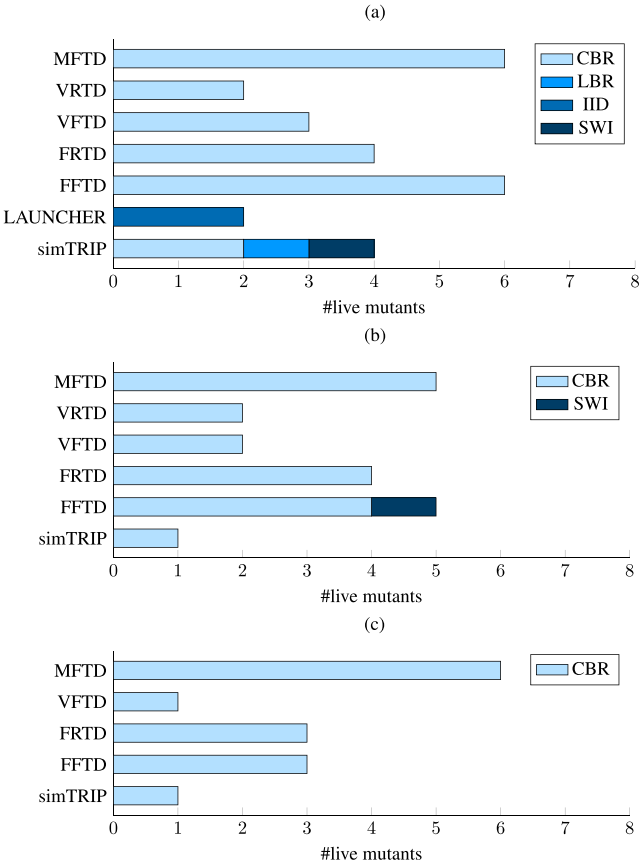

FIGURE 8 Type of mutants that FBDTester 2.0 failed to detect

TABLE 9 The results of CCC-suite for three FBD programs with different splitting factors

	simTRIP		simGRAVEL		LAUNCHER	
Splitting factor	**Test set size**	**Mutation score**	**Test set size**	**Mutation score**	**Test set size**	**Mutation score**
1	2	93.7	16	100	19	100
2	3	93.7	27	100	27	100
3	3	93.7	40	100	22	100
4	4	93.7	40	100	21	100
5	2	93.7	35	100	35	100
M-suite	10	100	10	100	8	100

Note: Test set size: number of test sequences.

TABLE 10 The results of CCC-suite for BP FBD programs with different splitting factors

	FFTD		FRTD		VFTD		VRTD		MFTD	
Splitting factor	**Test set size**	**Mutation score**	**Test set size**	**Mutation score**	**Test set size**	**Mutation score**	**Test set size**	**Mutation score**	**Test set size**	**Mutation score**
1	30	97.7	30	97.7	158	99.4	165	100	246	97.1
2	60	97.7	60	97.7	274	98.9	266	100	389	97.6
3	94	97.7	95	97.7	383	99.4	410	100	586	98.1
4	123	97.7	125	97.7	452	99.4	452	100	728	98.1
5	149	97.7	149	97.7	532	99.4	530	100	883	97.6
M-suite	83	100	83	100	128	100	129	100	132	100

Note: Test set size: number of test sequences.

TABLE 11 Mutation score on second-order mutants

Subject	#mutants	#equivalent mutants	M-suite	BC-suite	ICC-suite	CCC-suite
simTRIP	108	4	100	93.2	97.1	97.1
simGRAVEL	76	0	100	100	100	100
LAUNCHER	151	0	100	94.7	100	100
FFTD	8,683	34	99.9	99.4	99.6	99.7
FRTD	8,683	34	99.9	99.6	99.6	99.7
VFTD	20,127	90	99.9	99.8	99.8	99.9
VRTD	20,329	90	99.9	99.8	99.8	99.9
MFTD	22,833	112	99.9	99.6	99.7	99.7
Average	10,123.7	-	99.9	98.3	99.5	99.5

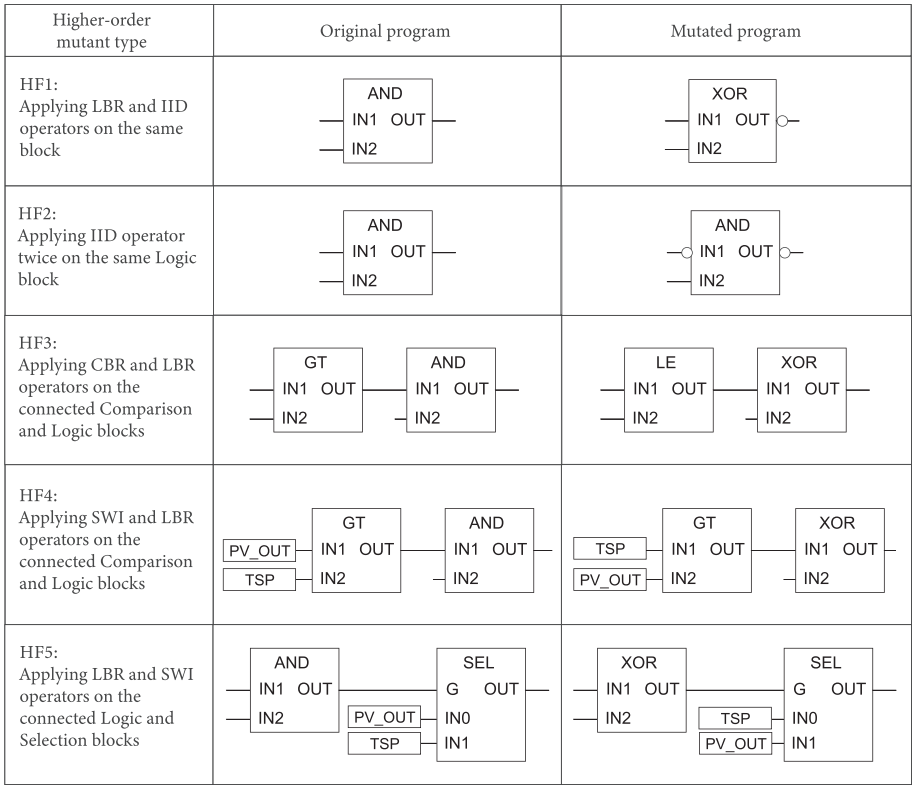

FIGURE 9 Live higher-order mutants

the mutation score of CCC-suite remained the same. Overall, Tables 9 and 10 show that although the test set size of CCC-suite increased, the mutation score did not improve.

We also investigated the effectiveness of test suites for detecting higher-order mutants. If a mutant includes more than one mutation, the mutant is called a higher-order mutant. Specifically, we generated over 10,000 higher-order mutants on average by pairing first-order mutants. We manually analysed live second-order mutants and identified more than 300 equivalent mutants. In Table 11, the mutation adequacy scores of the test suites are measured by running on second-order mutants. For all the subject programs, the M-suite achieved at least 99.9% mutation adequacy scores. On average, BC-suite, ICC-suite and CCC-suite achieved mutation scores of 98.3%, 99.5% and 99.5%, respectively. The mutation-based test sets detected a large percentage of second-order mutants.

As approximately 0.1% of mutants were not killed, we investigated whether M-suite is weak in some special cases. As shown in Figure 9, we categorized live non-equivalent mutants into five types. We defined HF as faults caused by higher-order mutation. HF1 contained a combination of LBR and IID mutation operators on the same block. HF2 contained the change obtained by applying the IID mutation operator twice on the same logic block. HF3, HF4 and HF5 applied two mutations on the connected two blocks from different groups. HF3 contained the application of CBR

and LBR mutation operators on the connected comparison and logic blocks. HF4 contained the change obtained by applying the SWI and LBR operators on the connected comparison and logic blocks. HF5 contained the change obtained by applying the LBR and SWI operators on the connected logic and selection blocks. Notably, all types of live higher-order mutants were related to logic blocks. The found higher-order mutant types can be considered to improve the mutation operators and to enhance the effectiveness of the test suite.

5.3.2 | RQ1-2. Fault detection effectiveness on real faults

We reproduced nine faulty programs, each of which included a single real fault. As shown in Table 3, the errors were caused by incorrect operator, misused variable name, incorrect logic and remaining test code. The incorrect operator is similar to the block replacement. For example, in the FRTD-2 program, a wrong comparison block was misplaced. Errors were caused by misused variable names when variables had highly similar names, such as 'PTSP_t17' and 'TSP_t17'. The incorrect logic cannot be fixed by a single change. For instance, in the HB-1 program, a selection block was omitted, and an inverter was inserted. The test code could be that an output variable was directly assigned to false at the end of the program. In this experiment, the mutation-based test suites successfully detected all the reproduced faults. ICC- and CCC-suites also identified all the reproduced faults except for BC-suites. However, BC-suites failed to detect the fault in the FRTD-2 program. These results correspond to that BC-suites are relatively weak at detecting incorrect comparison blocks.

5.3.3 | RQ2. Efficiency

Comparison of mutation-based and structural coverage-based approaches

We compared the test generation time of MuFBDTester and FBDTester 2.0. In this experiment, the preset time values of the subject programs were all set to 100, and the scan time was set to 50. Table 12 shows the test generation time and test suite size of MuFBDTester and FBDTester 2.0. Here, we define the total test input length as the total number of test inputs in a test sequence set. In Table 12, for the three simple FBD programs, both FBDTester 2.0 and MuFBDTester generated test suites within 1 s. For all the test suites, the average generation time per test sequence was approximately 0.05 s. As shown in Table 12, for more complex programs, the generation time of CCC-suite increased significantly by up to 12 h. For complex programs (VFTD, VRTD and MFTD modules), the mutation-based test suite generation time was more than that of ICC-suite but less than that of CCC-suite. For BC-suite, ICC-suite and M-suite, the average generation time per test sequence was approximately 1.5 s. However, for CCC-suite, the average generation time per test sequence was approximately 20 s. In Table 6, CCC-suite for the VRTD module shares the same degree of fault detection effectiveness as M-suite. CCC-suite for the VRTD module took 1 h to be generated while MuFBDTester took approximately 3 min to generate the mutation-based test suite. MuFBDTester took a reasonable amount of time to generate highly effective test sequences for complex programs.

FBDTester 2.0 utilized the strategy to reduce the test set size by using the 'max-sat' command. The 'max-sat' command searches for the solution to achieve the maximum test requirements; thus, it requires much more resources to search for test data. Thus, FBDTester 2.0 took a much higher amount of time to generate test suites based on a strong structural coverage criterion. For instance, for VFTD, VRTD and MFTD modules, the numbers of CCC test requirements increased up to 20,000–50,000 to be satisfied, as shown in Table 8. This massive amount of test requirements

TABLE 12 Test suite generation time and test suite size (sec / number of test sequences [total test input length])

Subject	Mutant generation	M-suite	BC-suite	ICC-suite	CCC-suite
simTRIP	0.047	0.87/10 (40)	0.06/1 (4)	0.08/2 (8)	0.07/2 (8)
simGRAVEL	0.040	0.82/10 (40)	0.71/15 (90)	0.69/15 (90)	0.67/16 (98)
LAUNCHER	0.056	0.89/8 (32)	0.08/2 (8)	0.64/20 (120)	0.65/19 (116)
FFTD	0.510	19.2/83 (332)	0.6/4 (16)	0.5/5 (20)	8.4/30 (180)
FRTD	0.561	19.7/83 (332)	0.3/4 (16)	0.5/5 (20)	8.2/30 (180)
VFTD	1.009	181.1/128 (512)	50.3/73 (430)	59.2/74 (438)	4145.6/158 (945)
VRTD	1.300	173.9/129 (516)	59.3/69 (411)	73.6/76 (453)	4270.4/165 (985)
MFTD	1.206	439.3/132 (528)	181.2/117 (698)	333.5/118 (702)	46,335.8/246 (1477)
combinedTD	292.748	325,366.6/1296 (5184)	Unknown	Unknown	Unknown

TABLE 13 Test suite generation time and test suite size (sec / total test input length)

Subject	M-suite	E-suite
simTRIP	0.82/40	34.02/11
LAUNCHER	0.83/28	1.43/3

TABLE 14 Mutation adequacy score on second-order mutants compared to Enoiu et al.'s approach

Subject	#non-equivalent mutants	M-suite	E-suite
simTRIP	104	100	95.1
LAUNCHER	151	100	99.3

#modules	Module size [#blocks]	Test generation time [sec]
1	29	19.7
3	89	392.2
4	118	986.6
5	164	9777.2
7	224	30016.4
9	297	96098.5
10	328	133743.8
11	357	181331.3
13	437	325366.6

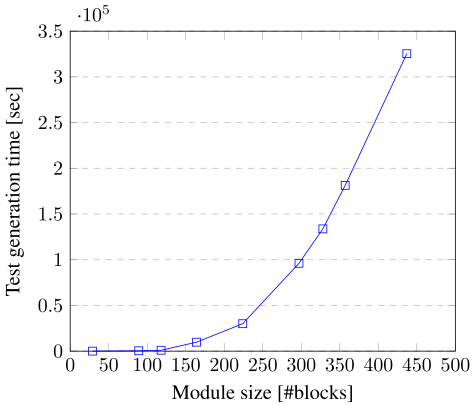

FIGURE 10 Test generation time over diverse module sizes

burdened the SMT solver to search for the solution, so the CCC-suite generation times for these modules had huge differences from other cases. Ultimately, structural coverage-based and mutation-based approaches are not absolutely competent. In addition, they can be combined to increase test effectiveness. We suggest that testers choose either one or both approaches in view of time cost.

Comparison with the mutation-based approach using model checking

The tool CompleteTest does not support counter blocks, so only simTRIP and LAUNCHER were used as subject programs among the three simple programs. It was found that Enoiu et al.'s approach cannot scale to industrial programs (BP system modules) because of state explosion. Because the UPPAAL model checker suffers from state explosion, the size of industrial programs is too large for the UPPAAL model checker to generate mutation-based test suites. To compare with Enoiu et al.'s approach, we modified MuFBDTester to generate mutation-based test sets by applying their proposed mutation operator set. The mutation-based test suite generated by Enoiu et al.'s approach was named E-suite. M-suite and E-suite were produced to kill the same mutant set.

Table 13 shows the test generation time and test suite sizes of M-suite and E-suite. Our approach reduced 40% of the time to generate four times more test input data compared to Enoiu et al.'s approach. We also investigated the effectiveness of the test suites. M-suite and E-suite were executed on second-order mutants generated from Jee et al.'s [21] mutation operator set. In Table 14, M-suite achieves higher mutation scores than E-suite. Because of the choices of base engines and generation strategies, our approach took less time to generate highly effective test suites than Enoiu et al.'s approach. However, this experiment might lack generality because of the small number of subjects.

5.3.4 | Discussion

Efficiency analysis with different factors

For the combinedTD module composed of over 400 blocks, MuFBDTester took approximately 4 days to generate mutation-adequate test suites. Our developed tool focuses on unit testing and the BP system is modularized, so we

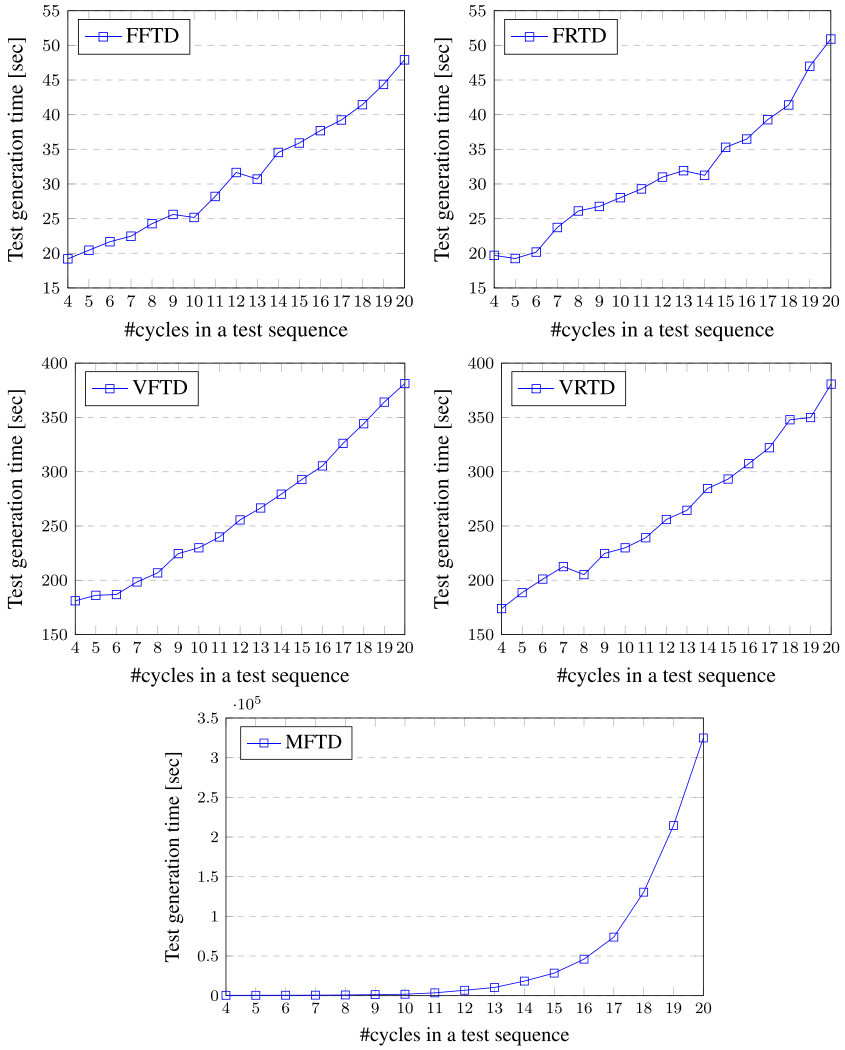

FIGURE 11 Test generation time over different cycle numbers in a test sequence (the minimum iteration number was 3)

preformed unit testing on each module. To investigate the efficiency of MuFBDTester more comprehensively, we tried to generate test suites over different module sizes. As the combinedTD module is composed of 13 modules, we tried to perform integration testing on the combinedTD module. Figure 10 shows the test generation time over different program sizes. As the module size increases, the test generation time grows nonlinearly.

The number of cycles in a test sequence also affects the efficiency of the testing tool. Thus, we measured the test generation time of the BP modules over different numbers of cycles for each test sequence. The number of cycles was constant for each test sequence in a test set. Figure 11 shows the test generation time over different cycle number settings. The results show that for all modules expect the MFTD module, the test generation grows linearly when the number of cycles in a test sequence increases. We expected that the BP programs would share similar results. However, the MFTD program had more feedback paths (inout variables) and inputs, which increased the complexity and burdened the SMT solver. Therefore, the number of feedback paths and inputs seemed to be critical factors for test generation time.

5.4 | Threats to validity

5.4.1 | Internal validity

It is difficult to find examples of real faults in studies of FBD programs, because most FBD programs are proprietary industry programs and are not made public for security reasons. We fortunately obtained real faults information for

the FBD programs from the output of the industrial project and reproduced them in this experiment. However, the number of real faults was very small, and experimental evaluation was conducted primarily using a large number of artificial faults (first- and second-order mutants). If many FBD programs could be made available and faults information could be shared, the testing research on FBD programs could be further advanced.

To evaluate MuFBDTester's performance, we showed its fault detection effectiveness and efficiency via experiments. However, there are many factors that can affect these evaluation aspects. For instance, the time efficiency is also dependent on the module size and complexity.

5.4.2 | External validity

The subject program selection is a threat to the external validity of this work. The number of subject programs was not large, and the subjects were not diverse. However, our subjects were industrial programs and closely related to NPP software. The FBD programs implemented in other types of power plants may include numerical functions, such as sine, cosine and logarithm, as defined in IEC 61131-3 [1]. In our approach, we used the Yices 1 SMT solver to generate test sequences. The Yices 1 SMT solver does not effectively support the test generation when the subject programs include complex numerical functions. Yices 1 was sufficient to support our subject programs because they did not include numerical functions. However, to process a wider range of FBD programs, it is necessary to be able to cope with nonlinear computation problems in test data generation. Utilizing multiple SMT solvers to cover numerical functions and nonlinear problems will form the basis of future studies.

6 | CONCLUSION

This paper presented an automated test sequence generation approach for FBD programs. We used mutation testing techniques to generate mutation-adequate test suites to address the problem of the test sets that meet the coverage criteria not ensuring the identification of specific types of errors. We implemented our approach with MuFBDTester, which generates a set of test sequences using Yices SMT solver.

We demonstrated the effectiveness and efficiency of the proposed approach. The test suites generated by MuFBDTester were able to detect not only all the artificial faults but also all the real faults that occurred in an industrial program. The mutation-based test suite generation time and test set size were usually between those of ICC-suite and CCC-suite of FBDTester 2.0. Overall, MuFBDTester took a reasonable amount of time to generate highly effective test sequences for complex programs.

In future work, we plan to utilize multiple SMT solvers to deal with FBD programs including complex numerical functions. Moreover, mutation operators can further be improved based on live higher-order mutants or by considering timing issues. For example, the CVR operator can also be applied to the preset time in timer block. We also plan to apply the proposed technique to industrial programs further.

ACKNOWLEDGEMENTS

This research was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2019R1I1A1A01062946), the Nuclear Safety Research Program through the Korea Foundation Of Nuclear Safety (KoFONS) using the financial resource granted by the Nuclear Safety and Security Commission (NSSC) of the Republic of Korea (No. 2105030), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2015-0-00250, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System) and the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2022-2020-0-01795) supervised by the IITP. [Correction added on 20 May 2022, after first online publication: the grant number has been corrected in the preceding sentence.]

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available on GitHub repository at https://github.com/Rienshaliu/MuFBDTester. Subject programs are shared with the exception of industry programs.

ORCID

Lingjun Liu https://orcid.org/0000-0001-6802-4090
Eunkyoung Jee https://orcid.org/0000-0003-0358-5369
Doo-Hwan Bae https://orcid.org/0000-0002-3152-5219

REFERENCES

1. IEC. International standard for programmable controllers: programming languages part 3, 1993. International Electrotecnical Commission.
2. Commission UNR. Software unit testing for digital computer software used in safety systems of nuclear power plants, 2013. Regulatory Guide, 1.
3. Guide IS. Software for computer based systems important to safety in nuclear power plants, 2000. Safety Standards Series NNS-G-11.
4. Lahtinen J. Automatic test set generation for function block based systems using model checking. In *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*, IEEE, 2014; 216–25.
5. Wu Y-C, Fan C-F. Automatic test case generation for structural testing of function block diagrams. Inf Softw Technol. 2014;56(10):1360–76.
6. Enoiu EP, Sundmark D, Pettersson P. Model-based test suite generation for function block diagrams using the UPPAAL model checker. In *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE: Luxembourg, 2013; 158–67.
7. Jee E, Shin D, Cha S, Lee J-S, Bae D-H. Automated test case generation for FBD programs implementing reactor protection system software. Softw Test, Verif Reliab. 2014;24(8):608–28.
8. Song J, Jee E, Bae D-H. FBDTester 2.0: automated test sequence generation for FBD programs with internal memory states. Sci Comput Program. 2018;163:115–37.
9. Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014. Association for Computing Machinery: New York, NY, USA, 2014; 435–45. https://doi.org/10.1145/2568225.2568271
10. Shin D, Jee E, Bae D-H. Comprehensive analysis of FBD test coverage criteria using mutants. Softw Syst Model. 2016;15(3):631–45.
11. Mathur AP, Wong WE. An empirical comparison of data flow and mutation-based test adequacy criteria. Softw Test, Verif Reliab. 1994;4(1):9–31.
12. Offutt AJ, Pan J, Tewary K, Zhang T. An experimental evaluation of data flow and mutation testing. Softw: Pract Exper. 1996;26(2):165–76.
13. Frankl PG, Weiss SN, Hu C. All-uses vs mutation testing: an experimental comparison of effectiveness. J Syst Softw. 1997;38(3):235–53.
14. Li N, Praphamontripong U, Offutt J. An experimental comparison of four unit test criteria: mutation, edge-pair, all-uses and prime path coverage. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, IEEE, 2009; 220–9.
15. Chekam TT, Papadakis M, Le Traon Y, Harman M. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017; 597–608.
16. Enoiu EP, Sundmark D, Čaušević A, Feldt R, Pettersson P. Mutation-based test generation for PLC embedded software using model checking. In *IFIP International Conference on Testing Software and Systems*, Springer, 2016; 155–71.
17. Mader A. 2000. A classification of PLC models and applications. In Discrete event systems Springer; 239–46.
18. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: help for the practicing programmer. Computer. 1978;11(4):34–41.
19. Offutt AJ. Investigations of the software testing coupling effect. ACM Trans Softw Eng Methodol (TOSEM). 1992;1(1):5–20.
20. DeMillo RA. Test adequacy and program mutation. In *ICSE*, 1989; 355–6.
21. Jee E, Song J, Bae D-H. Definition and application of mutation operator extensions for FBD programs. KIISE Trans Comput Pract. 2018;24:589–95.
22. Richter S, Wittig J-U. Verification and validation process for safety I&C systems. Nuclear Plant J. 2003;21(3):36–40.
23. Baresi L, Mauri M, Monti A, Pezzè M. PLCTools: design, formal validation, and code generation for programmable controllers. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 4, IEEE, 2000; 2437–42.
24. Lee SH, Lee SJ, Park J, Lee E, Kang HG. Development of simulation-based testing environment for safety-critical software, 2018. Nuclear Engineering and Technology.
25. McMinn P. Search-based software testing: past, present and future. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Computer Society: Berlin, Germany, 2011; 153–63.
26. Chen TY, Kuo F-C, Merkel RG, Tse TH. Adaptive random testing: the art of test case diversity. J Syst Softw. 2010;83(1):60–6.
27. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. IEEE Trans Softw Eng. 2012;38(2):278–92.
28. Staats M, Whalen MW, Heimdahl MPE. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11. ACM: Honolulu, HI, USA, 2011; 391–400.
29. Edvardsson J. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Systems Engineering*, ECSEL: Linköping, Sweden, 1999; 21–8.
30. Păsăreanu CS, Visser W. A survey of new trends in symbolic execution for software testing and analysis. Int J Softw Tools Technol Transf. 2009;11(4):339–53.
31. Bucur S, Ureche V, Zamfir C, Candea G. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11. ACM: Salzburg, Austria, 2011; 183–98. http://doi.acm.org/10.1145/1966445.1966463
32. Person S, Yang G, Rungta N, Khurshid S. Directed incremental symbolic execution. ACM SIGPLAN Notices. 2012;47(6):504–15.
33. Geldenhuys J, Dwyer MB, Visser W. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, ACM: Minneapolis, MN, USA, 2012; 166–76.
34. Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05. ACM: Chicago, IL, USA, 2005; 213–23. http://doi.acm.org/10.1145/1065010.1065036
35. Visser W, Pasareanu CS, Khurshid S. Test input generation with java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04. ACM Press: Boston, MA, USA, 2004; 97–107. http://dl.acm.org/citation.cfm?id=1007512.1007526
36. Beyer D, Chlipala AJ, Henzinger TA, Jhala R, Majumdar R. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04. IEEE Computer Society: Scotland, UK, 2004; 326–35. http://dl.acm.org/citation.cfm?id=998675.999437
37. Xie T, Marinov D, Schulte W, Notkin D. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 3440. Springer: Berlin Heidelberg, 2005; 365–81.

38. Vyatkin V. Software engineering in industrial automation: state-of-the-art review. IEEE Trans Ind Inform. 2013;9(3):1234–49.
39. Thramboulidis K, Frey G. An MDD process for IEC 61131-based industrial automation systems. In *Proceedings of the IEEE 16th Conference on Emerging Technologies Factory Automation (ETFA)*. IEEE: Toulouse, France, 2011; 1–8.
40. Enoiu EP, Čaušević A, Ostrand TJ, Weyuker EJ, Sundmark D, Pettersson P. Automated test generation using model checking: an industrial evaluation. Int J Softw Tools Technol Trans. 2014;2014:1–9.
41. da Silva LD, de Assis Barbosa LP, Gorgonio K, Perkusich A, Lima AMN. On the automatic generation of timed automata models from function block diagrams for safety instrumented systems. In *Proceedings of the 34th Annual Conference of IEEE Industrial Electronics*, IECON '08: Orlando, FL, USA, 2008; 291–6.
42. Kormann B, Vogel-Heuser B. Automated test case generation approach for PLC control software exception handling using fault injection. In *Proceedings of the 37th Annual Conference on IEEE Industrial Electronics Society*, IECON '11: Melbourne, Australia, 2011; 365–72.
43. Hametner R, Kormann B, Vogel-Heuser B, Winkler D, Zoitl A. Test case generation approach for industrial automation systems. In *Proceedings of the 5th International Conference on Automation, Robotics and Applications (ICARA)*, IEEE: Wellington, New Zealand, 2011; 57–62.
44. Magnus S, Krause J, Diedrich C. Test generation for model based fieldbus profiles. In *Proceedings of the 2012 IEEE International Conference on Industrial Technology (ICIT)*, IEEE: Athens, Greece, 2012; 682–7.
45. Hussain T, Frey G. UML-based development process for IEC 61499 with automatic test-case generation. In *Proceedings of the 2006 IEEE Conference on Emerging Technologies and Factory Automation*: Prague, Czech Republic, 2006; 1277–84.
46. Raymond P, Nicollin X, Halbwachs N, Weber D. Automatic testing of reactive systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98cb36279)*, 1998; 200–9.
47. Seljimi B, Parissis I. Automatic generation of test data generators for synchronous programs: Lutess v2. In *Workshop on Domain Specific Approaches to Software Test Automation: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ACM, 2007; 8–12.
48. Mohalik S, Gadkari AA, Yeolekar A, Shashidhar KC, Ramesh S. Automatic test case generation from Simulink/Stateflow models using model checking. Softw Testing, Verif Reliab. 2014;24(2):155–80.
49. Tekaya M, Bennani MT, Alagui MA, Ahmed SB. 2015. Aspect-oriented test case generation from Matlab/Simulink models. In Theory and engineering of complex systems and dependability Springer; 495–504.
50. Harman M, Jia Y, Langdon WB. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11. Association for Computing Machinery: New York, NY, USA, 2011; 212–22. https://doi.org/10.1145/2025113.2025144
51. Fraser G, Arcuri A. Achieving scalable mutation-based generation of whole test suites. Empirical Softw Engg. 2015;20(3):783–812.
52. Dadeau F, Ham P, Kheddam R. Mutation-based test generation from security protocols in HLPSL. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011; 240–8.
53. Papadakis M, Malevris N. Mutation based test case generation via a path selection strategy. Inf Softw Technol. 2012;54(9):915–32. http://www.sciencedirect.com/science/article/pii/S095058491200047X
54. Brillout A, He N, Mazzucchi M, Kroening D, Purandare M, Rümmer P, Weissenbacher G. Mutation-based test case generation for Simulink models. In *Formal methods for components and objects*, Springer, 2010; 208–27.
55. He N, Rümmer P, Kroening D. Test-case generation for embedded Simulink via formal concept analysis. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, IEEE, 2011; 224–9.
56. Kushigian B, Rawat A, Just R. Medusa: mutant equivalence detection using satisfiability analysis. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2019; 77–82.
57. Dutertre B, De Moura L. The Yices SMT solver. Tool paper at. 2006;2(2):1–2 http://yicescslsricom/tool-paperpdf
58. Marcos M, Estevez E, Perez F, Van Der Wal E. XML exchange of control programs. IEEE Ind Electron Mag. 2009;3(4):32–5.
59. Wong WE, Mathur AP. Reducing the cost of mutation testing: an empirical study. J Syst Softw. 1995;31(3):185–96.
60. Doosan Heavy Industry & Construction. KNICS.RPS.SDS231-01, Rev. 01, software design specification for the bistable processor of the reactor protection system, 2006.
61. KNICS-RPS-STR141 (Rev.00)—software test result for the bistable processor of the reactor protection system, Korea Atomic Energy Research Institute, 2006.
62. Jee E, Yoo J, Cha S, Bae D. A data flow-based structural testing technique for FBD programs. Inf Softw Technol. 2009;51(7):1131–9.
63. Yoo J, Cha S, Jee E. A verification framework for FBD based software in nuclear power plants. In *2008 15th Asia-Pacific Software Engineering Conference*, 2008; 385–92.
64. Cohen J. Statistical power analysis for the behavioral sciences. Routledge, 2013.

How to cite this article: Liu L, Jee E, Bae D-H. MuFBDTester: A mutation-based test sequence generator for FBD programs implementing nuclear power plant software. Softw Test Verif Reliab. 2022. e1815. https://doi.org/10.1002/stvr.1815

APPENDIX

TABLE A1 Size information of test suites in statistical comparisons (number of test sequences)

	FFTD				FRTD			
Run no.	M-suite	BC-suite	ICC-suite	CCC-suite	M-suite	BC-suite	ICC-suite	CCC-suite
1	79	74	85	85	79	66	81	86
2	89	78	81	84	89	70	81	84
3	84	64	79	84	88	70	72	82
4	76	80	82	85	73	76	83	81
5	80	80	85	84	82	66	82	83
6	81	88	83	84	81	86	83	83
7	82	74	84	86	83	68	82	85
8	87	80	84	86	87	72	78	86
9	80	68	77	86	79	66	82	84
10	80	88	83	83	79	80	82	84
Mean	81.8	77.4	82.3	84.7	82	72	80.6	83.8
	VFTD				VRTD			
Run no.	M-suite	BC-suite	ICC-suite	CCC-suite	M-suite	BC-suite	ICC-suite	CCC-suite
1	128	122	133	132	128	130	132	131
2	128	131	134	130	129	130	134	132
3	128	122	130	130	129	126	129	126
4	127	126	128	134	128	124	128	128
5	128	129	135	132	129	125	137	134
6	128	125	138	130	129	121	132	140
7	127	131	136	131	129	131	129	129
8	128	123	126	136	129	126	134	132
9	128	126	132	131	129	125	134	132
10	128	133	127	129	128	124	133	125
Mean	127.8	126.8	131.9	131.5	128.7	126.2	132.2	130.9

	MFTD			
Run no.	M-suite	BC-suite	ICC-suite	CCC-suite
1	132	134	124	125
2	132	137	130	125
3	132	139	130	124
4	132	136	133	130
5	132	133	126	126
6	132	135	130	128
7	132	135	129	133
8	132	139	127	127
9	132	135	132	126
10	132	132	128	126
Mean	132	135.5	128.9	127