KIPS Trans. Softw. and Data Eng.
Vol.10, No.4 pp.115~124
ISSN: 2287-5905 (Print), ISSN: 2734-0503 (Online)

https://doi.org/10.3745/KTSDE.2021.10.4.115

MuGenFBD: Automated Mutant Generator for Function Block Diagram Programs

Lingjun Liu† · Eunkyoung Jee†† · Doo-Hwan Bae†††

ABSTRACT

Since function block diagram (FBD) programs are widely used to implement safety-critical systems, effective testing for FBD programs has become important. Mutation testing, a fault-based testing, is highly effective in fault detection but computationally expensive. To support testers for FBD programs, we propose an automated mutant generator for FBD programs. We designed the MuGenFBD tool with the cost and equivalent mutant issues in consideration. We conducted experiments on real industrial examples to present the performance of MuGenFBD. The results show that MuGenFBD can generate mutants for FBD programs automatically with low probability of equivalent mutants and low cost. This tool can effectively support mutation analysis and mutation-adequate test generation for FBD programs.

Keywords : Mutant Generation, Mutation Analysis, Function Block Diagram, Software Testing

MuGenFBD: 기능 블록 다이어그램 프로그램에 대한 자동 뮤턴트 생성기

Lingjun Liu† · 지 은 경†† · 배 두 환†††

요 약

기능 블록 다이어그램(Function Block Diagram, FBD) 프로그램이 안전 필수 시스템 구현에 널리 사용되면서 FBD 프로그램에 대한 효과적인 테스트가 중요해졌다. 뮤테이션 테스팅은 오류 기반 테스팅 기술로, 오류 탐지에 매우 효과적이지만 비용이 많이 든다. 본 연구에서는 FBD 프로그램 테스터를 지원하기 위한, FBD 프로그램 대상 자동 뮤턴트 생성기를 제안한다. MuGenFBD 도구는 뮤턴트 생성 비용과 동등 뮤턴트 문제를 고려하여 설계되었다. MuGenFBD 도구의 성능을 평가하기 위해 실제 산업 사례에 대한 실험을 수행한 결과, MuGenFBD를 활용하여 뮤턴트 생성 시 동등 뮤턴트를 생성할 비율이 낮으며 적은 비용으로 FBD 프로그램 대상 뮤턴트를 효과적으로 자동 생성할 수 있음을 확인하였다. 제안하는 도구는 FBD 프로그램에 대한 뮤테이션 분석 및 뮤테이션 충분성 기준을 만족시키는 테스트 생성을 효과적으로 지원할 수 있다.

키워드 : 뮤턴트 생성, 뮤테이션 분석, 기능 블록 다이어그램, 소프트웨어 테스팅

1. Introduction

The testing for Programmatic Logic Controller (PLC) programs has become an important issue since the PLCs have been used to implement safety-critical systems, such as railway control systems and nuclear reactor protection systems. Function Block Diagram (FBD) is one of the standard PLC programming languages defined in IEC 61131-3 [1]. For instance, the reactor protection system in Korean Nuclear Instrumentation and Control System (KNICS) [2] was developed by implementing FBD programs. The system requires an extremely high quality. Thus, the effective testing of FBD programs is necessary.

The main goal of testing is to find faults in programs, but it is also required to evaluate test data. Mutation testing is an effective technique to measure fault detection capability of test data. Mutation testing is also a way to achieve the high quality required in critical software [3].

Jee et al. [4] presented a mutation operator set that contains 13 mutation operators for FBD programs.

※ 본 연구는 과학기술정보통신부 및 정보통신기획평가원의 대학ICT연구센터 지원사업(IITP-2021-2020-0-01795)과 2019년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. NRF-2019R1I1A1A01062946).
※ 이 논문은 한국정보처리학회 소프트웨어공학연구회가 주관한 2020 한국소프트웨어공학 학술대회(KCSE 2020)의 우수논문으로 "Automated mutant generation for function block diagram programs"의 제목으로 발표된 논문을 확장한 것임.
† 비 회 원 : 한국과학기술원 전산학부 석사과정
†† 정 회 원 : 한국과학기술원 전산학부 연구교수
††† 종신회원 : 한국과학기술원 전산학부 교수
Manuscript Received : August 24, 2020
First Revision : January 13, 2021
Accepted : February 4, 2021
* Corresponding Author : Eunkyoung Jee(ekjee@se.kaist.ac.kr)

This mutation operator set covers most of functions and function blocks defined in IEC 61131-3 [1]. Jee et al.'s work can be considered as the most comprehensive definition of mutation operator set for FBD programs. However, they manually applied the mutation operator set on small subject programs and did not automate the mutant generation.

We propose a tool called MuGenFBD to automatically generate mutants for FBD programs. We developed MuGenFBD based on the mutation operator set defined in [4]. Meanwhile, we also considered the cost of mutation testing and equivalent mutant raising issues. This tool can considerably ease the mutation analysis and the generation of mutation adequate test suite for FBD programs.

The remaining part of paper is organized as follows. Section 2 describes background knowledge and related works. Section 3 presents automated mutant generation for FBD programs. In Section 4, we apply our approach to the subject programs. We conclude this paper in section 5.

2. Background and Related Work

2.1 FBD Program

PLCs' operation is to repeatedly execute programs within a scan cycle. FBD is a graphical language for describing data flows through blocks [5]. FBD program consists of functions and/or function blocks. The key difference between functions and function blocks is related to internal memory of PLC. Functions yield outputs with the input values of current cycle. Thus, functions return same output values with same input values. When function blocks produce outputs, they consider not only the input values but also the data recorded in internal memory. Hence, function blocks may return different output results with same input parameters.

Functions can be categorized into 11 groups: Data type conversion, Numerical, Arithmetic, Bit shift, Bitwise Boolean, Selection, Comparison, Character string, Date and duration, Endianess conversion, and Validate. Function blocks can be categorized into 4 groups: Bistable, Edge detection, Counter, and Timer. There are total 21 different data types, including boolean, integer, real numbers, duration, etc.

Fig. 1 shows an example FBD program, which consists of *AND* function, *TON* (ON-delay Timer) function block, and CTU (Up Counter) function block. This example program is called simGRAVEL and used to control the amount of gravel transferred from a silo to a bin.

In the *TON* function block, the output *Q* value will be *true* if the input *IN* value has stayed *true* for a pulse time *(PT).* The output *ET* captures the elapsed time that the input *IN* value has been *true.* In Fig. 1, CTU_UP is assigned to *true* if the output value of *AND* block has been *true* for a certain time, which is *PULSE_TIME. TON_et* represents the duration that the output value of *AND* block has been *true.*

The *CTU* function block literally means Counting Up. It increments the counter when the input count *(CU)* is true. The output *(CV)* represents the current Counter Value. The output Q becomes *true* when the counter value *(CV)* reaches the count limit *(PV).* The input *R* represents Reset. The counter value is reset to zero when the input *R* is true. Thus, *BIN_EMPTY* denotes whether the bin is empty or not, so it is used to reset the bin level *(BIN_LEVEL).* When the output *CTU_Q* is *true,* the bin level reaches the setpoint *(SETPOINT).*

2.2 Mutation Testing

Mutation testing is a form of software fault-based testing. Different from general testing techniques, mutation testing is often used to measure the effectiveness of test data. In mutation testing, a basic assumption is that experienced programmers write correct or nearly correct programs [6]. The mutation

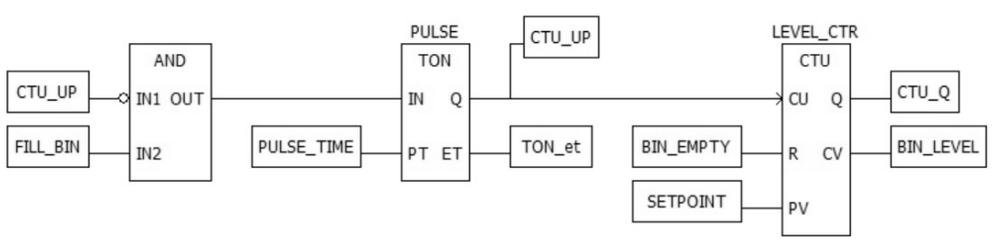


Fig. 1. An Example FBD Program

method is described as follows. A correct program P is given. Based on a set of mutation operators, mutated programs (or called mutants) are generated. The mutated programs are slightly different from the program P. To examine the test data T, the mutated programs are executed on T. If the result obtained from a mutated program differs from that from the program P on at least one test case, the mutated program is killed. This means the test data is capable of identifying the fault in the mutated program. If the test data T killed all the mutated programs, T is clearly adequate with regard to the set of mutant programs [7].

2.3 Related Work

Mutation testing is highly effective but computationally expensive, so it is required to have automated tools. Many mutation testing studies focus on developing mutation testing tools for specific programming languages. Mothra [8] is developed for Fortran language, and Proteum [9] is proposed for C language.

For FBD testing, some existing studies [10-12] evaluated effectiveness of test suites by mutation testing method. Shin et al. [10], Jee et al. [11], and Song et al. [12] conducted mutation analysis to investigate the fault detection capability of test suite that meet three structural coverage criteria. In Shin et al.'s work [10], 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). On the other hand, mutation testing can be used to generate mutation adequate test suites. Eniou et al. [13] proposed mutation-based test suite generation by model checking. They defined six mutation operators: Logic block Replacement Operator (LRO), Comparison Block Replacement Operator (CRO), Arithmetic Block Replacement Operator (ARO), Negation Insertion Operator (NIO), Value Replacement Operator (VRO), and Timer Block Replacement Operator (TRO). The difference between these two mutation operator sets is the additional TRO defined in Eniou et al.'s work.

Song et al. [12] utilized five mutation operators of [10] and four additional mutation operators for function block groups, i.e., Timer Block Replacement (TBR), Bistable Block Replacement (BBR), Edge detection Block Replacement (EBR), and Counter Block Replacement (CoBR), in their evaluation of coverage-based test suites for FBD programs.

Table 1. Mutation Operator Set for FBD Programs Defined in [4]

Operator Name	
CVR	Constant Value Replacement
IID	Inverter Insertion or Deletion
ABR	Arithmetic Block Replacement
CBR	Comparison Block Replacement
LBR	Logic Block Replacement
TBR	Timer Block Replacement
ConBR	Conversion Block Replacement
NBR	Numerical Block Replacement
SBR	Selection Block Replacement
BBR	Bistable Block Replacement
EBR	Edge detection Block Replacement
CouBR	Counter Block Replacement
SWI	SWitched Inputs

Jee et al. [4] defined 13 mutation operators by extending Shin et al.'s [10] and Song et al.'s [12] work. The extended mutation operator set, shown in Table 1, comprehensively covers functions and function blocks defined in IEC 61131-3 [1]. Also, considering the misplacement of inputs, this mutation operator set includes SWitched Inputs (SWI) operator. However, Jee et al. [4] did not automatically generate mutants from the defined mutation operator set. They manually seeded faults to generate mutants based on mutation operators, so the experiment was conducted on small-scale programs.

Our work focuses on the automation of generating mutants so that large-scale programs can be handled and a large number of mutants can be generated with little effort. We offer the mutation operator selection function so that users can choose what they want to generate for their purposes. We also considered equivalent mutant generation issues in automated mutant generation. In summary, the comparison of related studies and this work is presented in Table 2.

3. Mutant Generator for FBD Programs

3.1 Overall Process

The overall process of MuGenFBD is shown in Fig. 2.

Table 2. Comparison of Related Studies and Our Work

	Shin et al. (2012) [10]	Jee et al. (2014) [11]	Enoiu et al. (2016) [13]	Song et al. (2018) [12]	Jee et al. (2018) [4]	This work
Definition of new mutation operators	○	×	○	○	○	×
#mutation _operators defined or used	5	5	11 defined / 6 used	9	13	13
Automated mutant generation	○	○	○	○	×	○
Support for mutation operator selection	×	×	×	×	×	○
Equivalent mutant issue handling in mutant generation	×	×	×	×	×	○

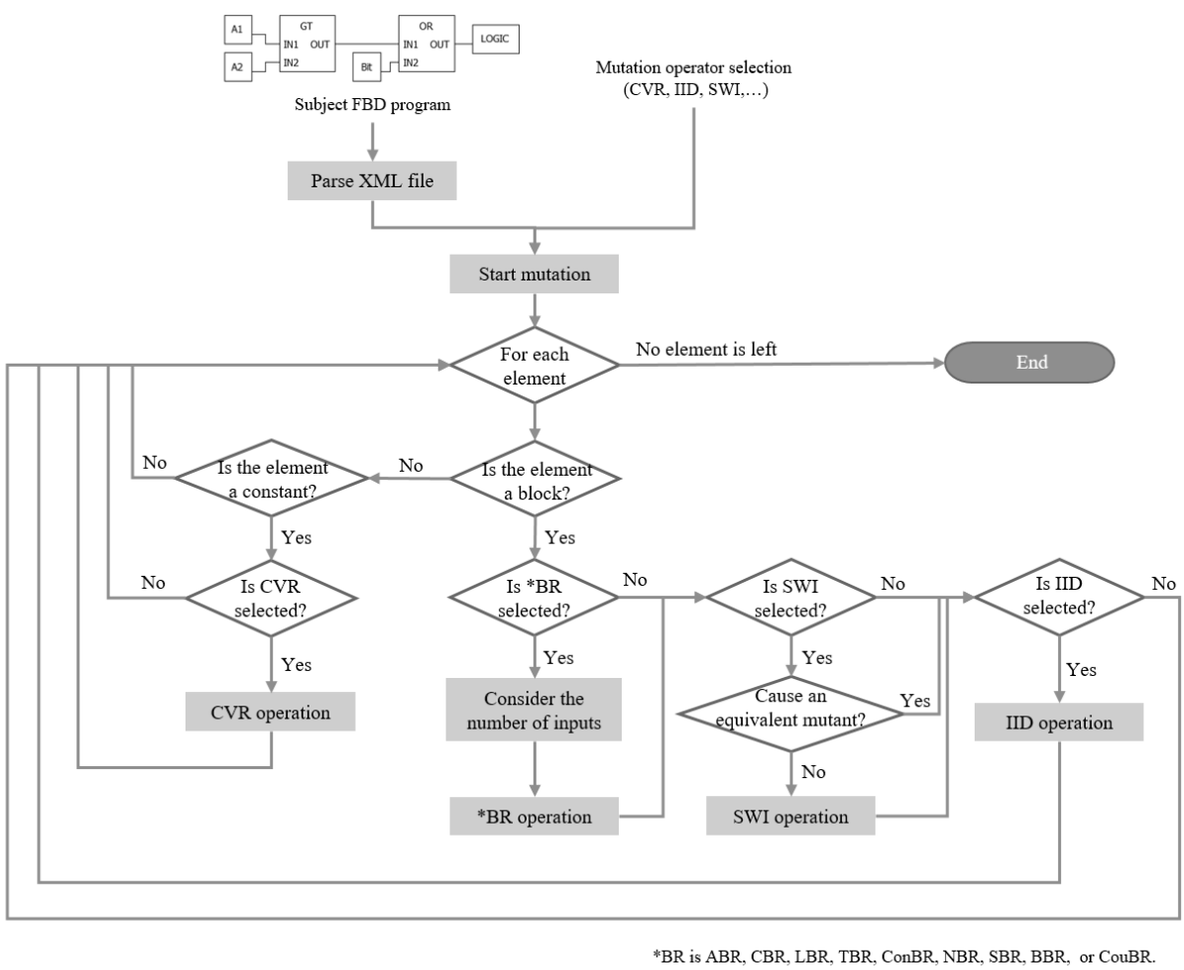


Fig. 2. Overall Process of MuGenFBD

MuGenFBD takes subject FBD programs in XML format and mutation operator selection as input. MuGenFBD parses the subject program XML file and extracts the structure of the program, including inputs, outputs, constants, and block elements. For each block, MuGenFBD applies the corresponding block replacement operator with number of inputs in consideration if the block replacement operator is selected; MuGenFBD applies the SWI operator with equivalent mutant issues in consideration if the SWI

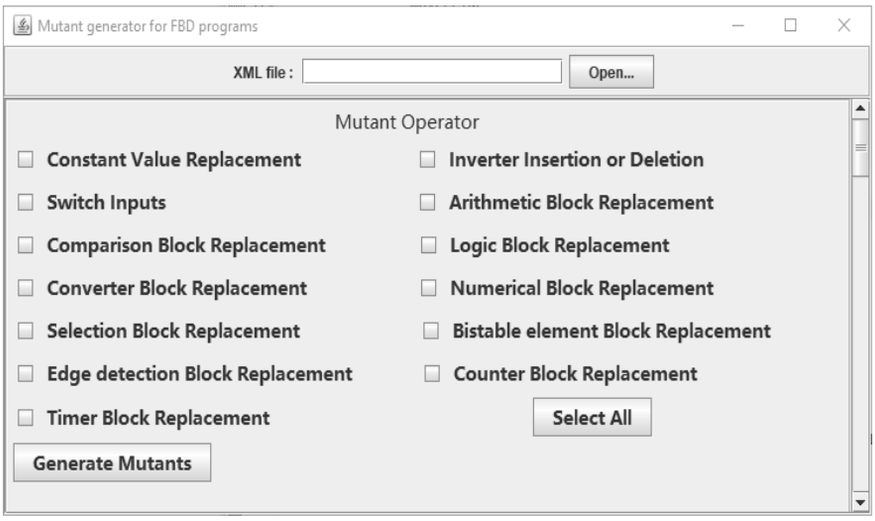


Fig. 3. A Screenshot of MuGenFBD

operator is selected; MuGenFBD applies the IID operator if the IID operator is selected. For each constant, MuGenFBD applies the CVR operator when the operator is selected.

As shown in Fig. 3, MuGenFBD provides an intuitive graphical user interface. Users can freely select desired mutation operators to generate mutants.

3.2 Issues of Block Replacement Operators

Among all functions defined in IEC 61131-3 [1], there are some extensible functions. The number of inputs in extensible functions can be increased. On the other hand, the number of inputs is fixed and cannot be extended in non-extensible functions. In the same group, there might be some extensible functions and some non-extensible functions.

Table 3 shows the definition of arithmetic functions in IEC 61131-3 [1]. When applying the ABR operator to an ADD block, if the number of inputs is three, it is restricted to replace the ADD function by the SUB function. Thus, when we designed the block replacement mutation operators, we also considered whether the block is extensible or not. Also, in the same block group, there might be some different numbers of input or output between blocks. For instance, the MOVE function only takes one input, but the other functions of the arithmetic group receive at least two inputs. Although the MOVE function belongs to the arithmetic group, we do not consider the MOVE function in applying the ABR operator.

Table 3. Arithmetic Functions [1]

Function name (description)	Explanation
Extensible arithmetic functions	
ADD (addition)	OUT:= IN1 + IN2 + ... + INn
MUL (multiplication)	OUT:= IN1 * IN2 * ... * INn
Non-extensible arithmetic functions	
SUB (subtraction)	OUT:= IN1 - IN2
DIV (division)	OUT:= IN1 / IN2
MOD (modulo)	OUT:= IN1 modulo IN2
EXPT (exponentiation)	OUT:= $IN1^{IN2}$
MOVE	OUT:= IN

3.3 Equivalent Mutant Raising Issues

An equivalent mutant is a mutant that is functionally equivalent to the original program. For mutation analysis, the mutation score is the ratio of the number of killed mutants over the total number of non-equivalent mutants [7]. Thus, the chance for generating equivalent mutants should be limited. SWI is literally to switch inputs in a block. However, we found this operator can possibly generate equivalent mutants. For instance, if we apply this mutation operator to the AND block, there's no influence on the logic (behavior). The order of input connections does not impact on the AND block operation. Hence, when applying the SWI operator, we carefully exclude commutative functions that produce equivalent mutants, including ADD (addition), MUL (multiplication), AND, OR, EQ (equal to), NE (not equal to) functions. The syntax condition is depicted below:

```
commutativeFunc = ["ADD", "MUL", "AND", "OR", "EQ", "NE"]
if functionName is in commutativeFunc
    skip
else
    do switch inputs
```

4. Empirical Evaluation

4.1 Subject Programs

We chose our subject programs from the KNICS project's BP system [2] implemented in FBD programs. The BP system, a part of the Reactor Protection System (RPS), determines whether a nuclear reactor must be stopped or not. Twenty modules included in the BP system can be categorized into six types: fix-falling trip decision (FFTD), fix-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. We selected one FBD program for each type of module in the BP system. Furthermore, the combinedTD module is developed by combining several modules in the BP system to test scalability of the proposed approach. Since some function block groups defined in IEC61131-3 [1] are not used in the BP system, we designed three more subject programs, which are simTRIP, simGRAVEL, and LAUNCHER. Table 4 shows the size information of each subject program.

4.2 Experiment

To demonstrate the performance of our tool, we applied our tool to subject programs. There are three aspects that we want to show the performance: (1) probability of producing equivalent mutants, (2) mutation operator selection, and (3) time efficiency.

1) Probability of Producing Equivalent Mutants

While selecting all the mutation operators, we executed MuGenFBD on all subject programs. Table 5 shows the probability of producing equivalent mutants for each subject program. In half of cases, no equivalent mutants were found. In average, there is only 1.3 percent of probability of generating equivalent mutants. We utilized the SMT solver to identify equivalent mutants from generated mutants. If the SMT solver cannot find a solution to distinguish the mutant from the original program, the mutant is considered as an equivalent mutant.

Table 4. Size Information of the Subject FBD Programs

Subject	#blocks	#inputs	#ouputs
simTRIP	3	3	2
simGRAVEL	3	3	4
LAUNCHER	4	2	1
FFTD	29	12	8
FRTD	29	12	8
VFTD	44	16	8
VRTD	44	17	8
MFTD	47	21	8
HB	19	6	1
combinedTD	437	221	92

Table 5. Probabilities of Generating Equivalent Mutants

Subject	#total mutants	Probability
simTRIP	14	0.000 (0/14)
simGRAVEL	8	0.000 (0/8)
LAUNCHER	13	0.077 (1/13)
FFTD	129	0.000 (0/129)
FRTD	129	0.000 (0/129)
VFTD	198	0.015 (3/198)
VRTD	199	0.015 (3/199)
MFTD	211	0.014 (3/211)
HB	114	0.000 (0/114)
combinedTD	1948	0.005 (9/1948)
Average	295	0.013

We found a few cases among equivalent mutants we found. Two examples are presented here. One is that the SWI operator generated equivalent mutants when the output of the SUB (subtraction) block is connected to the ABS (absolution) block, as depicted in Fig. 4. Since the absolute value is calculated by the output value of the SUB block, switching inputs of SUB blocks does not change the program functions. The other one is that the CBR operator generated equivalent mutants when the program has the logic to select an input value out of two before comparing two inputs, as depicted in Fig. 5. For

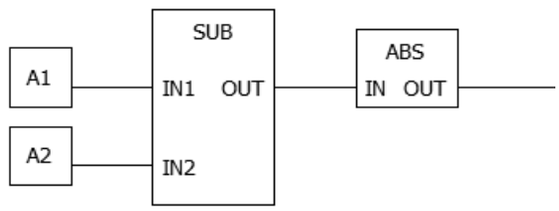


Fig. 4. Equivalent Mutant Case 1

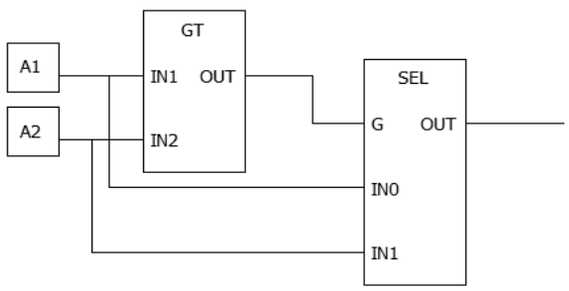


Fig. 5. Equivalent Mutant Case 2

instance, the GT (Greater-Than) function is replaced with the GE (Greater-than-or-Equal-to) function. Only if two inputs have the same value, this replacement can be identified. However, if two inputs have the same value, the output of selection would be not be influenced by the output of comparison. We have a plan to further elaborate our mutant generation algorithm to avoid potential equivalent mutants as much as possible.

2) Mutation Operator Selection

MuGenFBD can support users freely select their desired mutation operators. First, we selected the original mutation operator set: CVR, IID, ABR, LBR, and CBR [11]. Second, we selected all the implemented mutation operators. Table 6 shows number of mutants generated by the original mutation operator set and extended mutation operator set. In average, the extended mutation operator set generates over 44% more mutants than the original mutation operator set. Number of mutants generated by each mutation operator is presented in Table 7 and 8.

To investigate more aspects of the effect that mutation operator selection might bring, we conducted a mutation analysis on the existing test suites from Song et al.'s work [12]. Song et al. proposed a test sequence generation approach for FBD programs based on three coverage criteria. The three coverage criteria are Basic Converge (BC) criterion, Input Condition Coverage (ICC) criterion, and Complex Condition Coverage (CCC) criterion. Test suites that are guided by BC, ICC, and CCC criteria are named BC-suite, ICC-suite, and CCC-suite respectively.

We set up three kinds of mutation operator sets

Table 6. Number of Mutants Obtained by Original Mutation Operator Set [11] and Extended Mutation Operator Set [4]

Subject	#mutants generated by the mutation operator set of [11]	#mutants generated by the extended mutation operator set of [4]
simTRIP	11	14
simGRAVEL	5	8
LAUNCHER	10	13
FFTD	109	129
FRTD	109	129
VFTD	168	198
VRTD	168	199
MFTD	178	211
HB	100	114
combinedTD	1658	1948

Table 7. Number of Mutants Generated by Different Mutation Operators in Small Subject Programs

Mutation operator	simTRIP	simGRAVEL	LAUNCHER
IID	4	3	6
CBR	5	-	-
LBR	2	2	4
TBR	2	2	-
BBR	-	-	1
EBR	-	-	1
CouBR	-	1	-
SWI	1	-	1

Table 8. Number of Mutants Generated by Different Mutation Operators in Industrial Programs

Mutation operator	FFTD	FRTD	VFTD	VRTD	MFTD	HB	combined TD
CVR	-	-	8	8	-	16	24
IID	45	45	55	55	67	24	421
ABR	16	16	40	40	32	12	280
CBR	30	30	45	45	55	40	435
LBR	18	18	20	20	24	8	260
TBR	4	4	4	4	4	-	52
SWI	16	16	26	27	29	14	476

Table 9. Mutation Analysis Results with Three Different Mutation Operator Sets

Subject	BC-suite			ICC-suite			CCC-suite		
	MO-1	MO-2	MO-3	MO-1	MO-2	MO-3	MO-1	MO-2	MO-3
simTRIP	72.7 (8/11)	76.9 (10/13)	71.4 (10/14)	90.9 (10/11)	92.3 (12/13)	92.9 (13/14)	90.9 (10/11)	92.3 (12/13)	92.9 (13/14)
simGRAVEL	100.0 (5/5)	100.0 (8/8)	100.0 (8/8)	100.0 (5/5)	100.0 (8/8)	100.0 (8/8)	100.0 (5/5)	100.0 (8/8)	100.0 (8/8)
LAUNCHER	90.0 (9/10)	90.9 (10/11)	91.7 (11/12)	100.0 (10/10)	100.0 (11/11)	100.0 (12/12)	100.0 (10/10)	100.0 (11/11)	100.0 (12/12)
FFTD	94.5 (103/109)	94.7 (107/113)	95.3 (123/129)	96.3 (105/109)	94.5 (109/113)	96.1 (124/129)	97.2 (106/109)	97.3 (110/113)	97.7 (126/129)
FRTD	96.3 (105/109)	96.5 (109/113)	96.9 (125/129)	96.3 (105/109)	96.5 (109/113)	96.9 (125/129)	97.2 (106/109)	97.3 (110/113)	97.7 (126/129)
VFTD	98.2 (163/166)	98.2 (167/170)	98.5 (192/195)	98.8 (164/166)	98.8 (168/170)	99.0 (193/195)	99.4 (165/166)	99.4 (169/170)	99.5 (194/195)
VRTD	98.8 (164/166)	98.8 (168/170)	99.0 (194/196)	98.8 (164/166)	98.8 (168/170)	99.0 (194/196)	100.0 (166/166)	100.0 (170/170)	100.0 (196/196)
MFTD	96.6 (169/175)	96.6 (173/179)	97.1 (202/208)	97.1 (170/175)	97.2 (174/179)	97.6 (203/208)	96.6 (169/175)	96.6 (173/179)	97.1 (202/208)

named MO-1, MO-2, and MO-3. MO-1 denotes the original mutation operator set including CVR, IID, ABR, LBR, and CBR. To extend the targets of block replacement operators, MO-2 denotes the original mutation operator set plus seven more block replacement mutation operators, which are TBR, ConBR, NBR, SBR, BBR, EBR, and CouBR. Last, the new mutation type, which is the SWI operator, is added to MO-3, so MO-3 denotes the mutation operator set that includes all the mutation operators defined in [4]. Table 9 shows the mutation analysis results with three different mutation operator sets.

There is a tendency that mutation scores increase as the mutation operator set is set to MO-1 to 3 except for the BC-suite of simTRIP module. Furthermore, we found that, in most cases, the numbers of non-killed mutants stay the same for all three mutation operator sets. From this analysis, it is apparent that BC-suite, ICC-suite, and CCC-suite are effective to detect mutants generated from the plus four function block replacement operators and SWI operator. This mutation analysis results shows that the weakness of BC-suite, ICC-suite, and CCC-suite is on the mutants generated by the original mutation operator set. The mutation operator selection capability of MuGenFBD can contribute to a sophisticated analysis of test suites for FBD programs where strengths and weaknesses of the test suites can be revealed.

3) Time Efficiency

To present the efficiency, we selected all the mutation operators and executed MuGenFBD on the large scale program called combinedTD. For combinedTD, MuGenFBD took around three minutes to generate up to 1948 mutants. MuGenFBD is considered to provide practically usable performance.

4) Redundancy Issue

We observed some cases that two mutants are functionally equivalent to each other; thus, considered redundant. All test cases that can kill the mutant can also kill the other one, so one mutant is subsuming to the other one. Also, they are undistinguished. We manually analyzed mutants generated from small subject programs to check whether some mutants are functionally equivalent to each other. Total 5 mutants are found in the simTRIP program; 2 mutants are found in the simGRAVEL program. There is no case found in the LAUNCHER program.

First, the IID operator is applied on the same connection. Second, applying the IID operator on the output of GE function is functionally equivalent to replacing the GE function to the LT function. Third, applying the SWI operator on the GE function is functionally equivalent to replacing the GE function to the LE function. Last, applying the IID operator before the TON block is functionally

equivalent to replacing the TON block to TOF block. These kind of mutants are found, and this issue will be considered as our future work.

5. Conclusion

This study proposed an automated mutant generator for FBD programs. We designed and developed MuGenFBD by considering the cost and equivalent mutant issues. We evaluated our work with respect to probability of generating equivalent mutants and efficiency. MuGenFBD achieved significantly low chance for producing equivalent mutants by only 1.3 percentage. For large scale program, MuGenFBD generated 1948 mutants in around three minutes. According to results, MuGenFBD can ease the mutation analysis for FBD programs and contribute to the automated generation of mutation-based test suites for FBD programs. In addition, we conducted mutation analysis by utilizing existing test suites. The results showed that mutation operator selection capability could bring a sophisticated analysis for test suites. In future work, we plan to develop a tool which can automatically generate mutation-based test suite, with the help of MuGenFBD.

References

[1] International Electrotechnical Commission (IEC), "IEC61131-3: International Standard for Programmable Controllers - Part 3: Programming Languages," 2013.

[2] Doosan Heavy Industry & Construction, "Software design specification for the bistable processor of the reactor protection system," KNICS.RPS.SDS231-01, Rev.01, 2006. (In Korean)

[3] M. R. Woodward, "Mutation testing-its origin and evolution," *Information and Software Technology*, Vol.35, No.3, pp.163-169, 1993.

[4] E. Jee, J. Song, and D. H. Bae, "Definition and application of mutation operator extensions for FBD programs," *KIISE Transactions on Computing Practices*, Vol.24, No.11, pp.589-595, 2018. (in Korean)

[5] W. Bolton, "Programmable logic controllers," Newnes, 2015.

[6] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation Analysis," Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, 1979.

[7] R. A. Demillo, "Test Adequacy and Program Mutation," *Proceedings of the 11th International Conference on Software Engineering (ICSE)*, pp.355-356, 1989.

[8] R. A. Demillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King, "An extended overview of the Mothra software testing environment," *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, IEEE, pp.142-151, 1988.

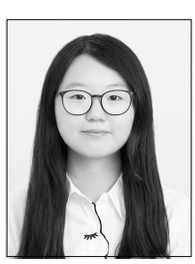
[9] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Proteum-a tool for the assessment of test adequacy for C programs user's guide," *Proceedings of Performability in Computing Systems*, Vol.96, pp.79-95, 1996.

[10] D. Shin, E. Jee, and D. H. Bae, "Empirical evaluation on FBD model-based test coverage criteria using mutation analysis," *International Conference on Model Driven Engineering Languages and Systems*, pp.465-479, 2012.

[11] E. Jee, D. Shin, S. Cha, J. S. Lee, and D. H. Bae, "Automated test case generation for FBD programs implementing reactor protection system software," *Software Testing, Verification and Reliability*, Vol.24, No.8, pp.608-628, 2014.

[12] J. Song, E. Jee, and D. H. Bae, "FBDTester 2.0: Automated test sequence generation for FBD programs with internal memory states," *Science of Computer Programming*, Vol.163, pp.115-137, 2018.

[13] E. P. Enoiu, D. Sundmark, A. Causevic, R. Feldt and P. Pettersson, "Mutation-based test generation for PLC embedded software using model checking," *Proceedings of the 28th International Conference on Testing Software and Systems, Lecture Notes in Computer Science*, Vol.9976, pp.155-171, 2016.



Lingjun Liu
https://orcid.org/0000-0001-6802-4090
e-mail : riensha@se.kaist.ac.kr
Lingjun Liu is a master student in School of Computing at Korea Advanced Institute of Science and Technology (KAIST). She received her B.S. degree in Computer Science from National Tsing Hua University. Her research interests include software testing and system-of-systems engineering.

Eunkyoung Jee

https://orcid.org/0000-0003-0358-5369
e-mail : ekjee@se.kaist.ac.kr

Eunkyoung Jee is a Research Associate Professor in School of Computing at KAIST. She was a Postdoctoral Researcher in the Computer and Information Science Department at the University of Pennsylvania. She received her B.S., M.S., and Ph.D. degrees in Computer Science from KAIST. Her research interest includes safety-critical software, software testing, formal verification, and safety analysis.

Doo-Hwan Bae

https://orcid.org/0000-0002-3152-5219
e-mail : bae@se.kaist.ac.kr

Doo-Hwan Bae is a Professor in School of Computing at Korea Advanced Institute of Science and Technology (KAIST). He received his Ph.D. at the Department of Computer Science in the University of Florida. He currently leads many projects funded by Korean government and industry. His research interests include software engineering for system of systems and software safety modeling and analysis.