Automated test case generation for FBD programs implementing reactor protection system software

Eunkyoung Jee¹, Donghwan Shin¹, Sungdeok Cha²,*,†, Jang-Soo Lee³ and Doo-Hwan Bae¹

¹KAIST, 291 Daehak-ro Yuseong-gu, Daejeon, Korea
²Korea University, Anam-dong Seongbuk-gu, Seoul, Korea
³Korea Atomic Energy Research Institute, 989-111 Daedeok-daero, Yuseong-gu, Daejeon, Korea

SUMMARY

Automated and effective testing for function block diagram (FBD) programs has become an important issue, as FBD is increasingly used in implementing safety-critical systems. This work describes an automated test case generation technique for FBD programs and its associated tool—FBDTester. Given an FBD program and desired test coverage criteria, FBDTester generates test requirements and invokes the Satisfiability Modulo Theories solver iteratively to derive a set of test cases. An industrial case study using reactor protection system software shows that the automatically generated test suites detected at least 82% of the known faults, whereas manually generated test cases only detected approximately 35%. Mutation analysis revealed that the automatically generated test suites substantially outperformed manually generated ones. Although test sequence generation requires some manual effort in the current FBDTester, it is apparent that the proposed approach significantly improves the efficiency and the reliability of FBD testing. Copyright © 2014 John Wiley & Sons, Ltd.

Received 17 December 2012; Revised 13 June 2014; Accepted 8 August 2014

KEY WORDS: automated test case generation; function block diagram; SMT solver; programmable logic controller

1. INTRODUCTION

Automated and effective testing for programmable logic controller (PLC) programs has become an important issue, as PLCs are now being widely used to implement safety-critical systems such as nuclear reactor protection systems. The function block diagram (FBD) is a commonly used PLC programming language among the five standard PLC programming languages defined by the International Electrotechnical Commission (IEC).

Industrial FBD testing relies mostly on functional testing in which test cases are manually derived from requirements expressed in natural language. Such test cases are often ineffective because they fail to accurately reflect the FBD program structure. Moreover, manual generation of test cases generally requires substantial time and effort.

This work presents an automated test case generation technique for FBD programs based on the structural test coverage criteria defined by Jee et al. [1]. Given a unit FBD program, test requirements expressed in logical formulas to achieve the chosen test coverage criteria are generated. The FBDTester uses the Yices SMT solver to generate test cases, maximally satisfying the test requirements.

*Correspondence to: Sungdeok Cha, Korea University, Anam-dong Seongbuk-gu, Seoul, Korea.
†E-mail: scha@korea.ac.kr

Copyright © 2014 John Wiley & Sons, Ltd.
A real-world industrial case study was conducted using trip (shutdown) modules of the bistable processor (BP) of reactor protection systems developed in the Korea Nuclear Instrumentation and Control System R&D Center project. Test cases were manually generated by FBD testing professionals, taking nearly three man-months to generate test cases for the BP system. In the case study, test cases automatically generated by the FBDTester are compared with respect to fault detection capability using real faults detected in development. In addition, mutation analysis was also performed to compare effectiveness against a large number of artificial errors (i.e., mutants).

The manually generated test cases only detected approximately 35% of the faults revealed in the preliminary version of the BP systems. The application of other techniques (e.g., model checking) detected the rest. Using the FBDTester, however, at least 82% of known faults in various modules were detected. In mutation analysis, the FBDTester also outperformed the manually generated test cases in almost all configurations. It is highly expected that the proposed approach could improve the efficiency and the reliability of FBD testing significantly.

This paper is organized as follows. Section 2 explains FBD test coverage criteria as background for the study and Section 3 presents a literature survey of the most relevant research. A test case generation technique and an automated tool for FBD programs are described in Section 4. Section 5 reports the results of the KNICS BP case study and presents an evaluation of the proposed approach. The paper is concluded in Section 6.

2. FBD TEST COVERAGE CRITERIA

2.1. FBD program

The main characteristic of the PLC programs is their indefinite and cyclic execution [2]. FBD, one of the standard PLC programming languages, is widely used because of its graphical notations and ease of developing applications with a high degree of data flow among the components.

The FBD is a data flow language based on viewing a system in terms of the flow of signals between processing elements [3]. A collection of blocks is wired together like a circuit diagram. Taken from the KNICS project, Figure 1 is used as an example throughout the paper. The output \( th_X_Trip \) represents a trip signal that would safely shut down a nuclear reactor when it is set to true. The output \( th_X_Trip \) becomes true when the processing value \( f_X \), representing pressure, temperature, and so on, is out of range \((k_X_{Min} \leq f_X \leq k_X_{Max})\), a module error occurs \((f_{Module_Error} = true)\), a channel error occurs \((f_{Channel_Error} = true)\), or a logical trip condition is met \((th_X_{Logic_Trip} = true)\). An inverter, denoted by a small circle and shown in the front end of the \( AND5\text{\_}out \) edge, can be attached to a Boolean input or output edge to negate the value.

The FBD programs consist of functions and/or function blocks. The main difference between functions and function blocks is that the former always produces the same result when called with the same input parameters as it has no memory. The latter has an internal data record and can, therefore, maintain status information [4]. In Figure 1, all the blocks such as \( GE\text{\_}INT \) (Greater than or Equal to), \( LE\text{\_}INT \) (Less than or Equal to), \( AND\text{\_}BOOL \), and \( OR\text{\_}BOOL \) are functions.

![Figure 1. An example function block diagram program for calculating \( th_X_Trip \).](image-url)
AUTOMATED TEST CASE GENERATION FOR FBD PROGRAMS

Figure 2. Timer on delay function block and its behavioral definition.

Figure 3. A simple function block diagram program including a timer function block.

Figure 2(a) shows an example of a function block, timer on delay (TON), that generates the Q output true when IN remains true during the delay time specified by variable PT because IN changed from false to true. ET captures the elapsed time of an internal timer. The behavioral definition of a timer such as TON is described by a timing diagram, as shown in Figure 2(b). It shows how the outputs Q and ET vary in response to different IN values, as time, labeled ‘t’, passes from left to right.

The TON has two internal variables, preIN and preET, denoting the values of IN and ET stored in the previous scan cycle, respectively. They are shown as implicit edges in Figure 2(a). Figure 3 shows an FBD program including a timer function block TON.

2.2. FBD test coverage criteria

This section briefly explains the primary concepts of the FBD coverage criteria, defined by Jee et al. [1], relevant to this work. An FBD program is considered a directed graph that consists of multiple inputs and outputs. The FBD program shown in Figure 1 has seven inputs and one output. Unique names (e.g., GE3_out, L E4_out, etc.) are given to internal edges. A d-path is a finite sequence of edges in an FBD program. The attached d-prefix distinguishes it from the general path in control flow graphs. The length of a d-path (i.e., the number of edges included in it) is guaranteed to be finite because FBD programs have no internal feedback loops.

The d-path condition (DPC) of a d-path p is the condition along the d-path of an FBD program under which the input edge value plays a role in computing the output edge. The d-path condition of a d-path p, DPC(p), is defined by a conjunction of the function condition, FC(ei, ei+1), for each function and function block condition, FBC(ej, ej+1), for each function block along the d-path.

Intuitively, FC(ei, e0) is the condition under which the value at the input edge ei has an influence on the value at the output edge e0 through a single function. When the data types of the input and the output edges are same, FC(ei, e0) is defined as the condition under which the value at the ei flows into the output edge e0. If the output edge has a different data type from the input edge, the input edge’s influence on the output edge is defined in a different way. Jee et al. defined FCs and FBCs formally [1].

If a function has n inputs, there are n FCs for each input to the output. For example, assume that AND has two inputs, IN1 and IN2. If eiIN1 is true, the same value flows into the output only if the other input eiIN2 is also true. If eiIN1 is false, the output is false regardless of other values. A formal definition of FCs for the AND block with two inputs is the following: if p1 = (eiIN1, eOUT) ∧ p2 = (eiIN2, eOUT) ∧ eOUT = AND(eIN1, eIN2),

\[ FC(p_1) = \text{if } e_{IN1} \text{ then } e_{IN2} \text{ else true} \]
\[ = \neg e_{IN1} \lor e_{IN2} \]
\[ FC(p_2) = \neg e_{IN2} \lor e_{IN1} \]

\( FBC((e_i,e_o)) \) has a similar concept to \( FC((e_i,e_o)) \), except \( e_i \) and \( e_o \) are connected by a single function block. Whereas FC definitions are relatively simple, FBCs are more complex due to internal variables, which are considered implicit edges of the corresponding function block.

The FBCs for the output \( Q \) of TON are evaluated as follows: if \( p_1 = \langle e_{IN}, e_Q \rangle \land p_2 = \langle e_{PT}, e_Q \rangle \land e_Q = \text{TON\_Q}(e_{IN}, e_{PT}) \),
\[ FBC(p_1) = \text{if } e_{IN} \text{ then } (i_{preIN} \land (i_{preET} \geq e_{PT})) \text{ else true} \]
\[ = \neg e_{IN} \lor (i_{preIN} \land (i_{preET} \geq e_{PT})) \]
\[ FBC(p_2) = (i_{preET} > 0) \]

For the \( FBC(p_1) \), when \( e_{IN} \) is true, the output \( e_Q \) is also true only if \( i_{preIN} \land (i_{preET} \geq e_{PT}) \). If \( e_{IN} \) is false, the output \( e_Q \) is false without any constraints.

The process of deriving the DPC is similar to the one used in backward symbolic execution. Because there are four functions along the d-path \( p_{5.1} = \langle f_X, GE3\_out, AND5\_out, OR6\_out, th\_X\_Trip \rangle \) in Figure 1, \( DPC(p_{5.1}) \) is the conjunction of four FCs as follows:
\[ DPC(p_{5.1}) = DPC(\langle f_X, GE3\_out, AND5\_out, OR6\_out, th\_X\_Trip \rangle) \]
\[ = FC(\langle f_X, GE3\_out \rangle) \land FC(\langle GE3\_out, AND5\_out \rangle) \]
\[ \land FC(\langle AND5\_out, OR6\_out \rangle) \land FC(\langle OR6\_out, th\_X\_Trip \rangle) \] (1)

Starting from the output edge of the given d-path, each FC or FBC is expanded. When backward symbolic computation is completed, a DPC becomes an expression containing only input and internal variables.

Three different test coverage criteria for FBD programs have been defined based on the definition of DPC: basic coverage (BC), input condition coverage (ICC), and complex condition coverage (CCC). Let \( DP \) denote the set of all d-paths from input edges to output edges.

\textbf{Definition 1 (Basic Coverage)}
A set of test data \( T \) satisfies the basic coverage criterion if and only if \( \forall p \in DP \ \exists t \in T \ | DPC(p)|_t = \text{true} \).

The BC criterion focuses on covering every d-path in an FBD program under test at least once. Test requirements for BC are DPCs for all the d-paths of the target program. A test case \( t \) is ‘meaningful’ if the input given to the d-path \( p \) has an influence in determining the output of \( p \). Such a condition is captured by \( |DPC(p)|_t = \text{true} \) in the aforementioned definition. Otherwise (e.g., \( |DPC(p)|_t = \text{false} \), \( t \) is unable to make the input of \( p \) influence the output. Such a test case is surely ineffective in covering the d-path, and it fails to contribute to meeting the coverage requirement. Although BC is straightforward in concept, it is often ineffective in detecting logical errors that FBD programs might have.

\textbf{Definition 2 (Input Condition Coverage)}
A set of test data \( T \) satisfies the input condition coverage criterion if and only if it satisfies the basic coverage criterion and for any d-path \( p \in DP \) with a Boolean input edge \( \text{in}(p) \), \( \exists t \in T \ | \text{in}(p) \land DPC(p)|_t = \text{true} \) and \( \exists t' \in T \ | \neg \text{in}(p) \land DPC(p)|_{t'} = \text{true} \).

To satisfy the ICC, it is no longer sufficient to choose an arbitrary value on the input edge whose values would influence the outcome (e.g., \( DPC(p_{3.1}) = DPC(\langle f\_Module\_Error, OR6\_out, OR7\_out \rangle) \)). A set of test data must now be chosen such that the input values include both \text{true} and \text{false} for Boolean inputs (e.g., \( DPC(p_{3.1}) \land f\_Module\_Error \) as well as \( DPC(p_{3.1}) \land \neg f\_Module\_Error \)).
**Definition 3 (Complex Condition Coverage)**
A set of test data \( T \) satisfies the complex condition coverage criterion if and only if it satisfies the basic coverage criterion and for any \( d \)-path \( p \in DP \) of length \( n \) with any Boolean edge \( e_i \) where \( 1 \leq i \leq n \), \( \exists t \in T \ | e_i \land DPC(p) \rangle_t = \text{true} \) and \( \exists t' \in T \ | \lnot e_i \land DPC(p) \rangle_{t'} = \text{true} \).

The stronger CCC criterion requires that every Boolean edge’s variation in the \( d \)-path be tested at least once under the satisfied DPC. Every test set satisfying the ICC criterion also satisfies the BC criterion. Similarly, the CCC criterion subsumes both the ICC and the BC criteria.

### 3. RELATED WORK

There are several tools that perform unit testing of FBD programs. A simulation-based validation tool named SIVAT [5] generates ANSI C code from an FBD program and performs functional testing based on the code. An integrated tool environment named PLCTOOLS [6] supports the entire development process of PLC programs, including specification, transformation, and simulation. PLCTOOLS transforms FBD programs into high-level timed Petri nets models and performs simulation-based functional testing based on the high-level timed Petri nets models. Unfortunately, such approaches support neither structural testing of FBD programs nor automated test case generation.

Test case generation for programs implemented in procedural languages is a mature research topic [7–11], and the use of symbolic execution in test case generation has also been extensively studied [12–19]. These approaches cannot be applied directly to FBD programs because their target languages and purposes differ.

As model-driven development is popular in the industrial automation domain [20, 21], several model-based test case generation approaches for PLC programs have been proposed [22–27]. Enoiu et al. [22] proposed model-based test suite generation for FBD programs. The authors transform FBD programs into timed automata models and automatically generate test suites from the timed automata models using the UPPAAL model checker. Test suites can be generated according to structural coverage criteria such as function coverage, decision coverage, or condition coverage. 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. Although the capability to generate test suites for test requirements expressed in CTL formulas is beneficial, the formulation of test requirements requires manual effort and some level of expertise.

Silva et al. [23] generated timed automata models from FBD programs to test whether the execution traces of the implementation conform to the traces of the specification model, using the UPPAAL TRON tool. Kormann and Vogel-Heuser proposed an automated test case generation approach for PLC control software exception handling using fault injection [24]. Their primary goal was not to generate test cases that achieve specific coverage criteria. Rather, they injected faults into Unified Modeling Language (UML) state charts and generated test cases that could detect exception handling errors of the target PLC programs.

Hametner et al. [25] presented an automated test case generation approach for industrial automation applications described by UML diagrams. IEC 61499 basic function blocks including the generated test suite could be generated. Test cases are generated only when test specifications are manually described. Magnus et al. [26] introduced a model-based procedure of automatic test generation for fieldbus profile specification. Their tool transfers a UML state machine, capturing dynamic specifications of function blocks, into a special Petri net dialect and generates test cases with the round-trip path coverage criterion. Hussain and Frey addressed an IEC 61499 compliant UML-based development process with automatic test case generation [27]. UML state diagrams and activity diagrams were used to generate test cases. The round-trip path coverage criterion was used for the state diagrams.

All model-driven techniques share the common aspect that they generate test cases from models with respect to specific goals such as coverage achievement and test requirements. Unlike other approaches [22–27], the proposed technique works directly on FBD programs without relying on...
intermediate models (e.g., UML, timed automata, or special Petri net dialect models). In addition to intuitive and straightforward coverage analysis, this work provides thorough analysis of the fault-detection effectiveness of generated test suites, whereas none of the aforementioned studies do.

There have been studies on test case generation for data flow languages similar to FBD. Parissis et al. proposed an automatic test generation approach for Lustre/SCADE programs [28–30]. Lutess is a black-box testing tool that generates test input sequences from environmental descriptions randomly or using directives such as operational profiles and behavioral patterns. Whereas generated test cases are guaranteed to satisfy a set of specified environmental properties, Lutess is unable to generate test cases with respect to structural coverage criteria.

GATeL is another test case generation tool for the Lustre language [31]. Test sequence generation from test objectives expressed in Lustre is handled by solving a constraint system involving program data flows. The capability to generate test sequences conforming to test objectives is beneficial. However, when generating test sequences considering the internal structure of the target Lustre program, GATeL works only when the path predicates are manually annotated.

Although structural testing on FBD programs is frequently mandated when used in safety-critical systems, no well-established guidelines for this procedure have been developed. A definition of coverage criteria that accurately reflects the features that are unique to FBD is a meaningful start [1]. This paper describes how the FBDTester extends the FBDTestMeasurer [32] by integrating automated test case generation functionality as well as test coverage assessment functionality.

4. AUTOMATED TEST DATA GENERATION FOR FBD PROGRAMS

Test data generation is the process of identifying program inputs that satisfy the selected testing criterion. Test requirements to achieve the BC, ICC, and CCC criteria for FBD programs are represented by propositionally complex formula including binary-valued functions of non-binary variables. Generating test data for FBD programs is considered a problem of finding solutions satisfying those test requirements (i.e., Satisfiability Modulo Theories (SMT) problems). The FBDTester uses Yices [33], an SMT solver developed by SRI International, because of its ability to solve weighted MAX-SMT problems and compute unsatisfiable cores. The current tool implementation is based on using the Yices 1.‡

4.1. Single test case generation

Figure 4 shows an overview of a single test case generation procedure. Given an FBD program under test and test coverage criteria as inputs, a test case is generated. Iterative execution of the same procedure with unsatisfied test requirements yields additional test cases.

Similar to usual path-oriented test case generation, the test case generation procedure for FBD programs consists of three major parts: (1) analyzing the program under test; (2) extracting test requirements; and (3) solving test requirements. In the first part, target program information required for test case generation, such as variables, blocks’ operations, FCs, FBCs and d-paths, are analyzed. After calculating DPCs for all the d-paths in the FBD program, test requirements are generated according to the selected test coverage criteria in the second part. Using this information, a Yices input file is created and executed by the Yices SMT solver.

Figure 5 is an initial Yices input file to generate test cases satisfying BC and ICC for the FBD program shown in Figure 1. The Yices input file generation rules are explained using this example.

Environmental setting (lines 2 and 3) Commands pertinent to our application of Yices are inserted here. The (set-evidence! true) command enables the construction of evidence such as models and unsatisfiable cores, which are important sources of test case generation. In order to display the assertion IDs that are necessary to distinguish each test requirement (i.e. an assertion) from others, the verbosity level is assigned the number 2 or higher.

‡When we mention Yices, it refers to Yices 1 throughout the paper.
Declaration of constants and variables (lines 8 through 13) All variables and constants included in the target FBD program are declared here.

Declaration of all the FBD blocks (lines 17 through 22) FBD blocks used in the program as well as dependencies are captured. For example, as shown in line 19, block (5) performs an AND operation of outputs of blocks (3) and (4). It then negates its own output so that it will be used as an input to block (6).

Definitions of FCs and FBCs (lines 26 through 33) It is efficient to have macro expressions specifying FCs and FBCs because an FC or FBC is usually used several times in the computation of test requirements. There are eight macro definitions used in FCs and FBCs for the FBD program shown in Figure 1. Macro definitions allow compact specifications of DPCs.

Definitions of DPCs for all the d-paths (lines 37 through 43) There are seven d-paths in the FBD program under test. DPC definitions are represented by the conjunction of FCs or FBCs. DPCs are essential parts of test requirements. Because they are usually used many times, it is efficient to have macro expressions (e.g., DPCp3_1 and DPCp3_2).

Assertions of test requirements (lines 48 through 61) The assert+ command captures a test requirement. It can be used to compute unsatisfiable cores and max-sat when weights are provided. Technical details are outside the scope of this paper, and readers should consult the Yices tool paper [33] for additional information. All the test requirements to be satisfied for achieving the selected test coverage criteria are inserted here with the assert+ command. The weight value indicates the priority within the strategy. A larger value (e.g., 8) is assigned when a weaker coverage criterion (e.g., BC) is selected. Similarly, a smaller value (e.g., 4) is assigned when a stronger criterion (e.g., ICC) is applied. The weights can be adjusted according to different strategies. The choice of weights will be discussed further in Section 4.3.2. There are 13 test requirements in the example (e.g., six for BC and seven for ICC, respectively).

Execution of max-sat (line 64) The max-sat command is issued to extract a maximal satisfying model. If the execution result is “sat,” a maximal satisfying model is generated, and it becomes a test case of the target FBD program.

4.2. Iterative test case generation

Figure 6 shows an iterative test case generation procedure. The default value of \( n \) is 1, and Yices begins to work on \( UTR_{1,1} \), which contains the initial set of unsatisfied test requirements.
Figure 5. Yices input file for generating test cases satisfying the basic coverage and input condition coverage criteria for the function block diagram program in Figure 1.
Given \( n \) (default \( n=1 \)), \( \{ \text{all TRs} \} = \text{TR}_1 \cup \text{TR}_2 \cup \ldots \cup \text{TR}_n \)
for each \( i \) (\( 1 \leq i \leq n \)), execute the following procedure

1. **1st iteration**
   - Analyze the program
   - Extract/Assert UTR\(_{i-1}\)
   - Solve TRs (max-sat)

   \[ \text{UTR}_{i-1} = \text{TR}_i \]

   \[ \text{TestSet}_i = \{ \text{TC}_{i,1} \} \]

2. **2nd iteration**
   - Analyze the program
   - Extract/Assert UTR\(_{i-2}\)
   - Solve TRs (max-sat)

   \[ \text{UTR}_{i-2} = \text{UTR}_{i-1} - \{ \text{TRs satisfied by TC}_{i,1} \} \]

   \[ \text{TestSet}_i = \{ \text{TC}_{i,1}, \text{TC}_{i,2} \} \]

   \[ \vdots \]

   \[ \vdots \]

   \[ \text{m}^{\text{th}} \text{ iteration} \]
   - Analyze the program
   - Extract/Assert UTR\(_{m-1}\)
   - Solve TRs (max-sat)

   \[ \text{UTR}_{m-1} = \text{UTR}_{m-1} - \{ \text{TRs satisfied by TC}_{i,m-1} \} \]

   \[ \text{TestSet}_i = \{ \text{TC}_{i,1}, \text{TC}_{i,2}, \ldots, \text{TC}_{i,m} \} \]

   \[ \text{UTR}_{i,m+1} = \text{UTR}_{i,m} - \{ \text{TRs satisfied by TC}_{i,m} \} \]

   \[ \text{TestSet}_i = \text{TestSet}_1 \cup \text{TestSet}_2 \cup \ldots \cup \text{TestSet}_n \]

   *TR: test requirement, TC: test case, UTR: set of unsatisfied TRs

   Figure 6. Iterative test case generation procedure.

Yices returns “sat” as the result when there is a solution making the test requirements maximally satisfiable.

Figure 7 illustrates how Yices works as a key component in test case generation in the FBDTester. It is initially invoked with the file named thXTrip.ys and shown in Figure 5. Initially, UTR\(_{1,1}\) has 13 test requirements (i.e., 13 assertions) with respect to BC and ICC criteria. Assertions are represented by their ids. For the first input file, Yices displays the result “sat” which means that there is a solution making the test requirements maximally satisfiable. The solution \( f_{\text{Module\_Error}} = \text{false}, f_{\text{Channel\_Error}} = \text{false}, \text{th\_X\_Logic\_Trip} = \text{false}, f_X = 2 \) becomes the test case TC\(_{1,1}\). It satisfies all the test requirements except three (i.e., assertions 8, 10, and 12), and another input file containing UTR\(_{1,2}\) is generated. UTR\(_{1,2}\) is essentially identical to UTR\(_{1,1}\) except that the satisfied test requirements are deleted, and unsatisfied assertions are sequentially renamed (e.g., 8 vs 1, 10 vs 2, etc.). The next iteration would generate a test case TC\(_{1,2}\) \( f_{\text{Module\_Error}} = \text{true}, f_{\text{Channel\_Error}} = \text{true}, \text{th\_X\_Logic\_Trip} = \text{true}, f_X = 2 \). Because all test requirements have been satisfied, the iteration does not need to be repeated. Should Yices return “unsat” as its result, the iteration must stop with incomplete satisfaction of the chosen test coverage. This means that the remaining assertions are infeasible test requirements.

Issuing the “(max-sat)” command to Yices with \( n = 1 \) is likely to deliver the minimum number of test cases. However, a minimal set does not necessarily mean that it is the most effective set. A test case that satisfies the maximum number of test requirements may fail to detect bugs in the
Figure 7. An example of the iterative test case generation.

FBD program. Furthermore, when performing “max-sat” computations on many test requirements, the execution time can increase exponentially. Sometimes, it makes practical sense to divide test requirements into \( n \) disjoint sets (e.g., \( TR_1 \) through \( TR_n \)) to reduce “max-sat” computation time and obtain more test cases.

Algorithm 1 describes the procedure of test case generation for FBD programs using Yices.

The FBDTester shown in Figure 8 provides an intuitive graphical interface. Users may select desired coverage criteria and output variables for which test cases are to be generated. The FBDTester generates test cases that maximally achieve the chosen coverage criteria and stores test cases in a text file. The d-path finder functionality visually highlights a specific d-path a user wants to analyze in the graphical view.

4.3. Test data generation issues

The generation of a test oracle is especially important in test automation. A test oracle is often obtained by specifying the expected output for each test case according to the requirements or design specifications. In this case study, expected outputs were computed by implementing functional requirements in Java. In FBD testing automation, however, there are several additional issues one must resolve.

4.3.1. Internal memory states. FBD programs consist of functions and function blocks. If an FBD program under test consists of only functions, test requirement formulas contain only input variables. However, when the target FBD program includes function blocks, test requirement formulas contain internal variables as well as input variables. Figure 3 shows an FBD program including a TON timer function block. The \( TRIP\_LOGIC\_out \) is set to true when the processing value \( PV\_OUT \) exceeds the set-point \( TSP \) constantly for the given time delay, \( K\_DELAY \).

There are four d-paths for the primary output \( TRIP\_LOGIC\_out \). One of the d-paths is \( p_{4,1} = \langle PV\_OUT, GE1\_out, AND2\_out, TRIP\_LOGIC\_out \rangle \) and \( DPC(p_{4,1}) \) is defined as follows:

\[
DPC(p_{4,1}) = (\neg GE1\_out \lor \neg TRIP\_LOGIC) \land (\neg AND2\_out \lor (pre\_AND2\_out \land (pre\_TON1\_et \geq K\_DELAY)))
\]
Algorithm 1 Test suite generation for FBD programs

\[ n \leftarrow \text{a splitting parameter} \]
\[ UTR = \{ \text{all of the test requirements to be satisfied} \} \]
\[ UTR_1, \ldots, UTR_n = \text{Split}(UTR, n) \quad \triangleright \text{Split } UTR \text{ into } n \text{ disjoint sets from } UTR_1 \text{ to } UTR_n \]
\[ i \leftarrow 1 \]
\[ \text{while } i \leq n \text{ do} \]
\[ k \leftarrow 1 \]
\[ TestSet_i \leftarrow \emptyset \]
\[ \text{while } UTR_i \neq \emptyset \text{ do} \]
\[ k^{th} \text{ input file } \leftarrow \text{Generate}(\text{FBD program}, UTR_i, k) \quad \triangleright \text{Generate } k^{th} \text{ input file by inserting an assertion for every } tr \in UTR_i \]
\[ \text{result } = \text{Execute}(\text{Yices.exe}, k^{th} \text{ input file}) \]
\[ \text{if result is } \text{‘sat’ then} \]
\[ TestSet_i \leftarrow TestSet_i \cup \{ \text{a model generated from the Yices execution} \} \]
\[ UTR_i \leftarrow UTR_i - \{ \text{satisfied assertions of test requirements} \} \]
\[ \text{else} \]
\[ \text{break} \quad \triangleright \text{Break test case generation for } UTR_i \]
\[ \text{end if} \]
\[ k \leftarrow k + 1 \]
\[ \text{end while} \]
\[ i \leftarrow i + 1 \]
\[ \text{end while} \]
\[ TestSet \leftarrow TestSet_1 \cup TestSet_2 \cup \ldots \cup TestSet_n \]
\[ \text{return } TestSet \]

Figure 8. Screenshot of the function block diagram (FBD) tester, a case tool for FBD test coverage assessment and test case generation.
In the DPC Equation (2), \(\text{pre}_\text{AND}_2\_\text{out}\) and \(\text{pre}_\text{TON}_1\_\text{et}\) are internal variables denoting the value \(\text{AND}_2\_\text{out}\) and \(\text{TON}_1\_\text{et}\) stored in the previous scan cycle, respectively. A combination of internal variables represents the internal memory states.

Satisfying test requirements accompanied by internal memory states requires sequences of test cases. One way to generate preambles (i.e., test data reaching specific internal memory states) is to use constraint solving techniques implemented in most SMT solvers. If cyclic execution can be accurately represented by including computations required of internal state variables, the SMT solver can automatically generate a sequence of test cases. In the case of the FBD program shown in Figure 3, when the scan cycle is 50 ms, \(K_{\text{DELAY}}\) has a constant value of 100 ms, and \(T \text{SP}_P\) has a constant value of 26805, a test sequence, \((\text{PV}_\text{OUT}_{t3} = 0) \rightarrow (\text{PV}_\text{OUT}_{t2} = 26805) \rightarrow (\text{PV}_\text{OUT}_{t1} = 26805) \rightarrow (\text{PV}_\text{OUT}_{t0} = 0)\), is automatically generated by executing Yices with the input file whose snippet is shown in Figure 9. In this example, the \(\text{TRIP}_\text{LOGIC}\) variable has a value of the \(\text{TRIP}_\text{LOGIC}_\text{out}\) of the previous cycle; thus, \(\text{TRIP}_\text{LOGIC}\) is not included in the input vector. This single test sequence covers all of the four test requirements with respect to the BC criterion, resulting in achieving 100% of basic coverage.

In Figure 9, the suffix ‘t1’ and ‘t2’ denote values observed at one and two cycles ahead of the current cycle, respectively. The value of \(\text{TRIP}_\text{LOGIC}\) in the current cycle is influenced by \(\text{TRIP}_\text{LOGIC}_\text{out}_t1\), which captures the value of the output variable \(\text{TRIP}_\text{LOGIC}_\text{out}\) observed in the previous cycle. Similarly, \(\text{TRIP}_\text{LOGIC}_t1\) is also defined by \(\text{TRIP}_\text{LOGIC}_\text{out}_t2\). The definition of \(\text{TON}_1\_\text{et}_t1\) includes \(\text{TON}_1\_\text{et}_t2\), its previous value, inside.

If computations to be repeated during each cyclic execution of the FBD program can be explicitly captured, the SMT solver can generate a sequence of test cases automatically. However, automatically determining the number of cycles to be repeated is an issue that remains unsolved. In Figure 9, information on executions to be repeated was manually inserted. Current implementation of the FBDTester supports fully automated test suite generation for the FBD program without function blocks. When it comes to function blocks, it supports automated test sequence generation only partially.

### 4.3.2. Choice of weights.

The choice of weights may indirectly affect the performance of the max-sat computation. As mentioned earlier, a larger weight value is usually assigned when weaker coverage criterion is chosen. In order to investigate how max-sat computation performs on different weight values, three different weight settings - \(W_1 = (1, 1, 1)\), \(W_2 = (8, 4, 2)\), and \(W_3 = (10, 8, 6)\) - were compared. They indicate the weight values assigned to test requirements for BC, ICC, and CCC criteria, respectively. \(W_1\), equal weights, indicates no priority among them. \(W_2\) and \(W_3\), on the other hand, represent two different but similar settings. In the evaluation, the variable-rate-falling trip decision (VFTD) module was used, and the experiment was repeated three times with the average execution time compared with avoid potential bias in the runtime environment. The VFTD module is complex enough in that there are 26614 test requirements to solve when the CCC criterion is applied. As expected, the average execution for \(W_1\) was the longest with 1514 s.
4.3.3. Infeasible test requirements. It may be impossible to generate a set of test cases achieving 100% coverage for any coverage criterion because some test requirements may turn out to be infeasible. Existence of infeasible test requirements does not necessarily mean that FBD program is incorrect. They are sometimes caused by an awkward programming structure, but they can also be caused by the inevitable logical structure of the target program. Not all infeasible test requirements are related to errors, but in many cases, infeasible test requirements indicate that the corresponding program structure needs to be improved.

5. EVALUATION

5.1. Experimental design

In order to evaluate the effectiveness and efficiency of the FBDTester, a preliminary FBD design of the BP system from the KNICS project was used. The BP system, classified as safety-critical by a regulatory agency, performs a core task to determine if a nuclear reactor must be stopped. It is a complex system in that its software design specification has about 190 pages. FBD implementation of the BP contains more than 1000 blocks, and about 1000 variables are used. BP testing is quite a daunting task in that the unit test report for the BP [34] consisted of 139 pages and had more than 300 test cases. Two experienced FBD engineers spent about 6 weeks together documenting the FBD testing plan and manually deriving test cases.

In order to provide a platform-independent testing environment, the FBDTester receives FBD programs in the PLCopen standard XML format. Unfortunately, not all PLC engineering tools support this interface via XML. The BP system was implemented using a system engineering tool called pSET, and it stores data in a proprietary format. Therefore, an input conversion tool was separately implemented.

The FBD program for BP consists of 18 modules implementing trip logics and two heart-beat (HB) monitoring modules. These 20 modules can be categorized into six types: fix-rising trip decision (FRTD), fix-falling trip decision (FFTD), variable-rate-rising trip decision (VRTD), VFTD, manual-reset-falling trip decision (MFTD), and HB. Modules belonging to the same type share similar logical structures. For example, four modules of the FFTD type initiate safe shutdown when processing values such as pressure and temperature inside the reactor remain below the predefined threshold value over a certain duration. In the experiment, a module from each type was included. Care was taken not to include only certain types or the simplest modules in size and complexity. Table I shows the size information on the target FBD modules. In the case of the most complex module, MFTD, it had as many as 51 blocks and more than 60000 CCC test requirements to satisfy.

<table>
<thead>
<tr>
<th></th>
<th>HB</th>
<th>FFTD</th>
<th>FRTD</th>
<th>VFTD</th>
<th>VRTD</th>
<th>MFTD</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>#blocks</td>
<td>18</td>
<td>33</td>
<td>33</td>
<td>48</td>
<td>48</td>
<td>51</td>
<td>38.5</td>
</tr>
<tr>
<td>#inputs</td>
<td>6</td>
<td>15</td>
<td>15</td>
<td>19</td>
<td>19</td>
<td>24</td>
<td>16.3</td>
</tr>
<tr>
<td>#outputs</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3.5</td>
</tr>
<tr>
<td>BC TRs</td>
<td>58</td>
<td>156</td>
<td>156</td>
<td>2086</td>
<td>2086</td>
<td>4176</td>
<td>1453</td>
</tr>
<tr>
<td>ICC TRs</td>
<td>58</td>
<td>208</td>
<td>208</td>
<td>2276</td>
<td>2276</td>
<td>6126</td>
<td>1859</td>
</tr>
<tr>
<td>CCC TRs</td>
<td>578</td>
<td>1396</td>
<td>1396</td>
<td>26614</td>
<td>23870</td>
<td>61362</td>
<td>19202</td>
</tr>
</tbody>
</table>

HB, heart beat; FFTD, fix-falling trip decision; FRTD, fix-rising trip decision; VFTD, variable-rate-falling trip decision; VRTD, variable-rate-rising trip decision; MFTD, manual-reset-falling trip decision; BC, basic coverage; ICC, input condition coverage; CCC, complex condition coverage.
In the experiment, several sets of test cases were generated by the FBDTester for each module according to the chosen coverage criteria. One must note that different test suites satisfying the same coverage level for the same module may differ in their fault detection capabilities. In order to reduce potential bias, the FBDTester was repeatedly invoked five times for each module and each coverage criterion. When dealing with a large number of test requirements, it is often practical to split them into multiple files. In order to investigate the impact of splitting factor on the FBDTester’s performance and effectiveness, the experiment was repeated with splitting factors of 1, 5, and 10. Therefore, the FBDTester was applied on a total of 270 test suites (6 module types × 3 coverage criteria × 3 splitting factors × 5 iterations each).

The experiment was conducted in two phases. First, the fault detection capabilities of the test cases were compared using the real faults detected in the KNICS project. That is, the test cases automatically generated by the FBDTester were used to determine how many of them could have been detected. Likewise, the test cases manually generated by domain experts were used as well. In the second phase, mutation analysis was applied to measure how effective the FBDTester is against various types of potential faults. That is, artificial but realistic faults were injected into FBD programs, and the FBDTester’s capability to detect them was measured. For example, the FRTD module had no fault detected in the design. But the 138 potential faults were injected into the module during mutation analysis. Information on real faults, reported by Jee et al. [35], and mutants is shown in Table II.

If a mutant is detected by a test case, the test case kills the mutant. The mutant score, mScore, is defined as the ratio of the number of killed mutants over the total number of mutants. Five mutation operators defined by Shin et al. [36] were applied: constant value replacement (CVR), inverter insertion or deletion (IID), arithmetic block replacement (ABR), logic block replacement (LBR), or comparison block replacement (CBR). They are sufficiently representative of faults often found in other FBD programs. For example, FBD designers often make the mistake of using the less than or equal to block rather than the less than block. Mutants on the CBR would cover such a possibility. Mutants were generated by applying the mutation operators to each block or each edge of the model whenever possible.

5.2. Fault detection effectiveness

5.2.1. Detection of known faults. A total of 10 types of distinct errors on the target software were found using the model checking technique [35]. One example was a misuse of a relational function. The GE (greater than or equal to) block was used instead of the greater than block. Because these two FBD blocks are quite similar, developers often mistakenly use one for the other. Although it seems to be quite a small error, this error could have led to a dangerous situation in which a trip does not occur when it should.

Table III shows the fault detection rate of the automatically and the manually generated test suites. The number of detected faults over the total known faults in each module is presented with the detection rate (%) in parentheses. For example, in the FFTD module, all the automatically generated test suites could detect both of known faults, whereas the manual test suite could detect none of them. Manually generated test cases detected only 35% of the known faults on average in our study. The application of other techniques, including model checking and safety analysis, fortunately detected the rest of faults in the project, but the experiment shows that the project’s manual test suite was not sufficiently effective to be able to detect those faults.
Table III. Detection rate of known faults.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Parameter</th>
<th>HB</th>
<th>FFTD</th>
<th>FRTD</th>
<th>VFTD</th>
<th>VRTD</th>
<th>MFTD</th>
<th>%average</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>$n=1$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>3.4/4</td>
<td>2.8/4</td>
<td>2.2/4</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>$n=5$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>4/4</td>
<td>4/4</td>
<td>3.2/4</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>$n=10$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>4/4</td>
<td>4/4</td>
<td>3.2/4</td>
<td>96</td>
</tr>
<tr>
<td>ICC</td>
<td>$n=1$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>2.8/4</td>
<td>2.8/4</td>
<td>2.8/4</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>$n=5$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>3.8/4</td>
<td>4/4</td>
<td>3/4</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>$n=10$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>4/4</td>
<td>4/4</td>
<td>3.2/4</td>
<td>96</td>
</tr>
<tr>
<td>CCC</td>
<td>$n=1$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>3.8/4</td>
<td>3.6/4</td>
<td>3/4</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>$n=5$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>4/4</td>
<td>4/4</td>
<td>3.2/4</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>$n=10$</td>
<td>1/1</td>
<td>2/2</td>
<td>—</td>
<td>4/4</td>
<td>4/4</td>
<td>4/4</td>
<td>100</td>
</tr>
<tr>
<td>manual</td>
<td>—</td>
<td>1/1</td>
<td>0/2</td>
<td>—</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>35</td>
</tr>
</tbody>
</table>

HB, heart beat; FFTD, fix-falling trip decision; FRTD, fix-rising trip decision; VFTD, variable-rate-falling trip decision; VRTD, variable-rate-rising trip decision; MFTD, manual-reset-falling trip decision.

Table IV. Detection rate of mutants (mutant score).

<table>
<thead>
<tr>
<th>Objective</th>
<th>Parameter</th>
<th>HB</th>
<th>FFTD</th>
<th>FRTD</th>
<th>VFTD</th>
<th>VRTD</th>
<th>MFTD</th>
<th>%average</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>$n=1$</td>
<td>57.3</td>
<td><strong>63.9</strong></td>
<td><strong>64.8</strong></td>
<td>88.9</td>
<td>87.9</td>
<td><strong>75.5</strong></td>
<td>73.0</td>
</tr>
<tr>
<td></td>
<td>$n=5$</td>
<td>62.7</td>
<td>90.0</td>
<td>84.2</td>
<td>95.1</td>
<td>95.0</td>
<td><strong>82.2</strong></td>
<td>84.9</td>
</tr>
<tr>
<td></td>
<td>$n=10$</td>
<td>61.1</td>
<td>91.6</td>
<td>91.3</td>
<td>96.8</td>
<td>96.8</td>
<td>84.2</td>
<td>87.0</td>
</tr>
<tr>
<td>ICC</td>
<td>$n=1$</td>
<td>61.3</td>
<td><strong>72.8</strong></td>
<td><strong>65.8</strong></td>
<td>88.2</td>
<td>88.5</td>
<td><strong>71.5</strong></td>
<td>74.6</td>
</tr>
<tr>
<td></td>
<td>$n=5$</td>
<td>66.3</td>
<td>85.5</td>
<td>88.3</td>
<td>96.0</td>
<td>95.3</td>
<td><strong>82.5</strong></td>
<td>85.6</td>
</tr>
<tr>
<td></td>
<td>$n=10$</td>
<td>61.5</td>
<td>93.3</td>
<td>91.5</td>
<td>96.0</td>
<td>96.2</td>
<td>84.7</td>
<td>87.2</td>
</tr>
<tr>
<td>CCC</td>
<td>$n=1$</td>
<td>74.7</td>
<td>87.5</td>
<td>88.0</td>
<td>93.5</td>
<td>92.3</td>
<td><strong>79.6</strong></td>
<td>85.9</td>
</tr>
<tr>
<td></td>
<td>$n=5$</td>
<td>77.9</td>
<td>96.5</td>
<td>97.7</td>
<td>97.9</td>
<td>98.1</td>
<td>84.3</td>
<td>92.0</td>
</tr>
<tr>
<td></td>
<td>$n=10$</td>
<td>81.1</td>
<td>99.4</td>
<td>99.4</td>
<td>98.2</td>
<td>97.9</td>
<td>86.0</td>
<td>93.6</td>
</tr>
<tr>
<td>manual</td>
<td>—</td>
<td>14.6</td>
<td>82.6</td>
<td>82.6</td>
<td>84.7</td>
<td>79.1</td>
<td>83.3</td>
<td>71.2</td>
</tr>
</tbody>
</table>

HB, heart beat; FFTD, fix-falling trip decision; FRTD, fix-rising trip decision; VFTD, variable-rate-falling trip decision; VRTD, variable-rate-rising trip decision; MFTD, manual-reset-falling trip decision.

Using the FBDTester, however, at least 82% of all of the known errors could be detected using the automatically generated test cases. As shown in Table III, every automatically generated test suite for satisfying BC, ICC, and CCC with $n=1$, 5, and 10 could detect more faults than the manually generated test suite. In the best cases, the FBDTester generated test cases that can detect all of the known errors. Note that the fault detection rates of the BC, ICC, and CCC test suites are averaged over five independent test suites. It is clear that the FBDTester with BC, ICC, and CCC objectives generates effective test cases in terms of detecting known faults. Let a $C$-suite denote a test suite satisfying $C$ coverage criterion. BC-suites detected a similar number of known faults as the ICC-suites, regardless of $n$. However, CCC-suites had a higher fault detection rate than BC-suites and ICC-suites for each $n=1$, 5, and 10, as expected.

5.2.2. Detection of mutants. Table IV shows the mutant detection rate of each test suite. In the case of $n=1$, results, as depicted in Figure 10, indicate that the automatically generated test cases detected on average 73% of mutants when the BC criterion was applied. Fault detection rates
Figure 10. mScore of test suites with respect to the basic coverage, input condition coverage, and complex condition coverage criteria.

Figure 11. mScore of test suites with respect to the complex condition coverage criterion when \( n = 1, 5, \) and 10.

ranged from 57% for the HB module to 89% for the VFTD module. When ICC and CCC criteria were applied, the average detection rates were 75% and 86%, respectively. The fault detection rate, mScore, generally increases as stronger coverage criteria are used. However, it must be emphasized that the subsumption relationship does not guarantee better fault detection effectiveness, as proved in [37]. In the VFTD and MFTD modules, the mScore associated with BC was higher than the corresponding value associated with ICC.

Splitting test requirements tends to increase the test suite size, thereby contributing to increased effectiveness. See Table IV and Figure 11 for a comparison of the effectiveness of test cases in splitting test requirements. As depicted in Figure 11, the error detection rate for the CCC criterion increased from 86% to 94% on average by splitting alone. In four of the six modules, the mScore value was 98% or higher when \( n = 10. \)

Compared to other modules, the mScore for the HB module was rather low. Detailed analysis revealed that the design of the HB module included more arithmetic blocks than other modules did. In the previous study [36], the proposed test coverage criteria were shown to be more effective in detecting faults related to Boolean logic and rather weak in detecting faults on arithmetic operations.

Table IV confirms that the mScore of the manual test suite was as good as the BC-suites or ICC-suites in some modules. An mScore lower than that of the manual test suite is presented in bold cases. An mScore lower than that of the manual test suite is presented in bold cases. For example, BC-suites with \( n = 1 \) killed less mutants than the manual test suite in the FFTD, FRTD, and MFTD modules. However, as a stronger coverage criterion is applied or a larger value of \( n, \) the splitting parameter, is used, the automatically generated test suites definitely had higher mutation scores than the manually generated ones.

In order to analyze the characteristics of the automatically generated and the manually generated test suites, killed mutants were classified into four distinct sets as follows:

- A-only: a set of mutants killed by automatically generated test suites only
- M-only: a set of mutants killed by manually generated test suites only
- Both: a set of mutants killed by both automatically and manually generated test suites
- None: a set of mutants killed by none of the test suites

If a mutant is killed by one or more CCC-suites but not by the manual test suites, the mutant is classified as A-only. Table V shows the distribution of the four sets of killed mutants. Among all the killed ABR mutants, 8.16% were A-only, whereas 4.08% were M-only. In the case of CVR mutants, 56.25% of those killed belonged to A-only while none were M-only. This clearly shows...
that automatically generated test suites are more effective than manual test suites in detecting constant value faults. On average, 18.90% of killed mutants were A-only and 3.93% were M-only. The automatically generated test suites could detect 90.70% of mutants, whereas the manually generated test suites could detect 75.72% on average.

5.3. Efficiency

Another important concern of automated test case generation is the number of the generated test cases and the test suite generation time. Such metrics determine the scalability and practicality of the proposed approach. Table VI shows the results. Most test cases were generated in less than a minute or two, and the average number of test cases generated did not exceed a couple of dozen. In the extreme case, when working on the MFTD module, the FBDTester took approximately seven hours (or 25311 s) on a PC equipped with an i7 2.8 GHz CPU and 12 GB memory to generate about 49 test cases when attempting to solve 61362 test requirements without splitting them. Even though the MFTD module may appear simple enough to contain only 51 blocks and process 24 inputs, complex dependency relations make the task of manual test case generation not only impractical but also potentially error-prone.

Test suite generation time consists of two parts: (1) Yices input file generation where a considerable portion comprises test requirement generation, and (2) constraint solving time. Table VII shows the test suite generation time for each subjective module with respect to the CCC criterion. The portion of constraint solving time over the total test suite generation time is shown in the rightmost column. Constraint solving time by Yices takes approximately 85% to 95% of the whole test suite generation time, whereas input file generation including test requirement generation takes 5% to 15% of the total time. The test suite generation time largely depends on the constraint solving strategy.

Execution time was generally reduced when the test requirements were split into five or ten subsets, as shown in Table VI, because the max-sat problems Yices had to solve were often easier and took less time.

The current implementation of the FBDTester only supports splitting test requirements into $n$ groups, each of which includes a similar number of test requirements. Different splitting strategies may result in different efficiencies. For example, putting conflicting test requirements such as $cond_1 \land cond_2$ and $cond_1 \land \neg cond_2$ in different groups would be better than putting these into one group.

When comparing the number of test cases, there are 10 to 20 test cases for each FBD module in the manual test suites. The FBDTester also generates less than 20 test cases for BC and ICC with $n = 1$ while it generates much more test cases for CCC with $n = 5$ and 10.

The FBD program used in the study consisted of only functions, and they were not subject to intentional modification. Therefore, it was unnecessary to find preambles. The time taken to investigate which errors were found was negligible because the experiment only dealt with known errors.

---

Table V. Classification of killed mutants into four sets (%).

<table>
<thead>
<tr>
<th>Mutant type</th>
<th>A-only</th>
<th>Both</th>
<th>M-only</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABR</td>
<td>8.16</td>
<td>84.69</td>
<td>4.08</td>
<td>3.06</td>
</tr>
<tr>
<td>CBR</td>
<td>26.44</td>
<td>58.64</td>
<td>5.08</td>
<td>9.83</td>
</tr>
<tr>
<td>CVR</td>
<td>56.25</td>
<td>32.14</td>
<td>0.00</td>
<td>11.61</td>
</tr>
<tr>
<td>IID</td>
<td>6.43</td>
<td>89.39</td>
<td>4.18</td>
<td>0.00</td>
</tr>
<tr>
<td>LBR</td>
<td>11.54</td>
<td>80.77</td>
<td>3.85</td>
<td>7.69</td>
</tr>
<tr>
<td>Total</td>
<td>18.90</td>
<td>71.80</td>
<td>3.93</td>
<td>5.37</td>
</tr>
</tbody>
</table>

ABR, arithmetic block replacement; CBR, comparison block replacement; CVR, constant value replacement; IID, inverter insertion or deletion; LBR, logic block replacement.

---

§Time in Table VII is for the first iteration in the FBDTester, not the total test suite generation time.
Table VI. Test suite size / test suite generation time (sec).

<table>
<thead>
<tr>
<th>Objective</th>
<th>n</th>
<th>HB</th>
<th>FFTD</th>
<th>FRTD</th>
<th>VFTD</th>
<th>VRTD</th>
<th>MFTD</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>n = 1</td>
<td>5.2 / 0.114</td>
<td>5.0 / 0.184</td>
<td>5.0 / 0.215</td>
<td>16.1 / 94.99</td>
<td>16.2 / 99.62</td>
<td>20.2 / 507.9</td>
<td>11.3 / 117.1</td>
</tr>
<tr>
<td></td>
<td>n = 5</td>
<td>7.4 / 0.229</td>
<td>17.4 / 0.335</td>
<td>17.0 / 0.319</td>
<td>67.0 / 81.13</td>
<td>66.2 / 84.39</td>
<td>86.0 / 356.7</td>
<td>43.5 / 87.18</td>
</tr>
<tr>
<td></td>
<td>n = 10</td>
<td>9.2 / 0.311</td>
<td>24.6 / 0.476</td>
<td>26.8 / 0.467</td>
<td>108.6 / 80.92</td>
<td>111.2 / 82.49</td>
<td>145.8 / 346.2</td>
<td>71.0 / 85.15</td>
</tr>
<tr>
<td>ICC</td>
<td>n = 1</td>
<td>5.2 / 0.113</td>
<td>6.0 / 0.320</td>
<td>6.0 / 0.365</td>
<td>16.2 / 111.6</td>
<td>16.4 / 106.7</td>
<td>20.2 / 754.7</td>
<td>11.7 / 162.3</td>
</tr>
<tr>
<td></td>
<td>n = 5</td>
<td>8.4 / 0.257</td>
<td>20.4 / 0.402</td>
<td>20.4 / 0.405</td>
<td>69.8 / 98.31</td>
<td>72.2 / 97.64</td>
<td>91.0 / 577.4</td>
<td>47.0 / 129.1</td>
</tr>
<tr>
<td></td>
<td>n = 10</td>
<td>9.6 / 0.342</td>
<td>30.4 / 0.555</td>
<td>31.6 / 0.619</td>
<td>117.2 / 88.86</td>
<td>115.6 / 93.29</td>
<td>162.0 / 548.6</td>
<td>77.7 / 122.0</td>
</tr>
<tr>
<td>CCC</td>
<td>n = 1</td>
<td>8.6 / 1.424</td>
<td>9.0 / 5.313</td>
<td>9.0 / 4.803</td>
<td>36.0 / 4440</td>
<td>35.5 / 3807</td>
<td>49.2 / 25311</td>
<td>24.6 / 5595</td>
</tr>
<tr>
<td></td>
<td>n = 5</td>
<td>13.4 / 1.754</td>
<td>42.0 / 3.993</td>
<td>42.0 / 4.003</td>
<td>157.4 / 3373</td>
<td>153.8 / 2868</td>
<td>214.8 / 20161</td>
<td>103.9 / 4402</td>
</tr>
<tr>
<td></td>
<td>n = 10</td>
<td>16.8 / 2.343</td>
<td>76.8 / 3.741</td>
<td>75.2 / 3.807</td>
<td>294.6 / 2659</td>
<td>289.6 / 2221</td>
<td>381.6 / 15252</td>
<td>189.1 / 3357</td>
</tr>
<tr>
<td>manual</td>
<td>—</td>
<td>3</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>22</td>
<td>11.8</td>
</tr>
</tbody>
</table>

HB, heart beat; FFTD, fix-falling trip decision; FRTD, fix-rising trip decision; VFTD, variable-rate-falling trip decision; VRD, variable-rate-rising trip decision; MFTD, manual-reset-falling trip decision.
Table VII. An analysis of test suite generation time.

<table>
<thead>
<tr>
<th>Module</th>
<th>Input file generation (sec)</th>
<th>Constraint solving (sec)</th>
<th>Constraint solving portion (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HB</td>
<td>0.128</td>
<td>0.783</td>
<td>85.9</td>
</tr>
<tr>
<td>FFTD</td>
<td>0.168</td>
<td>3.031</td>
<td>94.7</td>
</tr>
<tr>
<td>FRTD</td>
<td>0.156</td>
<td>2.848</td>
<td>94.8</td>
</tr>
<tr>
<td>VFTD</td>
<td>32.101</td>
<td>505.459</td>
<td>94.0</td>
</tr>
<tr>
<td>VRTD</td>
<td>27.891</td>
<td>439.873</td>
<td>94.0</td>
</tr>
<tr>
<td>MFTD</td>
<td>108.000</td>
<td>1919.861</td>
<td>94.7</td>
</tr>
</tbody>
</table>

HB, heart beat; FFTD, fix-falling trip decision; FRTD, fix-rising trip decision; VFTD, variable-rate-falling trip decision; VRTD, variable-rate-rising trip decision; MFTD, manual-reset-falling trip decision.

and mutants for which the authors already knew the location. However, the analysis of why a certain real error or mutant was not detected took considerable time (e.g., two or three hours each).

5.4. Threats to validity

There are several aspects readers must be aware of as potential threats to external validity. First, FBD implementation of the BP used in the study is not guaranteed to be a sufficient representative sample of real-world FBD programs. Although FBD programming is widely used, few industrial implementations are available in the public domain. Few open source projects use FBD as an implementation language. The reality is that most real industrial FBD programs are safety-critical applications in which FBD programs remain confidential and are stored in proprietary formats.

Second, the case study did not fully investigate the effectiveness of the FBDTester with respect to complex function blocks (e.g., TON), because the target program did not include them. However, the authors remain convinced that a toy-sized and artificially created sample carries little practical value. Rather, it is beneficial to demonstrate that the proposed approach is useful in delivering software quality even when applied to real-world applications without simplifications. Instead, the architecture of the FBDTester is designed to be extensible in order to easily add new functions or function blocks to the library files.

Finally, the type of SMT solver used in the experiment could be another threat. Yices is apparently not the only SMT solver in existence, nor is it necessarily the representative tool. Yices 1 was adequate for implementing the FBDTester and demonstrating that automated test case generation is feasible and highly effective. In addition, the target program did not require non-linear arithmetic functions such as SQRT, SIN, and COS, which are not supported by Yices 1. The FBDTester is not limited to Yices when it comes to choosing an SMT solver. Other SMT solvers could be integrated into the FBDTester, and multiple SMT solvers could possibly be deployed to best solve test requirements automatically.

6. CONCLUSION

This paper proposed how a test case generation procedure for FBD programs could be automated. Given a unit FBD program and chosen test coverage criteria, test requirements according to the coverage criteria were generated. In order to generate test cases meeting the required coverage goal, an SMT solver was used.

A real-world case study using the BP of reactor protection systems showed that the generated test suites could detect real errors contained in the target software and could also detect various mutants with much better performance than the manual test suites prepared by domain experts. It is apparent that the automated test case generation of FBD programs of industrial size and complexity is not only feasible but also highly effective. The FBDTester is an effective tool to provide assurance to test engineers who implement safety-critical software using FBD.

Although useful, when generating state-dependent test cases, the FBDTester identifies required values of internal memory states, but does not automatically generate a sequence of test cases reaching the intermediate states. Automated generation of preambles remains an area for further development and investigation.
research. Achieving 100% coverage is not always possible due to infeasible test requirements that are sometimes caused by an awkward programming structure. Providing adequate tool support on the proper handling of infeasible test requirements remains an interesting challenge for future work.

ACKNOWLEDGEMENTS

This research was partly supported by Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education(NRF-2013R1A1A2058618), the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (NIPA-2014-H0301-14-1023) supervised by the NIPA(National IT Industry Promotion Agency), the ICT R&D program of MSIP/IITP[10044457, Development of Autonomous Intelligent Collaboration Framework for Knowledge Bases and Smart Devices], and DGIST CPS Global Center.

REFERENCES


