Evolutionary Approach to Coverage Testing of IEC 61499 Function Block Applications

Igor Buzhinsky*, Vladimir Ulyantsev*, Jari Veijalainen†, Valeriy Vyatkin‡§*
* Computer Technologies Laboratory, ITMO University, St. Petersburg, Russia
† Department of Computer Science and Information Systems, Mattilanniemi 2,
University of Jyvaskyla, FI-40014 University of Jyvaskyla, Finland
‡ Department of Electrical Engineering and Automation, Aalto University, Finland
§ Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Sweden
igor.buzhinsky@gmail.com, ulyantsev@rain.ifmo.ru, jari.veijalainen@jyu.fi, vyatkin@ieee.org

***Abstract*—The paper addresses the problem of coverage testing of industrial automation software represented in the IEC 61499 standard, one of the recent standards for distributed control system design. Contrary to model-based testing (MBT), the paper focuses on implementation coverage, not model coverage. An approach based on evolutionary algorithms is presented which generates coverage test suites for both basic and composite IEC 61499 function blocks. It employs two third-party tools, FBDK and EvoSuite. The evaluation of the approach was performed on a set of control applications for two lab-scale demonstration plants. Results show that the approach is applicable and shows good performance at least on basic function blocks. The generated tests suites helped to discover several unreachable system parts, which pinpointed errors in the systems under test.**

I. Introduction

The paper is devoted to automated test suite generation for control software represented in the IEC 61499 standard [1], which establishes a way to design distributed control applications for industrial automation systems in a visually clear way. Unlike the Model-Based Testing (MBT) [2] approach, we are interested in test data generation to ensure 100% coverage of software implementation, not its specification-based models. Since the coverage of a specification does not imply the coverage of the whole implementation, this should allow to test software more thoroughly. While finite-state models play a central role in the IEC 61499 standard, we will still utilize several concepts from MBT, such as transition coverage of finite-state machines [3].

This paper presents a test input data generation method to handle the stated problem. As systems under test (SUTs), we employ *function blocks* (FBs), one of the basic concepts of the IEC 61499 standard, which encapsulate behavior and state and thus are similar to the concept of classes in object-oriented programming. This means that the considered problem, among the multiple testing levels, is mostly related to unit testing. Nevertheless, since entire software systems can be represented in the form of FBs, the scope of the problem is wider. Generated tests can be utilized in two ways. First, unreachable code in systems can be identified by examining uncovered parts. Second, input test data can be augmented with output data based on a model of the software. This will allow to search for faults in the software behavior.

As the way to achieve the stated goal, evolutionary computation [4], a general optimization methodology for both discrete and continuous problems, is applied. Among other heuristic search-based techniques, this methodology is widely used in software engineering [4], [5]. This approach has already been exploited for coverage test generation for general-purpose systems [6], but, to the best of our knowledge, was not applied to solve this problem for industrial automation software and for systems represented using the IEC 61499 standard in particular.

The remainder of this paper is structured as follows. In Section II, we discuss related research on MBT, source-based test generation approaches, and evolutionary methods. In Section III, we introduce the IEC 61499 standard, describe the SUT models we consider, and define the problem being solved more formally. Next, in Section IV we describe the proposed test generation approach and in Section V we evaluate it on two IEC 61499 conformant systems. The paper is concluded in Section VI.

II. Related Research

In this section we review works related to our study. First, several known MBT methods for industrial automation system are shortly discussed. Second, works about test generation methods for system implementations are examined. In the end, we also shortly review the field of evolutionary computation.

A. MBT Approaches for Industrial Automation Systems

Many coverage test generation methods are already known in the broad field of MBT. The general idea of MBT is to employ formal models of software, which can be obtained from its requirements, to analyze them and to generate test suites which can demonstrate the conformance of the software to its specification. Methods known from MBT often aim to achieve *coverage* properties of specification-based models of SUTs. Finite-state machines often serve as such models, and several coverage criteria [2], such as state, transition and path coverage, are based on them. Since automatic model-based test generation is a very broad field, we review several works from the more narrow field of test generation for industrial automation systems.

The authors of [7] propose an automated test case generation approach for industrial automation applications specified

978-1-4799-6648-6/15/$31.00 ©2015 IEEE

by UML state charts. Information about test cases is derived from the state charts which model both the plant's and the controller's behavior. The construction of test cases includes the automated generation of the test suite model from the test suite meta-model, which represents the test suite structure. The approach is tested on a sorting machine software system represented in the IEC 61499 standard, but it is also applicable for IEC 61131-3 [8] systems, as they are also based on the concept of an FB.

Next, in [9], a unit test case generation method, also based on UML diagrams, is presented specifically for the IEC 61499 standard. This method complements a complete software development process also proposed in this paper. Both state and activity UML diagrams, which represent software specification on different levels of abstraction, are subject to test generation. Round-trip path coverage [2] is attempted to be reached for state diagrams.

Finally, in [10] the MBT approach is augmented with simplified MBT model creation, which is supported by code generation from source information in the CAEX format [11], e.g. information about control loops within the system. The suggested approach is applied to a SUT represented using the IEC 61131-3 notation. In this study, Conformiq Designer[1] is used for both creating MBT models and for test generation.

B. Test Generation Methods for Software Implementations

We now move to studies devoted to solving the coverage test generation problem for software implementations written in general purpose languages, such as Java or C++.

One of the first approaches to this problem was the one introduced in [12]. The technique presented in [12] is based on the constraint satisfaction problem (CSP) and mutation analysis. The generated test data approximates relative adequacy, or mutation adequacy: a test satisfies the relative adequacy criterion, when it causes a certain number of incorrect programs to fail. In turn, incorrect programs are the results of mutations, or small changes, of the original program under test. Algebraic constraints are generated and then solved in order to ensure the failure of mutated programs. This approach is not suitable for coverage test generation, since it ensures mutation adequacy instead.

In [13], a survey on test data generation methods is presented. They are separated into three types. The simplest type is random testing: it just suggests randomly generating input test data for the SUT, and, quite obviously, it usually does not perform well in terms of coverage. The second approach is goal-oriented test data generation, which is subdivided into the chaining approach and the more successful assertion-oriented approach. In the former, data dependencies are used to solve branch predicates, and in the latter, assertions are inserted into the source code either manually or automatically, and then the test generator attempts to find any path of program execution which violates the assertions. Finally, path-oriented test data generation is the strongest one. In this approach, the test generator attempts to follow specific control paths.

A complex, combined approach to test generation is taken in [6], where a tool called EvoSuite is presented. This tool supports automated unit test case generation for Java source code. Generated test suites are compatible with the JUnit library[2]. The approach is based on evolutionary computation [4] and optimizes test suites with respect to source coverage. Other techniques employed include hybrid search, dynamic symbolic execution and testability transformation. In addition, test oracles, which assess the correctness of the program's behavior, are automatically created in the form of assertions which summarize the behavior of the program. The effectiveness of assertions is estimated using mutation testing, which has already been shortly described when describing the constraint-based approach [12].

Finally, there is a number of symbolic approaches [14] to test generation and dynamic approaches which combine symbolic and concrete execution. Such approaches traverse the control flow graph (CFG) of the program, maintaining a set of constraints which are required for the current path to be executed. Tests are obtained by solving these constraints.

C. Evolutionary Computation

Evolutionary algorithms and metaheuristics in general are optimization methods applicable for various discrete and continuous problems. Tasks, for which evolutionary algorithms are applied, are usually not solvable in polynomial time by precise algorithms (unless P = NP). Such problems include, for example, the traveling salesperson problem [15] and the job shop problem [16]. Evolutionary algorithms usually do not guarantee that the optimal solution of the considered problem will be found in a reasonable time. Still, they are effective in practice.

The basic idea of evolutionary computation is as follows. Evolutionary algorithms use some particular representations of possible solutions (also called *individuals*) and usually reach new solutions by making small adjustments to previous ones (these changes are called *mutations*) or by combining different solutions (this operation is known as *crossover*). A quality measure, *fitness function*, which maps individuals into the real axis, guides the evolutionary search, so that the worse individuals are discarded, and the best ones are retained. This procedure is known as *selection*.

The genetic algorithm [17] is one of the earliest evolutionary algorithms proposed. It simultaneously operates with a number of individuals, called the *generation*. On each iteration of this algorithm, individuals are recombined, mutated, and then selected. Another well-known and a much simpler algorithm is the random mutation hill climber [18], which operates with a single individual and applies mutations to modify it. Many further algorithms have been introduced recently, including, for example, Natural evolution strategies [19] and the Mutation-based ant colony optimization algorithm [20].

III. Problem Statement

In this section we define the problem we deal with more precisely. In order to do this, we first shortly review the basic concepts of the IEC 61499 standard.

[1] http://www.conformiq.com/

[2] http://junit.org/

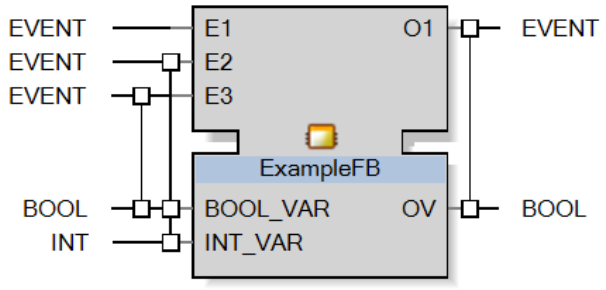

Fig. 1. An example of an FB interface

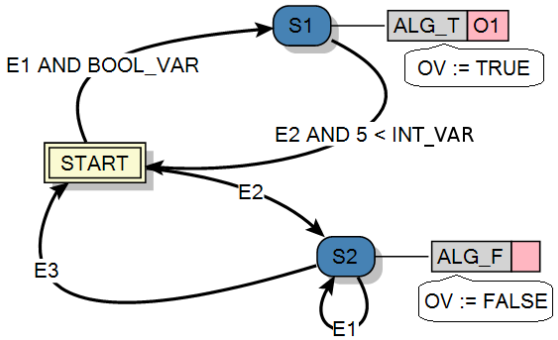

Fig. 2. An example of an ECC in a basic FB

A. IEC 61499 Function Blocks

The IEC 61499 [1] is an open standard for distributed control and automation. The purpose of its introduction was to allow the development of distributed control systems, which can be deployed into many programmable logic controllers (PLCs), with robust, reusable modules. Nowadays, the standard is attempted to be used in production: an example is shoe manufacturing industry [21]. However, according to [22], its application faces several challenges, including the unfamiliarity of practitioners with the semantics of the standard and the inability of the standard to address some stages of the development process.

The IEC 61499 standard requires a control application to be represented by a number of *function blocks* (FBs), either *basic* or *composite* ones, which are interconnected to form a network. An FB is an entity with a defined interface which can encapsulate both behavior and state. An example of an FB interface is shown in Fig. 1. It has inputs and outputs in the form of events and data of common types (e.g. Boolean, integer). Input events can be associated with input variables: this means that the FB requests the most recent values of these variables when the event is received. These associations are shown in Fig. 1 as vertical lines with boxes to the left of the FB body. Similar associations exist for outputs.

Basic FBs are implemented using the concept of *execution control charts* (ECCs), which are also referred to as Moore finite-state machines (FSMs). An ECC has several *states* and is in exactly one state at each moment of FB execution. One of the states is the *start state*. Each state might have *algorithms* to be executed when the state becomes active, and might generate *output events*. Algorithms are usually implemented in the Structured Text language. They operate with *variables*: input, output or internal ones. Next, states are connected to each other with *transitions*. Transitions are usually triggered by events and are executed if *guard conditions* are met. Such conditions are defined over the set of variables of the FB. The choice of transitions to be executed when an event is received is always deterministic: situations where several transitions can execute are arbitrated by the transition declaration order. It is possible that no transitions are executed when an event occurs. Moreover, it is possible that one input event causes several state changes: this is due to *spontaneous* transitions, which do not require events to be executed. If there is a spontaneous transition from the current state with a satisfied guard condition, then it always executes.

FB invocation by an input event can result in a reaction: an output event (possibly, several events or even an infinite sequence of events) and a change of output data variables associated with the emitted output event. The absence of a reaction can be explained by not emitting any events, by the lack of event-data associations (even if the event is emitted, the new data is not visible outside the FB), or by an infinite loop in the ECC. When an ECC is idle (i.e. there are no spontaneous transitions with satisfied guard conditions which can be executed right now), the FB's state is fully determined by the values of its variables and the state of the ECC.

An example of an ECC is shown in Fig. 2. This ECC is compatible with the interface shown in Fig. 1, and thus can form a basic FB together with it. The ECC has three states, two of which (S1 and S2) are associated with algorithms (ALG_T and ALG_F), and one of which (S1) has an output action (O1). Algorithms ALG_T and ALG_F alter the value of the Boolean output variable OV.

Inside a *composite FB* there is a network of FBs of other types with *event and data connections* between them. The inputs of composite FBs are connected to the inputs of nested FBs, and the outputs are linked in a similar way. Nested FBs, either basic or composite, may also have predefined input variable values. Composite FBs allow reusing various particular arrangements of lower level FBs.

B. Tests and Test Suites

To define a test, we first consider the FB under test, either basic or composite, which is the SUT in our case. Assume that it has input events $E_1, ..., E_n$ and input variables $V_1, ..., V_m$ with finite domains $D_1, ..., D_m$, where domains represent values of particular data types. Next, Boolean values $W_{i,j}$ signify whether the event E_i is associated with the variable V_j. In addition, consider the element $\perp$, which does not belong to any of $D_j, j = 1..m$. This element stands for "no value" and is used when an event is not associated with an input variable. An *input tuple* is a tuple $(E_i, \alpha_1, ..., \alpha_m)$, where $\alpha_j, j = 1..m$ is either from D_j, if $W_{i,j}$, or $\perp$ otherwise. Thus, an input tuple only contains the values of the variables a particular event is associated with. Input tuples can be fed to the FB and thus trigger its execution steps.

A *test* is a finite sequence of input tuples. Note that outputs are not included into tests, because they are not significant for defining coverage criteria and maximizing them. A test can describe a series of FB execution steps. It is also assumed that

TABLE I. AN EXAMPLE OF A TEST WITH LENGTH 4

Tuple number	E_i	α_1 (BOOL_VAR)	α_2 (INT_VAR)
1	E3	true	$\perp$
2	E1	$\perp$	$\perp$
3	E2	false	-100
4	E2	false	42

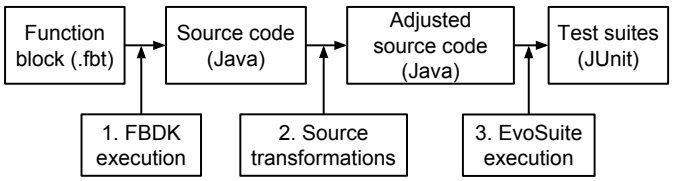

Fig. 3. The scheme of the proposed approach

before test execution the FB is in its initial state: all ECCs are in their start states, and all the variables are initialized with their default values. An example of a test for an FB with the interface from Fig. 1 is shown in Table I. Finally, a *test suite* is a finite set of tests.

Besides, assume that some *coverage criterion* is defined. A coverage criterion is a real-valued function of an FB and a test suite for this FB. For instance, *transition coverage*, which we will use later, is the fraction of all transitions inside the ECCs of all possible types in the FB which are executed at least once when all the tests are run. Another example of a coverage criterion is *branch coverage*, which is source-based (assume that the FB is transformed into a source code) and involves measuring the coverage of various methods and 'if-then-else' branches inside them.

Based on the presented definitions, we formulate the problem being solved in this paper: design a method which generates test suites and maximizes known coverage criteria for given FBs.

IV. COVERAGE TEST GENERATION APPROACH

The proposed coverage test suite generation approach combines FB transformation to Java source code and the evolutionary search of test suites which maximize the coverage of the obtained Java code. Two coverage criteria are considered:

- Transition coverage: the share of executed transitions in the ECC of a basic FB, or the share of executed transitions of all nested FBs inside a composite FB. In the latter case, if there are several nested FBs of one FB type, their transitions are counted once.
- Branch coverage of the Java source code obtained from an FB. For a basic FB, this criterion includes not only the coverage of its transitions, but also the coverage of all branches in its algorithms. For a composite FB, this criterion includes the coverage of the same items of all basic FBs inside it.

The approach is summarized in Fig. 3. The input of the test generation method is an .fbt XML file which describes the FB under test. If this FB is composite, XML descriptions of the nested FBs should also be available. The method comprises three stages.

Stage 1. A third-party tool, FBDK[3], transforms an .fbt description of the FB under test to a Java source file. For a basic FB, it creates a class with state, event and variable declarations, event processing methods and methods for its algorithms. A class for a composite FB declares its nested FBs and creates connections between them in its constructor. This transformation is automated and is implemented as a call to a Java library supplied with FBDK.

Stage 2. The obtained source code is transformed to prepare it for evolutionary test generation, which will be done by another tool. First, a new Java class is created which includes the FBDK-generated class as a nested one. For composite FBs, all their dependencies are also included as nested classes. Nested classes are marked as private to suppress the generation of tests which call their methods.

Next, for each input event of the FB under test a public method is created in the outer class. Thus, only such event methods are accessible from the outside. Each generated event method accepts the variables associated with the input event as arguments, updates variable values of the proper instance of a nested FBDK-generated class and executes a corresponding event method on this instance.

Additionally, for each transition in each nested FB class, an empty private method is added to the outer class. This method is executed along with the execution of the code corresponding to the transition. The purpose of these methods is to allow test generation which optimizes transition coverage (see the next stage).

Stage 3. The modified source code is fed to EvoSuite[4] [6], a tool which generates tests for Java programs using branch coverage as the fitness function. It implements several evolutionary algorithms, among which the default steady-state genetic algorithm [17] is chosen. Depending on the coverage criterion employed, EvoSuite is configured to either generate tests to cover the whole class, or to cover only the transition methods created in the end of the previous stage. The search is performed for a fixed time span. The result of EvoSuite execution is a JUnit test suite. As only event methods were left public in the previous stage of the approach, such test suites are comprised of sequences of their executions supplied with input variable values. Here is the example of a test from Table I as it would appear in the body of a single JUnit test:

```
ExampleFB fb = new ExampleFB();
fb.service_E3(true);
fb.service_E1();
fb.service_E2(false, -100);
fb.service_E2(false, 42);
```

The first two stages of the method were implemented in Java, and a bash script was written for the third stage. The source code of the whole project is available online[5].

V. EXPERIMENTS AND RESULTS

This section describes the conducted experimental evaluation of the proposed approach applied to two sets of FBs and the obtained results.

[3]http://www.holobloc.com/doc/fbdk/

[4]http://www.evosuite.org/

[5]https://github.com/igor-buzhinsky/indin2015_source

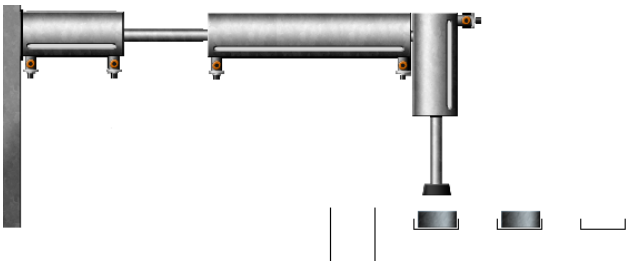

Fig. 4. A scheme of one of the implementations of the pick-and-place manipulator

A. Systems under Test

We employ two software systems which are designed to control simple plants in the laboratory environment. The first system or, more precisely, a set of similar systems, is the control application for the pick-and place (PnP) manipulator which was earlier used in [23] to evaluate an approach to a different problem. One of the implementations of this device is shown in Fig. 4. This screenshot from FBDK shows two horizontal and one vertical cylinders connected one to another. This system of cylinders should pick objects from three plates and place them into the bin to the left of the plates. The system consists of 31 basic and 17 composite FBs implemented in FBDK.

The second system is the application which regulates a heat production process (HPP). In [10], an IEC 62424 application is mentioned, and the system we work with is the result of the redesign of this application for the IEC 61499 standard. The nxtSTUDIO software[6] was used for the redesign. FBs designed in nxtSTUDIO can be processed with FBDK after minor adjustments. This version of the system, however, is not very modular and has only one composite FB. Twelve other FBs are basic.

The number of input events among the FBs from the described control systems ranges from 1 to 7 with the median value of 2. The number of input variables among these FBs is generally higher: it ranges from 0 to 34 with the median value of 6. Basic FBs have between 2 and 15 states and between 2 and 21 transitions with median values of 3 and 4, respectively. Finally, the length of FBs, counted as the number of lines of resulting Java code, ranges from 92 to 4725 with the median value of 320. Large code size (i.e. more than 1000 lines of code) is typical for composite FBs, as they contain the sources of their dependencies inside.

B. Experiment Setup

For each FB, we executed two experiments for each of both coverage criteria: transition and branch coverage. As described in section IV, FBs were transformed into Java source code which was fed to FBDK. Ten minutes were given to EvoSuite to generate tests for basic FBs, and twenty minutes were given for composite ones. The computation was performed on a PC with a 2.2 GHz Intel Core i7-2670QM CPU. Besides, if EvoSuite obtained 100% coverage, it could finish its work earlier.

[6]http://www.nxtcontrol.com/en/engineering/

TABLE II. OBTAINED COVERAGE VALUE STATISTICS

FB type, coverage criterion	Min	First quartile	Median	Third quartile	Max
Basic, branch	60.0%	88.3%	92.6%	94.8%	98.8%
Composite, branch	35.4%	79.5%	84.5%	91.0%	94.8%
Basic, transition	55.6%	100.0%	100.0%	100.0%	100.0%
Composite, transition	5.7%	92.0%	100.0%	100.0%	100.0%

C. Results

The results of the experiments are outlined in Table II, where basic statistics are shown for all four groups of experiments. The results are combined for both SUTs. The data suggests the following conclusions. First, transition coverage was generally easier to achieve. Perfect (100%) result was achieved for more than 75% basic FBs (in fact, for 42 out of 43) and for more than 50% (11 out of 18) composite FBs. This can be explained by the fact that achieving transition coverage is an easier goal: there is no need to cover all execution paths of ECC algorithms. Coverage values are also better for basic FBs independently of coverage criteria, and this can be explained by the size difference and the fact that perfect coverage is not always required for composite FBs, unless they represent whole software systems.

After the results had been obtained, the generated tests were run in the Eclipse IDE[7] with the EclEmma plugin[8], which integrates Eclipse with JUnit. Uncovered FB parts were manually examined. Based on this examination, several deductions were made:

- Some small parts of the automatically generated code appeared to be inaccessible due to the way FBDK generates code. An example of such part is a branch in an event processing method for the case of an invalid event (i.e. an event which matches none of the input events and thus is impossible in normal situations) If such parts are not considered in branch coverage, then 18 out of 43 basic FBs and 4 out of 18 composite FBs are perfectly covered by the generated test suites.
- In EvoSuite, branch coverage assumes the coverage of each combination of conditions in an ‘if’ decision. If this condition is weakened to just cover ‘then’ and ‘else’ branches in each decision, then additionally 6 basic FBs and 1 composite FB can be considered as completely covered.
- Some basic FBs, especially from the PnP application, contained algorithms which were not associated with any state and thus were inaccessible. This can be considered as a fault of the software design, but one does not need tests to understand that they are unreachable.

Since the evolutionary approach does not guarantee the optimality of solutions, we also attempted to cover the uncovered parts in basic FBs manually. Gaps in branch coverage of two FBs were covered by augmenting the test suite generated

[7]https://eclipse.org/

[8]http://www.eclemma.org/

for the branch criterion with a test from the corresponding transition-based test suite. For another FB, it was quite easy to modify one of the automatically generated tests to improve its branch coverage. Finally, we identified one basic FB with several states inaccessible due to a forgotten update of an internal variable and two basic FBs with algorithm branches inaccessible due to badly written 'if' decisions. For example, one algorithm inside an FB from the HPP system contained the following decision: `AI.value < PRESET_H.value & AI.value >= PRESET_H.value`, which is obviously unsatisfiable.

In addition, we were able to explain the low coverage results for two composite FBs. The first FB from the PnP system, which had got 35.4% and 5.7% for branch and transition coverage respectively, had missing event connections from its input interface to nested FBs. Some parts of the second FB, the only composite FB in the HPP system, which had got 64.4% for both branch and transition coverage, were inaccessible due to fixed default values of several variables. It also included faulty basic FBs with inaccessible parts.

VI. Conclusion

A method which generates input test data for IEC 61499 function blocks and tries to maximize test suite coverage has been proposed in the paper. The obtained results and their manual examination suggest that the proposed method has encouraging performance (at least on basic FBs) and is applicable in practice. In particular, it helped to identify several faults in the SUTs, which made some of their parts unreachable.

The performed study has several limitations. To begin with, as code generated by FBDK is quite rigid, the number of available coverage criteria is very limited. In addition to default branch coverage, transition coverage was implemented, but more complex criteria, such as path or loop coverage, are not supported. This issue can be overcome by using a separate FB representation, which does not depend on FBDK. Another limitation is connected with the nature of evolutionary algorithms, which do not always generate perfect solutions. To partially resolve it, it is possible to replace the third stage of the method (EvoSuite execution) with one of symbolic constraint-based approaches [14]. Next, we have not cared about output data which can be added to generated test suites so that they could check the correctness of FB outputs. Lastly, the used SUTs do not completely represent the complexity of industrial automation software, as they were designed to control relatively simple devices.

Acknowledgements

This work was financially supported by the Government of Russian Federation, Grant 074-U01.

References

[1] *International Standard IEC 61499-1: Function Blocks – Part 1: Architecture, 2nd ed.* Geneva: International Electrotechnical Commission, 2012.

[2] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-based testing of reactive systems: advanced lectures. Lecture Notes in Computer Science*. Springer, 2005, vol. 3472.

[3] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen, *Introduction to algorithms*. MIT press, 2001.

[4] M. Harman, "Software engineering meets evolutionary computation," *Computer*, vol. 44, no. 10, pp. 31–39, 2011.

[5] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[6] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.

[7] R. Hametner, B. Kormann, B. Vogel-Heuser, D. Winkler, and A. Zoitl, "Test case generation approach for industrial automation systems," in *5th International Conference on Automation, Robotics and Applications (ICARA 2011)*. IEEE, 2011, pp. 57–62.

[8] *International Standard IEC 61131-3: Programmable controllers – Part 3: Programming languages, 2nd ed.* Geneva: International Electrotechnical Commission, 2003.

[9] T. Hussain and G. Frey, "UML-based development process for IEC 61499 with automatic test-case generation," in *11th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2006)*. IEEE, 2006, pp. 1277–1284.

[10] J. Peltola, S. Sierla, P. Aarnio, and K. Koskinen, "Industrial evaluation of functional model-based testing for process control applications using caex," in *18th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2013)*. IEEE, 2013, pp. 1–8.

[11] *International Standard IEC 62424: Specification for representation of process control engineering requests in P&IDs*. Geneva: International Electrotechnical Commission, 2008.

[12] R. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[13] J. Edvardsson, "A survey on automatic test data generation," in *2nd Conference on Computer Science and Engineering*. ECSEL, 1999, pp. 21–28.

[14] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[15] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: A review of representations and operators," *Artificial Intelligence Review*, vol. 13, no. 2, pp. 129–170, 1999.

[16] F. Della Croce, R. Tadei, and G. Volta, "A genetic algorithm for the job shop problem," *Computers & Operations Research*, vol. 22, no. 1, pp. 15–24, 1995.

[17] L. Davis *et al.*, *Handbook of genetic algorithms*. Van Nostrand Reinhold New York, 1991, vol. 115.

[18] M. Mitchell, J. H. Holland, and S. Forrest, "When will a genetic algorithm outperform hill climbing?" *Advances in Neural Information Processing Systems*, vol. 6, pp. 51–58, 1994.

[19] D. Wierstra, T. Schaul, J. Peters, and J. Schmidhuber, "Natural evolution strategies," in *2008 IEEE Congress on Evolutionary Computation (CEC 2008)*. IEEE, 2008, pp. 3381–3387.

[20] D. Chivilikhin and V. Ulyantsev, "MuACOsm: a new mutation-based ant colony optimization algorithm for learning finite-state machines," in *15th Genetic and Evolutionary computation conference (GECCO 2013)*. ACM, 2013, pp. 511–518.

[21] M. Colla, A. Brusaferri, and E. Carpanzano, "Applying the IEC-61499 model to the shoe manufacturing sector," in *11th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2006)*. IEEE, 2006, pp. 1301–1308.

[22] K. Thramboulidis, "IEC 61499 in factory automation," in *Advances in Computer, Information, and Systems Sciences, and Engineering*. Springer, 2006, pp. 115–124.

[23] S. Patil, V. Vyatkin, and M. Sorouri, "Formal verification of intelligent mechatronic systems with decentralized control logic," in *17th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2012)*. IEEE, 2012, pp. 1–7.