Igor Buzhinskii

EVOLUTIONARY APPROACH FOR ACHIEVING STRUCTURAL COVERAGE IN TESTING IEC 61499 FUNCTION BLOCK SYSTEMS

UNIVERSITY OF JYVÄSKYLÄ
FACULTY OF INFORMATION TECHNOLOGY
2015

ABSTRACT

Buzhinskii, Igor
Evolutionary approach for achieving structural coverage in testing IEC 61499 function block systems
Jyväskylä: University of Jyväskylä, 2015, 59 p.

Software Engineering, Master's thesis
Supervisors: Veijalainen, Jari; Vyatkin, Valeriy; Shalyto, Anatoly.

The topic of this thesis is automated test generation for control software represented in a specific standard, the IEC 61499. This standard, which is largely based on the concept of function block, establishes a way to design distributed control systems in a visually clear way. The goal of the thesis was to design a test generation approach or a number of such approaches that would produce input test data with high coverage of the implementation of systems under test. Coverage is a measure which expresses the fraction of the system that was exercised at least ones when all tests in a test suite were run on this system. To reach the stated goal, evolutionary computation, a general optimization methodology, was employed. In this methodology, possible solutions of the problem (in our case, test suites) are developed during a simulated evolution process which involves mutating solutions (that is, altering them insignificantly) and combining them into new ones.

Two methods of test suite generation were designed based on the mentioned approaches. The experimental evaluation showed that one of them produces test suites with high coverage but is time consuming, and another one is more flexible and fast, but produces test suites with lower coverage. It was also shown that the proposed methods are capable of identifying faults in control software under test, which are mainly connected with unreachable system segments.

Keywords: IEC 61499, industrial automation system, function block, test generation, coverage criterion, testing automation, evolutionary computation.

TIIVISTELMÄ

Buzhinskii, Igor
Evoluutiomenetelmän käyttö IEC 61499 -standardia noudattavien toimintolohkojen testiaineistojen generointiin
Jyväskylä: Jyväskylän yliopisto, 2015, 59 s.

Ohjelmistotuotanto, pro gradu – tutkielma
Ohjaajat: Veijalainen, Jari; Vyatkin, Valeriy; Shalyto, Anatoly.

Tämän opinnäytetyön teemana on testitapausten automaattinen generointi IEC 61499-standardin mukaisille toimintolohkoille. Kyseinen standardi perustuu oleellisesti äärellisiin automaatteihin, joille on määritelty kaksiulotteinen visuaalinen esitysmuoto ja joista voidaan toisiinsa kytkemällä koota yhdistettyjä toimintolohkoja. Lohkojen tilasiirtymiin liittyvän laskennan kuvaamiseen on standardissa tarjolla erityinen ohjelmointikieli. Opinnäytetyön tarkoituksena oli yhden tai useamman menetelmän kehittäminen, jotka generoivat testiaineistoja em. standardin mukaisille toimintolohkojen ohjelmallisiille toteutuksille. Tavoitteena oli tuottaa testiaineistoja, joilla on mahdollisimman korkea tilasiirtymä- tai haarakattavuus. Testiaineistojen generointi perustuu evoluutioalgoritmeihin, jotka pyrkivät maksimoimaan em. kattavuudet toteutuksen suhteen. Algoritmit muokkaavat testiaineistoa sukupolvi sukupolvelta paremmaksi em. kriteerien suhteen (periytyminen). Muokkaus perustuu yksittäisten testitapausten vähäiseen muuttamiseen (mutaatioihin) ja vanhojen testitapausten yhdistämiseen uusiksi testitapauksiksi testiaineistossa (rekombinaatio).

Työssä kehitettiin kaksi testiaineistojen generointimenetelmää, joiden ominaisuuksia testattiin kahdella yhdistetyn toimintolohkon toteutuksella. Toinen näistä ohjaa pientä poimi-ja-sijoita laitekokonaisuutta ja toinen lämpövoimalaitoksen yksinkertaistalaborotoriomallia. Tulokset osoittavat, että toinen menetelmä generoi testiaineistoja, joilla on korkea kattavuus, mutta laskenta-aika oli suhteellisen suuri. Toinen menetelmä on joustavampi ja laskee nopeammin, mutta tuotettujen testiaineistojen kattavuus onoli pienempi kuin edellisen. Menetelmiä kokeiltaessa selvisi myös, että ne kykenevät löytämään testattavista järjestelmistä virheitä, eli saavuttamattomia tilasiirtymiä ja ohjelmahaaroja.

Asiasanat: IEC 61499, teollinen automaatiojärjestelmä, toimintolohko, testiaineiston generointi, kattavuuskriteeri, testauksen automatisointi, evoluutiolaskenta.

FIGURES

TABLES

TABLE OF CONTENTS

1 INTRODUCTION

Control software is an important element in modern industrial automation systems (Zoitl & Vyatkin, 2009), examples of which are manufacturing and material handling systems. It is responsible for the safety and correctness of their operation. This means that these systems should be properly tested or verified. One of the recent standards for the design of such control systems is IEC 61499 (IEC, 2012), which uses *function blocks* as building units of a software system. It is aimed at increasing the flexibility and adaptability of such systems (Zoitl etc., 2009) and is oriented towards distributed control.

One of the ways of ensuring that control systems work correctly is testing (IEEE, 2013). In testing, a set of *test cases*, or *tests* for simplicity, is prepared for the system to be checked for errors (such sets are called *test suites*). Each test is a sequence of calls to the system (e.g. method calls for a particular class in case of object-oriented programming). Systems of our interest have finite interface specifications with events and input variables, and thus tests for them will consist of event submissions with corresponding variable values. Such submissions can also be represented as method calls.

Unlike software verification, testing cannot guarantee the correctness of the system in practice (though, in theory, the problem can usually be solved by preparing an impractically large set of test cases that would cover all parameter values that would ever be used in production), but can still reveal many errors. In addition, test execution usually takes less time than the verification procedure. It is also possible to automate the procedure of test creation. A simple and formal measure of the quality of a test suite is *coverage*, which might be defined in a number of ways.

In the previous studies, much work was done in the field of model-based testing (Broy, Jonsson, Katoen, Leucker & Pretschner, 2005, pp. 281–387). First, various coverage criteria were defined (Broy etc., 2005, pp. 295–297), including the ones which explicitly involve finite-state machines (Cormen, Leiserson, Rivest & Stein, 2001), entities which are widely employed in the IEC 61499 standard. Then, test generation methods were developed which aimed at covering the specification-based model of the software. Conversely, the goal of

this Master's thesis is to design a method of generating tests which cover the *implementation* of IEC 61499 conformant software. Test generation for software implementation is also a known problem, and one of its recent successful solutions (Fraser & Arcuri, 2011) involves the evolutionary approach (Harman, 2011). In this approach, possible solutions of the problem (in our case, test suites) are explored during a simulated evolution process, in which they are altered (mutated) and combined with each other.

To the best of our knowledge, the evolutionary approach, as well as other implementation-based approaches, was not applied to industrial automation software and, in particular, to IEC 61499 control systems. This thesis addresses the mentioned issue and presents two test suite generation methods. The first method employs two third-party tools and suggest to split the problem into two parts: first, translate the function block under test to a source code in a general purpose language (e.g. *Java*), and second, to optimize the coverage of this source code. The second approach uses a similar scheme, but is much less based on third-party software, which makes it more flexible. The proposed methods, which differ in their advantages and disadvantage, are shown to be applicable in practice: they are able to detect real faults in control applications.

The following research questions are considered in the thesis:

1. Which features of the IEC 61499 standard are relevant for the test suite generation problem?
2. How to represent tests and test suites for IEC 61499 applications?
3. Which approaches that tackle the stated problem or similar problems already exist? Which elements of these approaches can be used in this thesis?
4. Which evolutionary algorithms can be applied to solve the problem, if any?
5. How to design the test generation method which will fulfill the goal of the thesis?
6. How to evaluate the proposed solution?

The rest of the thesis is organized as follows. Section 2 outlines the employed research methods, which will answer the research questions, namely the literature review and constructive research. As an overall framework, design science paradigm is adopted. Next, Sections 3, 4 and 5 describe the background of the study: the IEC 61499 standard, software testing automation and the field of evolutionary computation. The contribution of the thesis, namely two test suite generation methods, is presented in Sections 6 and 7. Finally, the evaluation and the comparison of the methods are performed in Section 8, and the results of the thesis are concluded and discussed in Section 9.

2 METHODS

This section shortly describes the methodology of the research. To answer the research questions stated in the introduction, two research methods are used. The first research method is the literature review (Creswell, 2007). Three fields of knowledge will be reviewed: the IEC 61499 standard, existing testing automation techniques and evolutionary computations. Thus, research questions 1–4 will be answered.

The second research method is artifact construction. Both methods are embedded in the design science approach (Hevner, 2004), in which a new artifact (in our case, a new method to generate test suites) is created and then evaluated to show that it solves some yet unsolved problem or it solves it more effectively than earlier approaches. The use of this method will help to answer research questions 5 and 6.

In (Hevner, 2004), a framework is presented which addresses design science research in information systems. However, the applicability of this framework exceeds the domain of information systems. According to the framework, two forces guide the research: *business needs* and *applicable knowledge*. The latter includes such foundations and methodologies as theories, models, methods, data analysis techniques, measures. The research itself comprises two interconnected phases: the phase of *development*, where new constructs are created, and the phase of *evaluation* of the construct. Typically, evaluation follows development, but then the research process can continue with further development and further evaluation. When new knowledge is created in the process, it is added to the body of knowledge in the field.

In our case, the necessity to ensure the quality of the control software can be viewed as an example of a business need. The employed knowledge will be reviewed in the following sections of the thesis: the IEC 61499 standard, concepts and approaches related to software testing and test generation, and the evolutionary methods. The phases of the design research which will be considered in the following sections are as follows:

1. During the phase of development, a new method of automated test generation for FBs will be created. This phase will be split into two sub-phases. First, after a review of existing software, a method will be constructed based on third-party tools: the first tool will reduce the problem to the more general problem of source coverage test generation, and the second tool will solve this more general problem. Second, an approach with more novelty will be developed which will exceed the limitations of the first approach.
2. During the phase of evaluation, the proposed methods will be tested on a number of instances, which will be selected from several systems under test. Such systems are obtained based on a literature review. The performance (i.e. obtained coverage percentage) and the execution time of the methods will be measured and compared between each other. These activities will involve empirical research, as dependent variables (performance, run time and test suite size) will be measured and the results of the measurements will be analyzed.
3. This thesis and the conference paper accepted to the INDIN'2015 conference (Buzhinsky, Ulyantsev, Veijalainen & Vyatkin, 2015) will report the results into the body of knowledge in the FB testing field.

3 THE IEC 61499 STANDARD

The IEC 61499 (IEC, 2012) is an open standard for distributed control and automation which was introduced in 2005 by The International Electrotechnical Commission (IEC). This standard is based on its predecessor, IEC 1131. The purpose of its introduction was to allow the development of distributed control systems, which can be allocated into many programmable logic controllers (PLCs), with robust, reusable modules. Nowadays, the standard is attempted to be used in production. An example is its application in shoe manufacturing (Colla, Brusaferri & Carpanzano, 2006). However, its application faces several challenges (Thramboulidis, 2006; Hall, Staron & Zoitl, 2007), namely, unfamiliarity of practitioners with the semantics of the standard and the inability of the standard to address the whole development process (e.g. it does not address requirement elicitation). Nowadays, several tools support the development of control systems represented in the standard. They include *ISaGRAF* (Vyatkin & Chouinard, 2008), *NxtStudio* (nxtControl, 2014), and *FBDK* (Vyatkin etc., 2008).

The IEC 61499 standard suggests viewing a control application as a number of *function blocks* (FBs), either basic or composite ones, which are interconnected to form a network. The concept of function block will be explained in more detail in the following subsections. When developed, function blocks are usually represented in the XML format.

3.1 Function blocks

An FB is an entity with a defined interface which can encapsulate both behavior and state. Thus, FBs and their instances are close to the concepts of classes and objects in object-oriented programming (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, 1991); however, they do neither support inheritance nor polymorphism. There are two main types of FBs, basic and composite FBs, which will be described in the following subsections.

First, let us define an *FB interface* which is present in both FB types. In this thesis, an FB interface is an octuple (E_I, V_I, D_I, E_O, V_O, D_O, M_I, M_O) where E_I is the set of *input event* types such that each event instance fired during the execution of the system belongs to one of these types, V_I is the set of *input variables*, $D_I = \{ D_1, \ldots, D_{|V_I|} \}$ is the set of input variable domains (domains are finite and directly correspond to variable types typical for general-purpose programming language: BOOL, INT, REAL, TIME, ARRAY, etc., and are described in the standard), E_O is the set of *output event* types (their instances are also referred to as *output actions*), and V_O and D_O are the sets of output variables and their domains.

Finally, M_I and M_O are Boolean (i.e. consisting of zeros and ones) matrices with sizes $|E_I| \times |V_I|$ and $|E_O| \times |V_O|$ respectively that define which events are associated with which variables. Each step of an FB execution is triggered by one of its input events. Multiple events cannot occur simultaneously: they will always be processed sequentially. An association between an input event and an input variable means that when the event is received, the corresponding input variable is updated with the value which originates from another FB (in particular, events and variable values may originate in *service interface FBs* which may model the plant's sensors). Next, output events can be generated and output data can be updated during FB execution steps. If an output event is associated with some output variable, then the new value of the output variable becomes available for other FBs (or for the plant's actuators which also can be modeled as service interface FBs), when the event is generated. A more precise definition of association is given in Section 3.3.

A scheme of an FB is presented in Fig. 1, and an example of a concrete FB is shown in Fig. 2. This FB has three input event types (namely, E1, E2 and E3), two input variables (a Boolean variable BOOL_VAR and an integer variable INT_VAR), one output event (O1), and one Boolean variable OV. Furthermore, the input event E2 is connected with both input variables, the input event E3 is connected with BOOL_VAR, and the output event type O1 is connected with output variable OV. It is visible from the figures that FBs are typically represented in the "head and body" graphical notation, where event connections are attached to the head, and data connections are attached to the body.

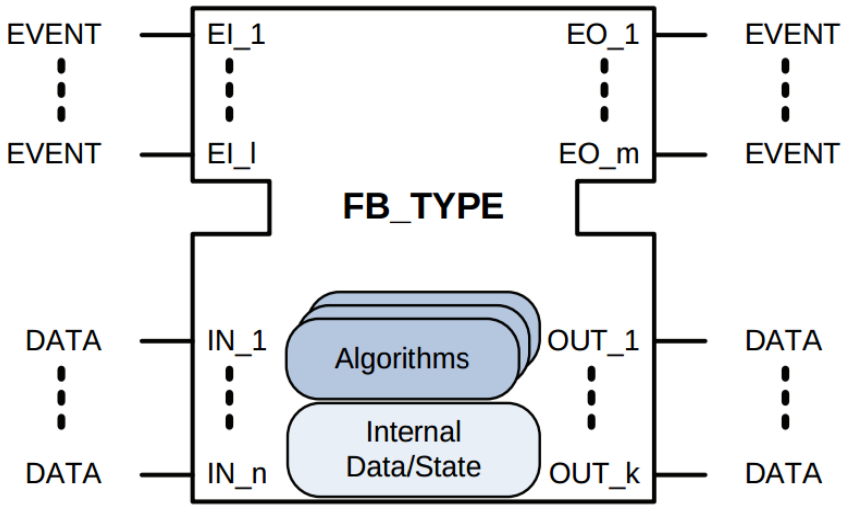

FIGURE 1 A general scheme of an FB

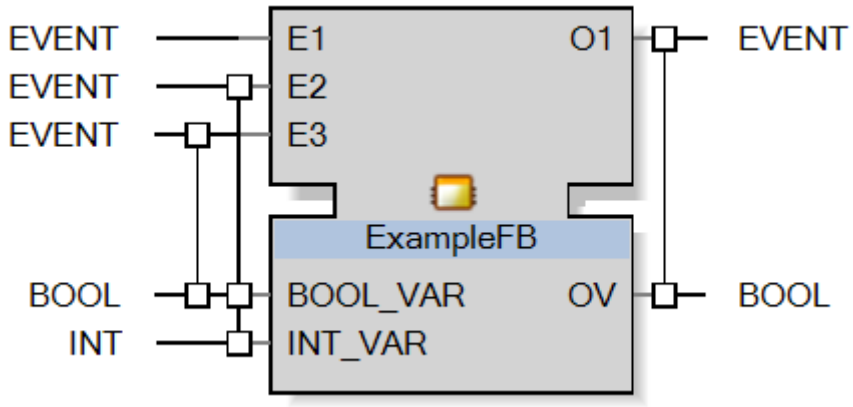

FIGURE 2 An example of a concrete FB

3.2 Basic function blocks

According to the standard, basic FBs are implemented using the concept of *execution control charts* (ECCs), which are also referred to as *finite-state machines* (FSMs). Formally, a basic FB is a octuple (I, V, D, S, s_0, δ, λ, a) where I is the FB interface, V and D are the sets of internal variables and their domains (internal variables are separate from input and output ones), S is the set of *states* (only one state is active at each moment), s_0 is the *start state*, $\delta : S \times E \times D_1 \times \ldots \times D_{|V|} \rightarrow S$ is the *transition function*, which determines the new state when an event is received, $\lambda : S \rightarrow 2^{E_O}$ is the *output function*, which determines the output events for each state, and $a : S \rightarrow L$ is the *algorithm function*, which defines an algorithm (in the *Structured Text* language L, which is based on *Pascal*) to be executed when a state becomes active. Algorithms can operate with all three kinds of variables: input, output or internal ones. In

particular, they are the only means of updating internal variables and moving data between input, internal and output variables.

All variables inside a basic FB are assumed to have default values (e.g. false for the type BOOL), which means that transitions with unassociated events and variables in guard conditions are possible.

ECCs are graphical diagrams for basic FBs. In them, 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 input variables V_I, which is reflected in the comprehensive domain of the transition function δ. The choice of transitions to be executed when an event is received is always deterministic: situations when several transitions can execute are arbitrated by priorities. It is also 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), a state change and a change of variables. 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 of an FB is shown in Fig. 3. This ECC is compatible with the interface shown in Fig. 2, 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.

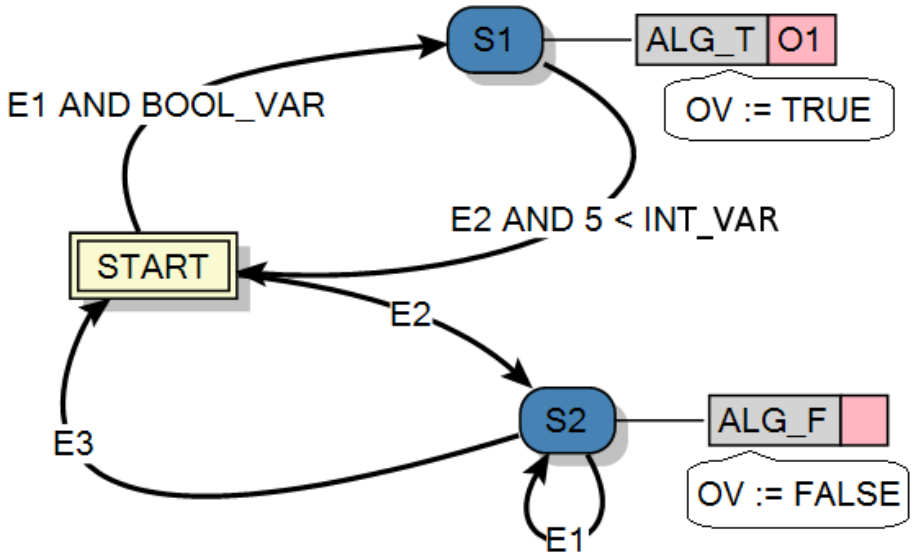

FIGURE 3 An example of an ECC in a basic FB

3.3 Composite function blocks

Inside a *composite FB* there is a network of FBs of other types with event and data connections between them. A composite FB is a quintuple (I, B, C_E, C_D, P) where I is the FB interface, B is the set of *nested FBs*, C_E and C_D are the sets of *event and data connections*, and P is the set of predefined input variable values of nested FBs. Each connection joins outputs and inputs (either events or variables) of nested FBs to each other, or, possibly, to inputs and outputs of the composite FB being defined. Predefined variable values are useful when no input connection is associated with a particular input variable of a nested FB. An example of a composite FB is shown in Fig. 4 (this is a screenshot made in *NxtStudio*). Its interface is visible at the left and at the right of the figure.

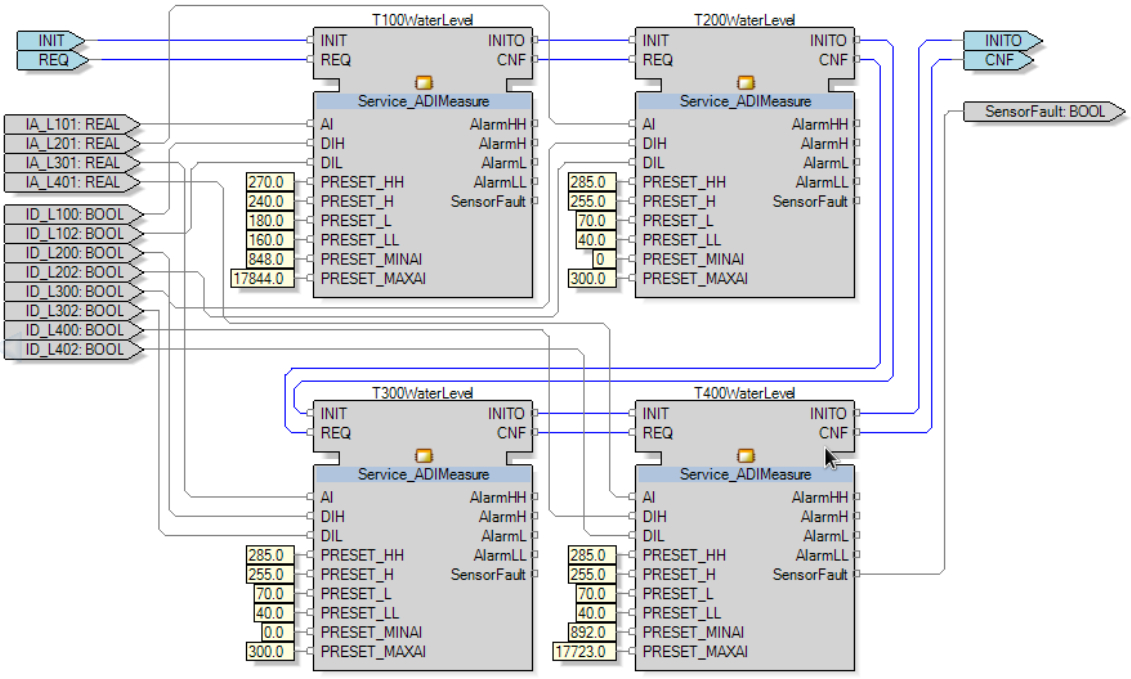

FIGURE 4 An example of a composite FB

Composite FBs are convenient abstractions, since they allow reusing various particular arrangements of FBs of lower levels. Moreover, there is no requirement to deploy all nested FBs inside a composite FB to a single device. Thus, composite FBs represent a unified way of representing both distributed and centralized control systems.

When an event is received by a composite FB, it is propagated to the inner FBs it is linked to. This event triggers the execution of these FBs, which, in turn, can generate new events. The standard assumes that the events inside composite FBs are propagated in the breadth-first manner (Cormen etc., 2001). We now describe the semantics of event-variable associations more precisely. Assume that output event e_1 is associated with output variable v_1 in FB fb_1, and input event e_2 is associated with input variable v_2 in FB fb_2. If e_1 is fired by fb_1,

and then e_2 is received by fb_2 after some time or immediately after this (e_2 is not obliged to originate in fb_1 or even be of the same event type: it is possible for an FB to accept an event from one FB and read variable values prepared by another FB), then v_2 will be updated with the value of v_1 at the moment of firing e_1. If either of the associations is missing, the update will not take place. However, some implementations of the standard violate the described principles. For instance, the implementation of the standard from *FBDK* assumes that events inside composite FBs are propagated in the depth-first manner (Cormen etc., 2001) and that all changes in the output variables become immediately available to FBs which receive their values as input ones, ignoring possible absences of event-variable associations.

3.4 Service interface function blocks

Until now, we have not considered any interaction of the system of FBs with the underlying devices. *Service interface FBs* are responsible for such activities. They represent low level services provided by either software or hardware of the devices.

The behavior of service interface FBs is difficult to consider during automated test generation: to do so, one needs to have formal models of the devices. However, as service interface FBs typically perform I/O operations, they can be excluded from consideration and replaced by the inputs and outputs of composite FBs. This can be done in the following way. Assume that there is a service interface FB in the top-level FB of the system. Each its input and output interface element (event or variable) will be transformed to an output or input interface element of the top-level FB, respectively. Then the service interface FB will be removed. It will further be assumed that basic and composite FBs are the only types of FBs during the design of the test generation method. Service interface FBs may contain errors too, though, so leaving them out of scope of the thesis is one of the limitations of this study.

4 SOFTWARE TESTING AUTOMATION

As mentioned in the introduction, software testing is one of the ways to ensure its reliability. Many types of software testing are known: unit testing, functional testing, integration testing, load and stress testing, and so on (IEEE, 2013). Manual creation of test cases can be hard, as the developer or the tester must manually check different paths of software execution. Thus, to reduce the cost of testing and to further improve the reliability of this process, test automation, and, in particular, automation of input data generation can be considered (Edvardsson, 1999).

In particular, we are interested in functional and unit testing of control applications for industrial automation systems. For such systems, an especially important testing stage is factory acceptance testing, which is the first integration test of the software (Peltola, Sierla, Aarnio & Koskinen, 2013). Another approach significant for such systems is loop checking, which "verifies the I/O connectivity, control strategy and safety aspects of the control loop application against the specifications" (Peltola etc., 2013). Furthermore, interviews conducted in (Peltola etc., 2013) have also shown that coverage analysis and source-based test generation is one of the target areas of functional software testing. This area is very close to the one which will be dealt with in the thesis.

Various approaches of software testing automation will be explored in this section. First, a broad field of model-based testing (Broy etc., 2005, pp. 281–387), heavily connected with the use of finite-state machines, will be reviewed. Second, several other approaches, including constraint based and evolutionary approaches (the latter will be adopted in the thesis), will be considered. The section will finish with a description of testing formalism which will be employed in the thesis.

4.1 Model-based testing

The general idea of *model-based testing* (MBT) (Broy etc., 2005, pp. 281–291) is to employ formal models of software, which can be obtained from the requirements, to analyze the systems and to generate test suites which can show conformance of software to its specification. For reactive systems, on which we focus in this thesis, finite-state machines (FSMs) can serve as such models. In MBT, *coverage* of models is usually an important goal to achieve. However, the topic of the thesis deals with the test coverage of software implementation. Nevertheless, since FSMs are one of the key entities of the IEC 61499 standard, many ideas from MBT might facilitate the design of a coverage test generation method. In this section, several approaches to coverage test suite generation and coverage criteria known from MBT will be reviewed.

4.1.1 Coverage criteria

Coverage criteria of test suites are used to assess their adequacy. Many criteria can be viewed either in the strict way, or in percentage values. According to (Broy etc., 2005, pp. 295–297), there are three types of coverage criteria: structural, functional and stochastic ones. *Structural criteria* are based on the structure of the software model – in our case, on the structure of FSMs:

- *State coverage* requires that all the states are visited during test execution.
- *Transition coverage* assumes that all the transitions of the finite-state specification are covered.
- *Boundary interior coverage* demands all loops to be covered a certain number of times.
- *Path coverage*, the strongest structural criterion and the hardest one to achieve, designates that each path in the specification is covered by at least one test case.
- *Round-trip coverage* (Binder, 2000) requires that all transition sequences which begin and end in the same state (e.g. the initial one), or round-trip paths, are covered. More precisely, every transition and every loop on each round-trip path must be exercised at least once.

Functional coverage criteria assume that some model of environment is available together with the specification. This model specifies several possible scenarios of system behavior and thus restricts test cases by them. These scenarios are used to obtain the expected outputs of the implementation, while the inputs originate from the specification.

The final type of coverage criteria is represented by *stochastic criteria*. They are based on the probabilities of entering different parts of the specification, which are calculated according to the user's behavior. In the simplest case,

when all the transitions of the model are equiprobable, test case selection is performed randomly.

In the context of white-box test generation approaches, another subdivision of coverage criteria into *control flow oriented* and *data flow oriented* is also considered. Control flow oriented coverage criteria are usually defined in terms of decisions and conditions, which are typically expressed as if-then-else constructs:

- *Decision coverage*, or branch coverage, requires that each outcome of each decision in the specification is covered. For example, if there is an 'if' decision, then both 'then' and 'else' branches must be covered by some test cases in the test suite (it is possible that both branches are covered in a single test case).
- *Condition coverage* demands the coverage of both outcomes of each condition inside each decision. For example, if an 'if' decision is represented as an 'and' operation of some conditions A, B, and C, then each outcome of A, B, and C must be triggered by at least one test case in the same test suite.
- *Decision condition coverage* requires both decision and condition coverage, achieved by the same test suite.
- Finally, *multiple condition coverage* states that each combination of condition outcomes is reached. For example, if there is a composite decision (A and B) or (C and D) in an 'if' or 'while' clause, all 16 value combinations for A, B, C, and D should be tested.

Data-based coverage criteria, in contrast to control flow oriented ones, are defined in terms of a flow graph, which represents the program (e.g. a compiled version of a model) as a set of linear computations (nodes of the graph) and decisions, which transfer control between the nodes, and in terms of paths in this graph. The general idea is to follow variables from the points of their definition to the points where they are used. In this review, we will omit concrete data-based criteria, as their descriptions require a large amount of definitions to be done beforehand.

Another known technique is *partition based testing* (Gutjahr, 1999). This technique suggests splitting the input domain (e.g. possible values of some input variable) into several subdomains. Such divisions can be obtained from conditions, and the requirement is to have a test in a test suite that involves at least one input value from each of the subdomains. This makes partition based approaches similar to the ones which employ structural coverage criteria, which were reviewed before.

4.1.2 Test case generation techniques

According to (Broy etc., 2005, pp. 323–324), three main test generation approaches are known: theorem proving, symbolic execution and model

checking. *Theorem proving* deals with partitioning the software model into several equivalence classes, such that for each equivalence class, any test case in this class is assumed to check the presence of the same error, typical of this equivalence class. Once the equivalence classes are identified, each of them is viewed as a single test case. The partition is based on a formal specification of the software, and the number of equivalence classes can vary. In particular, it can depend on the size of the specification. For example, Helke, Neustupny & Santen (1997) transform predicates from the specification into a disjunctive normal form (DNF), and the number of equivalence classes is equal to the number of obtained disjuncts.

The second technique is *symbolic execution*, which is actually a software verification approach. It suggests replacing the inputs of the system with symbols (variables and constraints over them) and thus can handle unbounded values. Symbolic execution is applicable for both models and code. An example of the application of this approach is the work in (von Styp & Yu, 2013).

There is also a number of techniques which combine symbolic execution with constrain solving, including the ones that use symbolic and concrete execution together. A survey of such techniques can be found in (Cadar & Sen, 2013). Symbolic methods of this kind 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.

The final approach is *model checking* (Clarke, Grumberg & Peled, 1999), which is again a verification method. One of the types of model checking involves testing a model of a system against its temporal specification. In case of failure, model checking algorithms generate counterexamples, which explain why the model does not conform to the specification. To apply model checking for MBT, test specifications are expressed as temporal properties, and the problem of test generation is simply reduced to the identification of counterexamples for these specifications. For example, this approach is taken in (Enoiu, Sundmark & Pettersson, 2013).

4.2 Other testing automation techniques

One of the first approaches to automated test generation was the one introduced by DeMilli & Offutt (1991). The technique presented in the paper is based on the constraint satisfaction problem 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 generated as mutations of the original program under test. Algebraic constraints are generated and then solved in order to ensure the failure of mutated programs.

Edvardsson (1999) separates test data generation methods into three types. The first and the simplest type is *random testing*: it just suggests randomly generating input test data for the program unit under test, 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 breaks the assertions. The final approach, *path-oriented test data generation*, is the strongest one. In this approach, test generator attempted to follow specific paths.

In (Hussain & Frey, 2006), a UML-based unit test case generation method is presented specifically for the IEC 61499 standard. This method complements the whole development process also proposed in (Hussain & Frey, 2006). Both state and activity UML diagrams, which represent software specification on different levels of abstraction, are subject to test generation. Round-trip path coverage is attempted to be reached for state diagrams, because it can disclose missing event/action pairs. To do this, test cases are generated from finite transition trees constructed from each of the state diagrams. As for activity diagrams, they are assumed to represent the functionality of basic FBs. Test cases generated from them cover particular paths and are obtained from control flow graphs. To enforce the execution of such paths, certain internal variables are set to specific values, and certain input events are activated.

In (Peltola etc., 2013), MBT is augmented with simplified model creation, which is supported by code generation from source information stored in the CAEX format (IEC, 2008). This format is applicable for storing various hierarchical objects and supports object-oriented concepts. In this case, it is used, for example, to store information about a control loop within the system. The suggested approach is applied to a system under test represented in the using the IEC 61131-3 notation (IEC, 2003). In this study, Conformiq Designer (Conformiq, 2014) is used for both creating MBT models (which contain state diagrams and some additional information) and test generation. The coverage results of the obtained tests are encouraging.

A complex, combined approach to the test generation problem is taken in (Fraser & Arcuri, 2011), 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. The approach is based on evolutionary search (see Section 5) 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 was already mentioned when describing the constraint-based approach (DeMilli etc., 1991). These assertions can be manually checked for semantic correctness by the developer. The successful results of *EvoSuite* reflect the words from (Edvardsson, 1999): "The most promising search methods seems to be

simulated annealing and genetic algorithms for their data type independence and iterative relaxation for its predictability."

4.3 Tests, test suites and coverage criteria for the considered problem

In this subsection, some theory about tests, which will be used in the thesis, is explained, and the problem which will be dealt with in it is stated. To define a test, we first fix an FB. Assume that it has events E_1, …, E_n and input variables V_1, …, V_m with domains D_1, …, D_m, where domains represent possible values of particular data types. Next, Boolean values W_{ij} signify whether the event E_i is associated with the variable V_j. In addition, consider an 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 $(E_i, a_1, \ldots, a_m)$, where a_j, $j = 1..m$ is either from D_j, if W_{ij}, or is $\perp$ 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 single execution step. Also note that since multiple events cannot arrive simultaneously, there is only one event in the tuple.

A *test case*, or *test* for short, 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, one step per input tuple, between which the FB persists its state. It is also assumed that 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. 2 is shown in Table 1.

TABLE 1 An example of a test with length 4

Tuple index	E_i	a_1 **(BOOL_VAR)**	a_1 **(INT_VAR)**
1	E3	true	$\perp$
2	E1	$\perp$	$\perp$
3	E2	false	–100
4	E2	false	42

Finally, a *test suite* is a finite set of tests. The purpose of considering test suites as objects subject to optimization is that many tests can be required to achieve full coverage of a software system.

We are now ready to define some coverage measures of basis FBs, which are based on the measures reviewed previously:

- *Transition coverage* is the share of all transitions inside the ECC of the FB which are executed at least once when all the tests are executed.

- *n-transition coverage* is the share of executed *n*-tuples of consequent transitions of the ECC. For example, for $n = 2$ this means the share of all transition pairs.
- *Branch coverage* is the branch coverage of the source code (see section 4.1.1) which represents the FB. In this case, it is assumed that the source is obtained from the FB using some deterministic transformation.

As for coverage measures for composite FBs, they might be calculated as integrated measures of the inner basic FBs. One might count either instances or instance types inside a composite FB (we will further consider the coverage of instance types). In addition, one can also measure the number of visited event and data connections.

At this point, we are ready to answer three of the six research questions stated in the introduction. First, we have identified that the features of the IEC 61499 standard which are relevant for test suite generation are the interfaces of FB and the internal finite-state structure of basic FBs (this answers research question 1). The interface of an FB, or, more precisely, its input interface, which consists of input events and variables, defines the form of the test elements – input tuples. In turn, ECCs inside basic FBs can be employed to define coverage criteria, such as transition coverage. Second, research question 2 about the representation of tests and test suites for IEC 61499 applications has just been answered in this subsection. Next, in Section 4.2 we have reviewed several existing test data generation techniques and, in particular, the evolutionary approach taken in (Fraser & Arcuri, 2011). This approach will be considered in more detail in Section 5. Thus, we have answered research question 3 and will answer research question 4 in the next section.

Based on the formalism presented in this subsection and on the knowledge reviewed previously, the problem of the research can be defined: design a method which generates test suites and maximizes one of the coverage criteria for a given FB (either basic or composite). Further we will use transition coverage, *n*-transition coverage and branch coverage as coverage criteria. Among them, transition coverage is the one widely used in MBT, and it employs a specific feature of our problem – finite-state structure of basic FBs. *N*-transition coverage is not so popular, but it is an example of a more complex coverage measure. Eventually, branch coverage is widely applied in software engineering and, unlike other considered coverage criteria, requires that all parts of algorithms inside basic FBs are covered. To design the test generation method, we will take the evolutionary approach.

5 EVOLUTIONARY COMPUTATION

In this section, the concept of evolutionary computation will be presented. Several simple (e.g. the random mutation hill climber (Mitchell, Holland & Forrest, 1994)) and more complex (e.g. the genetic algorithm (Koza, 1992)) algorithms will be presented. The presented algorithms might be used within the test generation methods which will be proposed in the thesis.

Evolutionary and genetic algorithms are general optimization methods which are applicable for various discrete and continuous problems. Problems, for which evolutionary algorithms are applied, are usually not solvable in polynomial time by precise algorithms (unless P equals NP). Such problems include, for example, the travelling salesperson problem (Larrañaga, Kuijpers, Murga, Inza & Dizdarevic, 1999) and the job shop problem (Della Croce, Tadei & Volta, 1995). Evolutionary algorithms usually do not guarantee that an 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 initial 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 (we further assume that the aim is to maximize the fitness function), so that the worse individuals are discarded, and the best ones are retained. This procedure is known as *selection*, and its exact implementation (e.g. how to discard individuals, how many individuals to discard) varies among different evolutionary algorithms.

Many concrete techniques exist that employ evolutionary ideas. In the following subsections, basic evolutionary operators (mutation, crossover, and selection) and several concrete evolutionary methods will be reviewed.

5.1 Evolutionary operators

Throughout the review of the evolutionary operators, we will consider two optimization problems. The first one, the OneMax problem (Schaffer & Eshelman, 1991), is very simple and well-known in the literature. In this problem, a certain bit string of length n should be guessed, which corresponds to the maximal value of the fitness function. For simplicity, it is often assumed that this string is formed of n ones. In this case, the fitness function of a bit string is equal to the number of ones in this string, and the goal is to identify the n-one string being guided by fitness function values. No prior knowledge of the target string is assumed. The second considered problem is the problem stated in this research: find a test suite with a high value of a chosen coverage criterion. The selected coverage criterion is used as the fitness function.

5.1.1 Mutation

The idea of the mutation operator is to apply a small change to an individual. This operator receives an individual and, assuming the availability of some source of randomness, produces a new individual. The following mutations are typically used for the OneMax problem:

- Flip a bit on a random position.
- For each position, flip a bit at this position with the probability $1/n$.

As for the coverage test generation problem, possible mutations are:

- Select a random test in the test suite, select a random position in it, randomly replace an event at this position, and randomly generate input data for this event.
- Select a random test in the test suite, select a random input data value, and randomly generate a new value.
- Select a random test in the test suite, select a random input data value, and adjust this value (e.g. for integer variables, either add or subtract a small number).

5.1.2 Crossover

The crossover operator uses two individuals to generate one or two new individuals. Similarly to the mutation operator, it needs to access some source of randomness. For OneMax and for string optimization problems in general, several typical crossover operators are known:

- The *single-point crossover* operator selects a random position and exchanges the portions of the strings after this position.

- The *two-point crossover* operator selects two random positions and exchanges the portions of the strings between them.
- The *uniform crossover* operator exchanges strings on each position independently with some probability (often ½).

The described crossover operators are illustrated in Fig. 5, where the exchanged parts of bit strings before and after the transformations are shown in blue.

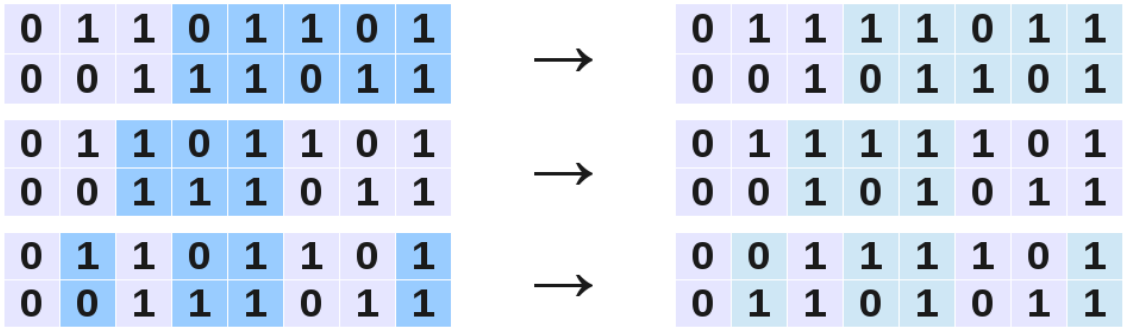

FIGURE 5 Examples of three crossover operators: single-point (top), two-point (middle), and uniform (bottom)

Similar ideas can be applied for the test suite optimization problem, where an individual is a test suite. Possible ideas of crossover between test suites include:

- Exchange two different test suites on the test basis according to one of the three crossover types.
- Select two random tests from both test suites and exchange them on the event basis according to one of the three crossover types.

5.1.3 Selection

The selection operator is typically applied in algorithms which operate with many individuals in each time, like the genetic algorithm, which will be reviewed further. In this case, the problem is to retain a certain number of individuals while discarding the others. Possible options for the selection operator include:

- Sort the individuals according to their fitness values and select the required number of best ones. This technique is the simplest one.
- *Tournament selection*: for each individual to choose, select several random individuals (often 2) and then select the best one among them.
- *Roulette selection*: align the individuals on the $[0, 1]$ segment with the lengths proportional to their fitness values, and then select the required

number of individuals by uniformly drawing points from the segment. This process is illustrated in Fig. 6.

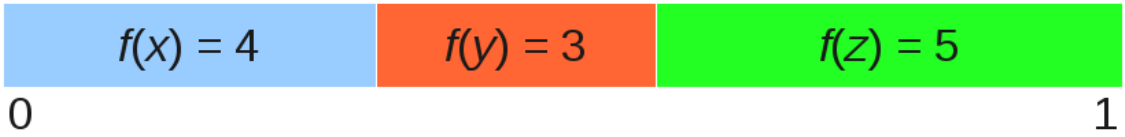

FIGURE 6 The [0, 1] segment, the random choice of point on which determines the selected individual (x, y, or z) in roulette selection

5.2 Evolutionary algorithms

Several evolutionary algorithms will be presented in this subsection. We will start from the trivial random mutation hill climber, and continue with the genetic algorithm (GA) and multi-objective optimization. A more thorough survey of evolutionary algorithms and metaheuristics (algorithms from a more general class of optimization techniques) can be found in (Boussaïd, Lepagnot & Siarry, 2013).

5.2.1 Random mutation hill climber

The random mutation hill climber (RMHC) (Mitchell etc., 1994) is a simple evolutionary algorithm. It stores only one individual, the current solution, in memory. First, an initial solution x is randomly generated (e.g. a random bit string in case of OneMax). Then, until the stopping criterion is reached, the following actions are iterated: a new solution y is generated as a mutation of x, and, if the fitness value $f(y) \geq f(x)$, then x is replaced by y. The following stopping criteria are often used:

- A certain number of iterations are executed.
- A certain fitness value is reached.
- There has been no fitness improvement during a certain number of iteration (so-called *stagnation*).

5.2.2 Genetic algorithm

The genetic algorithm (Koza, 1992), or simply GA, is an algorithm which stores multiple individuals (circa 100) at the same time. This pool of individuals is called a *generation*. Initially, the generation is comprised of randomly generated individuals. After that, on each iteration some individuals are subject to crossover and mutation, and then a new generation of the same size is selected from both the old and the new individuals. The scheme of a single iteration is shown in Fig. 7.

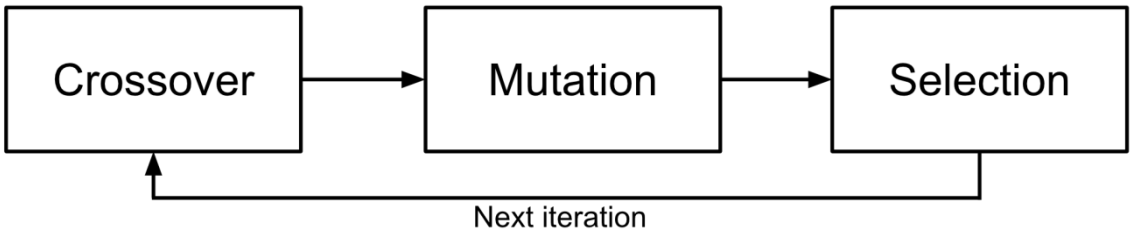

FIGURE 7 The general scheme of an iteration of the genetic algorithm

Many variations of the GA exist. They include, for example, the island GA, where there are several generations on several computational devices with subtle migration between them, and the steady-state GA, where each iteration is performed on just two individuals.

5.2.3 Multi-objective optimization

Multi-objective optimization (Deb, 2001) is different from the classical evolutionary methods in the way that it aims to optimize several criteria simultaneously. This approach might be reasonable for the problem of coverage test generation, because it would allow considering several coverage criteria in one run. The size of the test suite might be also considered as an additional criterion to minimize, because, among two test suites with the same coverage, the smaller one is often more beneficial.

Let $f_1, \dots, f_n$ be the criteria to maximize. A solution x is dominated by a solution y, if $f_i(y) \geq f_i(x)$ for all i, and $f_i(y) > f_i(x)$ for at least one i. If neither x dominates y nor y dominates x, then these individuals are incomparable. Multi-objective algorithms, like NSGA-II (Srinivas & Deb, 1994), often involve an approximation of the so-called *Pareto frontier*, which is the inclusion maximal set of solutions not dominated by each other. The quality of the approximation, which can be measured as the difference between the hypervolumes (Deb, 2001, p. 332) of the optimal frontier and the found one, is usually improved during the algorithm's execution. Multi-objective evolutionary algorithms are actively developed presently. A survey of recent multi-objective algorithms in this field can be found in (Zhou, Qu, Li, Zhao, Suganthan & Zhang, 2011).

6 APPROACH BASED ON THIRD-PARTY TOOLS

Now, having finished reviewing the literature and answering research questions 1–3 (in the end of Section 4) and 4 (in Section 5), we are finally ready to start to construct test generation methods. One of them will be presented in this section, another one in Section 7, and Section 8 will deal with their evaluation. Thus, the remaining research questions 5 and 6 will be resolved.

The first 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. The approach employs two third-party tools: *FBDK* (http://www.holobloc.com/doc/fbdk/) and *EvoSuite* (Fraser & Arcuri, 2011) and supports the optimization of branch and transition coverage. The approach is summarized in Fig. 8. The input of the test generation method is an .fbt XML file which describes the FB under test. Such files can be created using development environments such as *FBDK* or *NxtStudio* (nxtControl, 2014). If this FB is composite, XML descriptions of the nested FBs should also be available. The method comprises three stages, which are described below. The first two stages of the method were implemented in *Java*, and a bash script was written for the third stage. After describing the stages of the method, we discuss its limitations.

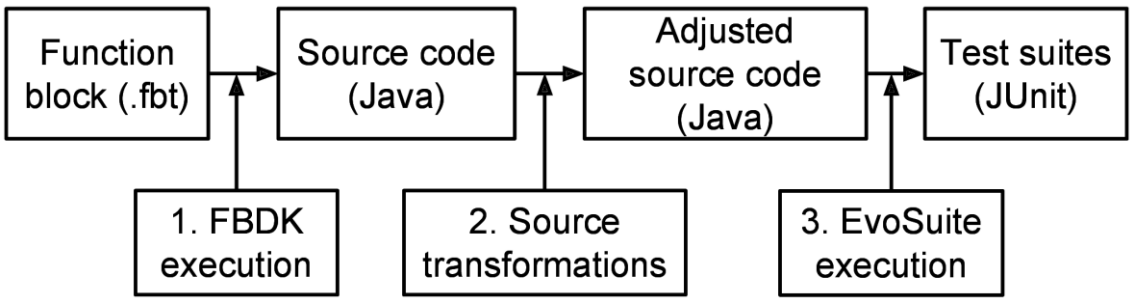

FIGURE 8 The scheme of the approach based on third-party tools

6.1 The first stage

A third-party tool, *FBDK*, transforms the .fbt description of the FB under test to a *Java* source file consisting of a single *Java* class. For a basic FB, it creates a *Java* 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*.

6.2 The second stage

On the second stage, the obtained source code is transformed to prepare it for evolutionary test generation, which will be done by another tool. It is important that these transformations must not alter the behavior of the FB. 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 mentioned above. 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 and, due to its emptiness, does not change the behavior of FBs. The purpose of these methods is to allow test generation which optimizes transition coverage (see the next stage): if all these methods are covered, then all transitions are covered, and vice versa. When branch coverage is selected to be optimized instead, such methods are not generated.

An example of code obtained from *FBDK* and transformed according to the rules described in this subsection, including additional dummy methods for transitions, is shown in Appendix 1. This code represents a composite FB `my_sensor2` from the PnP system, which will be described in Section 8.1. The interface and the FB network of this FB are presented in the beginning of the appendix.

6.3 The third stage

On the third stage, the modified source code is fed to *EvoSuite*, 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 GA is chosen. Depending on the coverage criterion employed, *EvoSuite* is configured to either generate tests to cover the whole class (in case of branch coverage), or to cover only the transition methods created in the end of the previous stage (in case of transition coverage). 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 1 as it would appear in the body of a single *JUnit* test:

```
@Test
public void test_0() {
    ExampleFB fb = new ExampleFB();
    fb.service_E3(true);
    fb.service_E1();
    fb.service_E2(false, -100);
    fb.service_E2(false, 42);
}
```

In this case, the *JUnit* test suite consisting of this single test case is an ordinary *Java* class with the single method `test_0()` inside. An example of an entire test suite for a composite FB is presented in Appendix 2. It comprises of two tests. This test suite was generated by *EvoSuite*, in response to the FB description from Appendix 1.

6.4 Limitations of the approach

The main limitation of the approach is the small number of supported coverage goals. It natively supports branch coverage, since this is the coverage measure optimized by *EvoSuite*. The method also supports transition coverage, which is implemented by method stub insertion into the *Java* code, so that the coverage of all these method is equivalent to the coverage of all transitions of all ECCs inside the FB under test. One might also implement state coverage in a similar way. Nonetheless, there are coverage measures which cannot be implemented by method insertion.

An example of such a measure is n-transition coverage for $n > 1$: in this case the objects subject to coverage are tuples of consequent transitions, and the coverage of each tuple requires the execution of several code segments in a

particular order. Hence, the method insertion trick is not applicable. Other examples of coverage criteria not supported by the approach are boundary interior coverage and path coverage. A possible way of addressing such coverage measures is to consider the order of executed code segments at runtime. However, the use of *FBDK* to translate FBs into *Java* does not allow this solution. In the next section, the facilitation of the translation implemented specifically for this thesis will allow us to handle such coverage criteria as *n*-transition coverage.

Another limitation, in case of branch coverage, is the presence of code branches which are always covered or are impossible to cover at all due to the technical artifacts of the FB translation to *Java*. An example of a code segment which is covered in every test is the constructor of the FB class. Next, consider the following example of a branch of *FBDK*-generated code which is impossible to cover:

```
public void service_INIT() {
    if ((eccState == index_START)) {
        state_INIT();
        transition_OR_2();
    }
}
```

This method determines the right transition to execute in case of the incoming event INIT. If we assume that the START state is only one in the ECC, then the condition `eccState == index_START` always holds, and thus the implicit 'else' branch of the conditional operator is always missed. This fact does not imply the fault in the FB, but the branch coverage of the *FBDK*-generated code of this FB will never become 100%.

Finally, while translating FBs into *Java*, *FBDK* assumes nested FBs of a composite FB are executed in the depth-first search order, while the IEC 61499 standard specifies the breadth-first search traversal. This means that the execution of composite FBs in *FBDK*-generated code is not truly equivalent to the behavior specified by the standard. Imagine a composite FB *fb*, such that several basic FBs $fb_1, \ldots, fb_n$ inside emit different events $e_1, \ldots, e_n$ when they are executed, and these events are connected to the event outputs of *fb*. Then, depending on the execution order of $fb_1, \ldots, fb_n$, *fb* will generate output events $e_1, \ldots, e_n$ in different order. As for the execution of basic FBs, such problems do not arise, and the *Java* code simply presents the behavior of an ECC explicitly.

Moreover, if the developer of FBs uses *FBDK*, then the problem does not arise even for composite FBs, because the behavior shown by *FBDK* is exactly the one demonstrated by the *FBDK*-generated *Java* code, and the transformations described in Section 6.2 do not alter its behavior. However, if one applies a different development tool (e.g. *NxtStudio*), it is recommended to check whether the generated tests are executed in the same way by the development tool and the *Java* code. One of the reasons for tools with different implementations of the standard to exist is that the IEC 61499 standard is

imprecise in some aspects, including the semantics of FB execution. Thus, one cannot speak about the equivalence of the behavior of a code to the behavior of the corresponding FB without specifying the concrete implementation of the standard.

7 APPROACH BASED ON INTERNAL TEST REPRESENTATION

The second test generation approach, which is presented in this section, is aimed to overcome some of the limitations of the approach based on third-party tools and to get better execution times by facilitating the domain knowledge. The scheme of the approach is presented in Fig. 9. The first stage of the method involves the translation of FBs into *Java* similar to the one performed by *FBDK*. However, it is more flexible and allows processing visited FB parts (e.g. transitions and states) at runtime, which widens the set of supported coverage criteria (see Section 6.4). The second stage of the method constitutes running a simple evolutionary algorithm, the random mutation hill climber (RMHC) mentioned in Section 5.2.1, to find a proper test suite for the given coverage criterion. In the remainder of this section we describe both stages of the method.

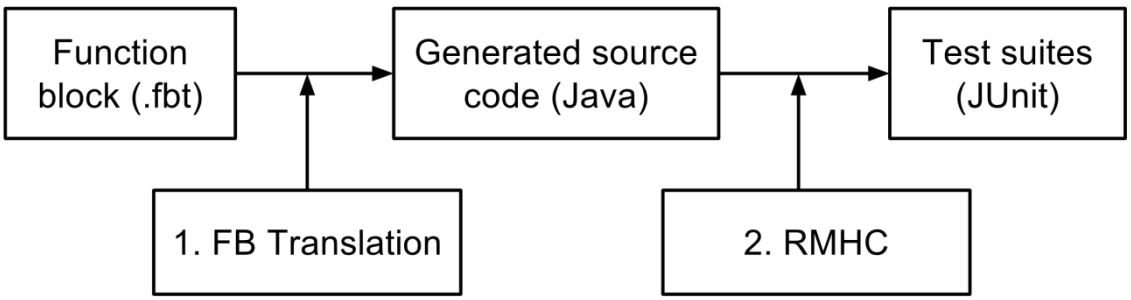

FIGURE 9 The scheme of the approach based on internal test representation

7.1 Function block translation into Java

In this subsection, instead of providing the complete description of the implemented FB translation to *Java*, which is mostly technical, we focus on several features of the translations which are important in the context of this thesis. These features are listed below.

- The translation explicitly schedules the execution of basic FBs inside a composite FB. The units subject to scheduling are only basic FBs: all boundaries of composite FBs, except the ones of the FB under test, are removed for simplicity. That is, if FB fb_2 was nested inside fb_1, and fb_3 was nested inside fb_2, then fb_2 will be removed and fb_3 will be directly nested inside fb_1. If there was a connection entering fb_2 and further going to fb_3, then after the boundary removal it will directly go to fb_3. The similar property holds for connection originating in fb_3 and escaping fb_2.
- It is possible to execute FBs both in the breadth-first and in the depth-first search order. The first strategy is specified by the IEC 61499 standard, and is used, for example, in the *NxtStudio* development environment, and the second one is assumed in *FBDK*. More generally, if one has an FB development tool with a particular implementation of the standard (which possibly violates the standard in some minor ways, like *FBDK* does), it is easy to modify the proposed approach to generate the code with the behavior representing the mentioned implementation.
- For a basic FB, the translation creates a class with state and variable declarations, algorithm methods and event processing methods, which accept relevant variable values as parameters. Thus, the translation is visually similar to the one performed by *FBDK*. To simplify the translation of algorithms, their code is still generated by *FBDK*. Classes for composite FBs embody all the classes for basic FBs they depend on as nested ones.
- While executing a transition, a special method is called which accepts the information about the transition (its source, destination and guard condition). This method, which is implemented outside the automatically generated code, calls a processing routing specific for the selected coverage criterion. For instance, for transition coverage this routine counts the number of unique executed transitions, and for branch coverage it does nothing, because this criterion depends only on covered pieces of code.

7.2 Implementation of the evolutionary algorithm

There are many options among evolutionary algorithms to choose from. The number of these options is far beyond the number of algorithms reviewed in Section 5.2. For simplicity, the random mutation hill climber was implemented. Implementations of algorithms which use generations with more than one individual (e.g. the genetic algorithm) were not considered due to the fact that it is always possible to construct a solution which is not worse than a given set of solutions. To do this, it is sufficient to merge all the tests from the solutions: this will ensure that the obtained composite test suite will cover all system parts covered by the initial test cases. It is also possible to minimize the composite

test suite (i.e. remove the tests which can be removed without the loss of coverage) to improve its sizes (smaller test suites are usually preferred to larger ones because they take less time to execute and are easier to comprehend). Thus, one individual in the generation is sufficient.

The evolution starts with a single test composed of a single randomly generated input tuple, and stops if the fitness value has not improved during the last 1000 mutations. Below we describe our implementation of fitness functions and our choice of the mutation operator.

7.2.1 Fitness functions

The employed fitness functions naturally correspond to chosen coverage criteria: branch, transition and 2-transition coverage. They additionally take into account the size of the test suite: for two test suites with identical coverage values, the shorter one is preferred.

The branch coverage fitness value is computed by the means of the *JaCoCo Java* code coverage library (http://www.eclemma.org/jacoco/). Before each fitness evaluation, it processes the .class *Java* byte code corresponding to the FB under test to insert information used in determining the branch coverage value. Unlike branch coverage, the evaluation of transition and 2-transition coverage is much simpler: the information about unique executed transitions and transition pairs is maintained during test execution as described in the end of Section 7.1. In this case, the special transition processing method calls a routing which updates the set of unique executed transitions or transition pairs.

Finally, each evaluated test suite is executed with the help of reflection, the feature of *Java* which permits the execution of methods when their names and parameters and unknown at compile time, but known at run time. An alternative, but much slower solution would be to compile each test suite before fitness function evaluation.

7.2.2 Mutation operator

The mutation operator is parameterized by three numbers: p_{rem} – the probability to try reducing the test suite, p_{adj} – the probability to try adjusting the test suite by adding or modifying input tuples, and N_{op}, the maximum number of operations which can be performed during mutation. The last parameter controls the strength of the mutation. In our study, we use $p_{\text{rem}} = 0.3$, $p_{\text{adj}} = 0.4$ and $N_{\text{op}} = 3$. The detailed description of the mutation of a test suite is listed below.

- **Removal mutation.** With the probability of p_{rem} either a random test or a random input tuple in a random test (each option with the probability of ½) is attempted to be removed from the test suite. In this case the new individual will be accepted by the hill climber, if its coverage value has not been decreased as a result of mutation. This mutation is aimed to

reduce the size of the test suite, which is naturally increased during the evolution.

- **Copy mutation.** If the removal mutation was not applied, with the probability if p_{adj} select a random test, copy it and perform a random number of adjustments of the test, which is uniformly distributed between 1 and N_{op}. Each adjustment is either a replacement of a random input tuple in the test by a randomly generated one, or an insertion of a new random generated input tuple into a random position within the test (each option with the probability of ½). For this mutation, it is crucial that the test is copied prior to the adjustment since this ensures than none of the coverage goals passed by the old test suite will be missed by the new one. However, the disadvantage of such mutations is the fast increase of the test suite size, which should be compensated by removal mutations.
- **Creation mutation.** If other mutations were not applied, generate a new test and insert it into the test suite. The length of the new test is uniformly distributed between 1 and N_{op}.

8 EXPERIMENTAL EVALUATION

This section describes the conducted experimental evaluation of the proposed test generation approaches applied to two sets of FBs and the obtained results. The first, obvious objective of the experimental evaluation is to evaluate the proposed test generation methods – in terms of achieved coverage values, execution time and size of the generated test suites. An additional objective is to compare both approaches.

8.1 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 (Patil, Vyatkin & Sorouri, 2012) to evaluate an approach to a different problem. The system consists of 31 basic and 17 composite FBs implemented in *FBDK*.

One of the hardware implementations of this device is shown in Fig. 10 as a screenshot made in *FBDK* (this tool can be used not only to develop FBs, but also to model the interaction of the control system with the model of the hardware). This screenshot 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 screenshot from *FBDK* in Fig. 11 shows the FB network of the FB `PnpCylinders`, which models the connections between the cylinders.

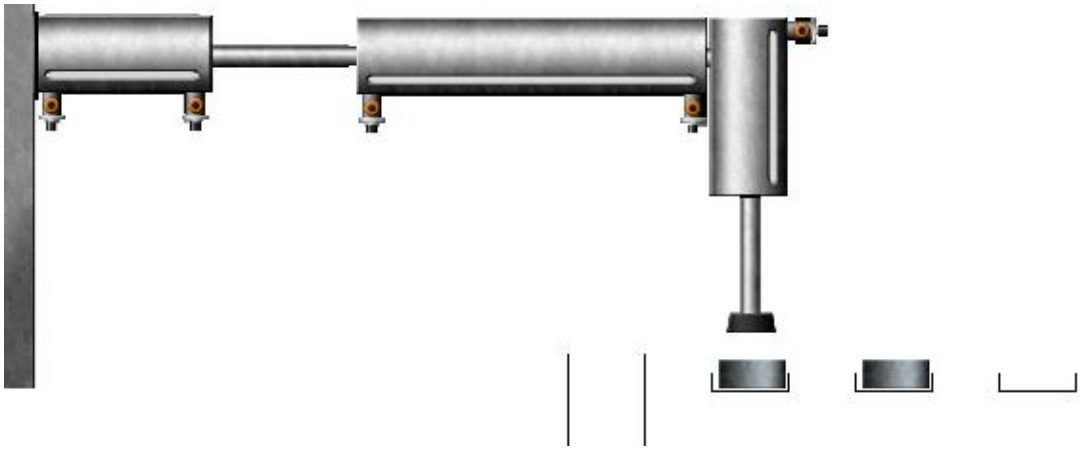

FIGURE 10 The scheme of one of the implementations of the pick-and-place manipulator

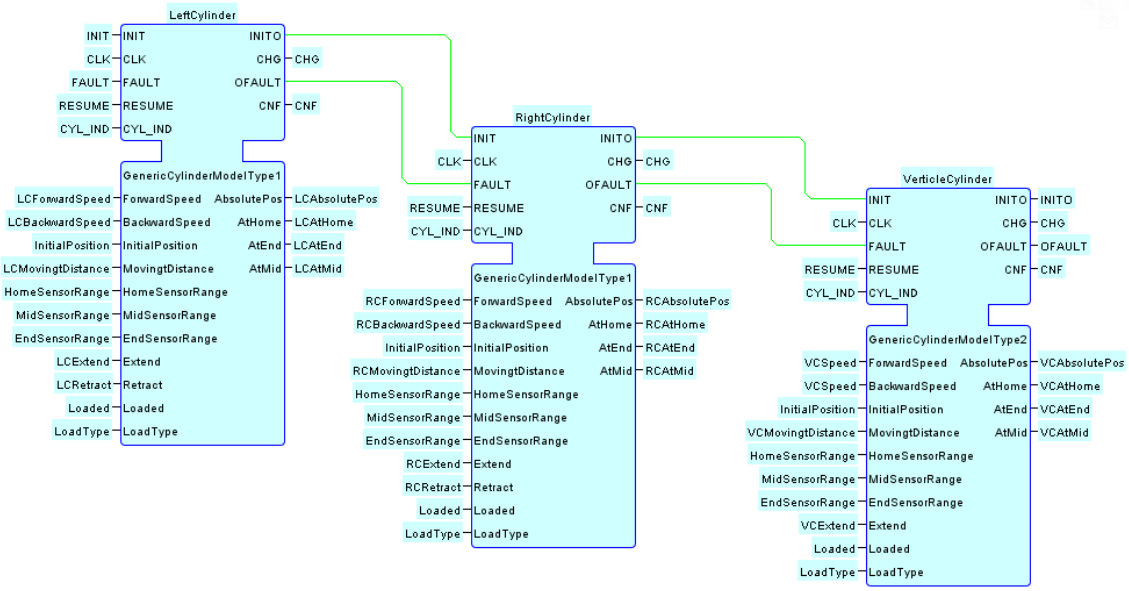

FIGURE 11 The FB network corresponding to the model of three connected cylinders shown in Fig. 10

The second system is the application which regulates a heat production plant (HPP) shown in Fig. 12. In (Peltola etc., 2013), an IEC 61131 application is mentioned, and the system we work with is the result of the redesign of this application to comply with the IEC 61499 standard. The *NxtStudio* software (nxtControl, 2014) 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, which represents the entire application. Twelve other FBs are basic.

FIGURE 12 Heat production plant

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 source size (i.e. more than 1000 lines of code) are typical for composite FBs, since the code for FBs on which the composite FB depends is included into the code of the composite FB.

8.2 Experiment setup

We separately evaluated the approach based on third-party tools (it is described in Section 6, below it will be referred to it as Approach 1) and the approach based on internal test representation (it was described in Section 7 and will be called Approach 2). The computation was performed on a PC with a 2.2 GHz *Intel Core i*7-2670*QM* CPU. For each FB, each considered coverage measure, a single run of each approach was performed. Branch and transition coverage were aimed to be maximized for both methods. Additional experiments for 2-

transition coverage were performed only with Approach 2, since Approach 1 does not support this coverage criterion.

We used different stopping criterions for different approaches. For Approach 1, a fixed time span was given for *EvoSuite* to find the solution. Note that the execution time of FB translation by *FBDK* and the time of the adjustment of the obtained code were not considered since they lasted less than a second. Ten minutes were available for *EvoSuite* to generate tests for basic FBs, and twenty minutes for composite ones. The time given to find a test suite for composite FBs was larger since the size of *Java* code generated for them was also larger. As for Approach 2, due to its manual implementation it was possible to use a better stopping criterion based on stagnation: the run was stopped after the fitness value did not improve during 1000 last evaluations. Finally, both approaches stopped if they obtained 100% coverage.

Each test suite found by each approach was additionally minimized after the runs. For Approach 1, this operation employed the built-in feature of *EvoSuite*, and for Approach 2, a greedy procedure was used: input tuples were removed from the test suite until no tuple could be removed without the decrease of the coverage value.

8.3 Results

The results of the experiments are outlined in Tables 2–5, where basic statistics is shown for all groups of experiments. Coverage value statistics is shown for both approaches in Tables 2 and 3:

TABLE 2 Coverage value statistics for Approach 1

FB type, coverage criterion	**Min**	**First quartile**	**Median**	**Third quartile**	**Max**
Basic, branch	54.1%	86.4%	91.7%	94.4%	98.7%
Composite, branch	32.0%	77.0%	83.0%	89.7%	94.3%
Basic, transition	55.6%	100.0%	100.0%	100.0%	100.0%
Composite, transition	5.7%	92.0%	100.0%	100.0%	100.0%

TABLE 3 Coverage value statistics for Approach 2

FB type, coverage criterion	**Min**	**First quartile**	**Median**	**Third quartile**	**Max**
Basic, branch	42.9%	71.4%	82.7%	93.0%	98.8%
Composite, branch	7.7%	56.6%	66.7%	86.6%	91.2%
Basic, transition	55.6%	100.0%	100.0%	100.0%	100.0%
Composite, transition	5.7%	70.0%	92.2%	100.0%	100.0%
Basic, 2-transition	41.2%	100.0%	100.0%	100.0%	100.0%
Composite, 2-transition	0.4%	64.6%	86.4%	99.7%	100.0%

Meaningful execution time statistics is available only for Approach 2, since for Approach 1 had a priori determined execution times (10 minutes for basic FBs, 20 minutes for composite FBs). It is provided in Table 4:

TABLE 4 Execution time statistics for Approach 2, time is shown in seconds

FB type, coverage criterion	Min	First quartile	Median	Third quartile	Max
Basic, branch	2.4	3.3	4.8	7.7	17.4
Composite, branch	3.2	18.2	38.0	65.0	298.9
Basic, transition	0.0	0.0	0.1	0.3	2.5
Composite, transition	0.0	1.2	5.4	15.2	160.6
Basic, 2-transition	0.0	0.1	0.2	1.1	29.4
Composite, 2-transition	0.0	3.9	37.8	185.4	1646.9

We additionally provide statistics of test suites for both FB types and both approaches, which is available in Table 5:

TABLE 5 Test suite size statistics for both approaches, size is shown in the number of methods called (i.e. input tuples from the evolutionary algorithm's point of view)

FB type, approach	Min	First quartile	Median	Third quartile	Max
Basic, Approach 1	1	4	12	27	77
Basic, Approach 2	1	4	11	17.5	52
Composite, Approach 1	1	8	32	64	95
Composite, Approach 2	1	5	39	47.75	89

8.3.1 Results overview

We start analyzing the results with the overview of the data presented in the tables. First of all, coverage values are 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 entire software systems.

Next, transition coverage was generally easier to achieve than the branch one. Perfect (100%) result was achieved with Approach 1 for more than 75% of the 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. As for Approach 2, its results are worse, but still encouraging. Approach 2 is also capable of optimizing 2-transition coverage, unlike Approach 1, and the values of 2-transition coverage are quite close to the ones of transition coverage for Approach 2.

Following that, branch coverage values are generally lower than the transition coverage ones, and again, they are better for Approach 1. However, as mentioned in Section 6.4, the presence of always covered and unreachable goals does not allow to properly compare branch coverage values of both

methods. Thus, we manually examined the tests generated by the approaches and confirmed the superiority of the Approach 1 on branch coverage. Detailed comments on this examination will be given further.

As we see, Approach 1 outperforms Approach 2 in terms of obtained coverage values. We additionally tried to improve the results of Approach 2 by tuning its parameters p_{rem}, p_{adj}, and N_{op} with the *irace* tool (López-Ibáñez, Dubois-Lacoste, Stützle & Birattari, 2011) and by running the approach twice and combining two test suites, but such adjustments weakly influenced the performance of the Approach 2. The attempt to apply branch coverage as a secondary optimization criterion, that is, if the transition coverage value of two test suites is equal, to prefer the one with higher branch coverage, did not improve its performance either. However, as Table 4 suggests, Approach 2 requires significantly less time (less than one minute in more than 50% of cases) compared to Approach 1, which takes 10–20 minutes to obtain high coverage values (we remind that the execution times were fixed for Approach 1), not to mention its ability to support more coverage goals.

We finally compare the sizes of test suites obtained by the two approaches, which are shown in Table 5. Median sizes of test suites obtained by different approaches are very close to each other, but the third quartile of the size is larger for Approach 1. This is not surprising, since this approach produces test suites with better coverage values on some FBs, and to achieve better coverage one might need larger test suites.

8.3.2 Examination of generated test suites

After the results had been obtained, the generated tests were run in the *Eclipse IDE* (https://eclipse.org/) with the *EclEmma* plugin (http://www.eclemma.org/), which integrates *Eclipse* with *JUnit*. Uncovered FB parts were manually examined. Based on a brief examination of the generated test suites, several conclusions can be drawn:

- If inaccessible coverage goals mentioned in Section 6.4 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 test suites generated by Approach 1. As for Approach 2, these numbers are 16 and 4 respectively.
- In *EvoSuite* and *JaCoCo*, branch coverage assumes the coverage of each combination of conditions in an ‘if’ decision. If this condition is weakened to just cover both ‘then’ and ‘else’ branches in each decision (i.e. to have tests which trigger each of the two possible outcomes of the whole decision, regardless of the number of conditions in it), 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: such methods can be detected with static code analysis, for example performed by the *Eclipse IDE*.

Next, we examined tests suites produced by Approach 1 in more detail. 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 for two FBs were covered by augmenting the suite generated for the branch criterion with a test from the transition-based test suite. For another FB, it was quite easy to modify one of the automatically generated test suites to improve its branch coverage.

Finally, we identified one basic FB from the HPP system with several states inaccessible due to a forgotten update of an internal variable and two basic FBs with algorithm branches inaccessible due to faults in 'if' decisions. This last situation is illustrated in Fig. 13, where an evidently unsatisfiable decision (`AI.value < PRESET_H.value & AI.value >= PRESET_H.value`) prevents the execution of the branch colored in red. Other colors denote partial coverage (yellow) and proper coverage (green). The coloring was automatically performed by *EclEmma*.

```
5809        public void alg_REQ() {
5810            ;
5811            if (((DIH.value & DIL.value)
5812                    | (AI.value >= PRESET_HH.value & !DIH.value)
5813                    | (AI.value <= PRESET_LL.value & !DIL.value)
5814                    | (AI.value > PRESET_MAXAI.value) | (AI.value <
5815                SensorFault.value = true;
5816                AlarmHH.value = false;
5817                AlarmH.value = false;
5818                AlarmL.value = false;
5819                AlarmLL.value = false;
5820            } else {
5821                SensorFault.value = false;
5822                if (AI.value >= PRESET_HH.value | DIH.value) {
5823                    AlarmHH.value = true;
5824                    AlarmH.value = false;
5825                    AlarmL.value = false;
5826                    AlarmLL.value = false;
5827                } else if (AI.value < PRESET_H.value
5828                        & AI.value >= PRESET_H.value) {
5829                    AlarmHH.value = false;
5830                    AlarmH.value = true;
5831                    AlarmL.value = false;
5832                    AlarmLL.value = false;
5833                } else if (AI.value <= PRESET_L.value
```

FIGURE 13 An example of unreachable code due to an erroneous decision

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 32.1% and 5.7% for branch and transition coverage respectively, had missing event connections from its input interface to nested FBs, which signifies an error during its development in *FBDK*. 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 some variables. It also included faulty basic FBs with inaccessible parts.

All software faults detected by test suites generated by Approach 1 were also possible to detect with Approach 2, but using test suites produced by Approach 2 typically requires more manual work, since it yields more "false alarms", i.e. uncovered code segments which are in fact reachable and are usually reached by Approach 1. This, in turn, is a result of the fact that *EvoSuite* outperforms the implemented evolutionary algorithm in terms of coverage, possibly because it follows a combined approach to optimization, not only the evolutionary one (see the end of Section 4.2).

9 DISCUSSION AND CONCLUSIONS

In the beginning of the thesis we have reviewed several fields which are important for the design of the coverage test generation method for software systems represented in the IEC 61499 standard. First, the knowledge about the standard has been obtained, which has helped to understand the nature of the systems under test better. Second, various coverage criteria and approaches for test automation have been investigated. Evolutionary computation, which is one of such methods, has also been examined in a separate section. The obtained knowledge has allowed us to answer several research questions of the thesis, to formally state the problem of the constructive research, and to apply the reviewed concepts in it.

The rest of the thesis, starting from Section 6, has answered the remaining research questions, which concern the development and the evaluation of new artifacts. Two methods which generate input test data for IEC 61499 function blocks and try to maximize test suite coverage have been proposed in the thesis. The first one is based on third-party tools, while the second one is implemented independently of them. The obtained results and their manual examination suggest that the proposed methods are applicable in practice. In particular, they have helped to identify several faults is the systems under test, which made some of their parts unreachable. The methods, however, differ in the quality of obtained test suites (the first one is superior), in execution time, and in the set of supported coverage goals (the second one is superior). Thus, the choice of the method to apply depends on whether quality or time is more important.

The inferiority of the method which employed the implementation of evolution independent from the one of *EvoSuite* in terms of coverage of generated test suites might be due to complex techniques used in *EvoSuite* and mentioned in Section 4.2, while the manual approach is based on evolutionary computation only. It is also possible to combine both approaches: use the translation of FBs into *Java* which is not based on *FBDK*, and apply *EvoSuite* on the obtained *Java* code. Such combined approach would be similar to the one totally based on third-party tools, but would support FB execution semantics

different to the one assumed in *FBDK*. The implementation and evaluation of the combined approach, however, are out of scope of this thesis.

The performed study has several limitations, which might be overcome in future research. The first limitation is connected with the nature of evolutionary algorithms, which do not always generate perfect solutions. To resolve it, it is possible to replace the evolutionary search with one of symbolic constraint-based approaches (Cadar & Sen, 2013). Next, we have not cared about output data assertions which can be added to generated test suites. For example, correct outputs for a defined sequence of inputs might be available, if there exists a detailed model of the application. Besides, some outputs might be forbidden by assertions if they signify software fallacies. Furthermore, the used systems under test do not truly represent the complexity of industrial automation software, as they were designed to control relatively simple devices. Nevertheless, they contain various interactions between FBs. Finally, it is sometimes possible that the behavior of the code obtained from an FB is not equivalent to the one of this FB, when it is run in the tool where it has been developed. For the first presented test generation method, this can happen when the FB was created not in *FBDK*. This issue has been partly resolved in the second method, which can use different FB execution order, though.

Our final claim concerns the connection of the proposed approaches with MBT. MBT suggests test generation based on the formal model of the requirements for the software, which are prepared independently of the implementation. In contrast, the proposed methods use the implementation, not the requirements, as input data. Hence, they do not fit in the domain of MBT, although this field was very useful in the thesis due to the definitions of coverage criteria.

REFERENCES

Binder, R. (2000). *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional.

Boussaïd, I., Lepagnot, J. & Siarry, P. (2013). A survey on optimization metaheuristics. *Information Sciences*, vol. 237, pp. 82–117. Elsevier.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M. & Pretschner, A. (eds.). (2005). *Model-Based Testing of Reactive Systems; Advanced Lectures*. *Lecture Notes in Computer Science*, vol. 3472. Berlin-Heidelberg: Springer Verlag.

Buzhinsky, I., Ulyantsev, V., Veijalainen, J. & Vyatkin, V. (2015). Evolutionary Approach to Coverage Testing of IEC 61499 Function Block Applications. *13th IEEE International Conference on Industrial Informatics (INDIN'15)*, Cambridge, UK. IEEE. In press.

Cadar C. & Sen K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM*, vol. 56, no. 2, pp. 82–90. ACM.

Clarke, E.M., Grumberg, O. & Peled, D. (1999). *Model checking*. Cambridge: MIT press.

Colla, M., Brusaferri, A. & Carpanzano, E. (2006). Applying the IEC-61499 model to the shoe manufacturing sector. *11th IEEE Conference on Emerging Technologies and Factory Automation*, *ETFA'06*, pp. 1301–1308. IEEE.

Conformiq – Automated Test Design. (2014). Retrieved 14.12.2014 from http://www.conformiq.com/

Cormen, T.H., Leiserson, C.E., Rivest, R.L & Stein, C. (2001). *Introduction to algorithms*. Vol. 2. Cambridge: MIT press.

Creswell, J. (2007) *Review of the Literature*, Chapter 2 of *Research Design: Qualitative, Quantitative, and Mixed Method Approaches*. Thousand Oaks: Sage Publications.

Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms*. Vol. 16. John Wiley & Sons.

Della Croce, F., Tadei, R. & Volta, G. (1995). A genetic algorithm for the job shop problem. *Computers & Operations Research*, vol. 22, no. 1, pp. 15–24. Elsevier.

DeMilli, R.A. & Offutt, A.J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910.

Edvardsson, J. (1999). A survey on automatic test data generation. *2nd Conference on Computer Science and Engineering*, pp. 21–28. Linkoping. ECSEL.

Enoiu, E.P., Sundmark, D. & Pettersson, P. (2013). Model-based test suite generation for function block diagrams using the UPPAAL model checker. *6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 158–167. IEEE.

Fraser, G. & Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. New York, NY, USA, pp. 416–419. ACM.

Gutjahr, W. (1999). Partition testing versus random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 661–674. IEEE.

Hall, K.H., Staron, R.J. & Zoitl, A. (2007). Challenges to industry adoption of the IEC 61499 standard on event-based function blocks. *5th IEEE International Conference on Industrial Informatics (INDIN'07)*, Vienna, Austria, vol. 2, pp. 823–828. IEEE.

Harman, M. (2011). Software Engineering Meets Evolutionary Computation. *Computer*, vol. 44, no. 11, pp. 31–39. IEEE.

Helke, S., Neustupny, T. & Santen, T. (1997). Automating test case generation from Z specifications with Isabelle. Jonathan P. Bowen, Michael G. Hinchey, and David Till (eds.), *Proceedings of the 10th International Conference of Z Users: The Z Formal Specification Notation (ZUM 1997). Lecture Notes in Computer Science*, vol. 1212, pp. 52–71. Springer Berlin Heidelberg.

Hevner, A.R., March, S.T., Park, J. & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, vol. 28, no. 1, pp. 75–105. Minneapolis, MN, USA. MISRC.

Hussain, T. & Frey, G. (2006). UML-based development process for IEC 61499 with automatic test-case generation. *IEEE Conference on Emerging Technologies and Factory Automation (ETFA'06)*, Prague, Czech Republic, pp. 1277–1284. IEEE.

IEEE. (2013). *P29119-1-FDIS, Apr 2013 – IEEE Approved Draft Standard for Software and Systems Engineering – Software Testing – Part 1: Concepts and Definitions.* IEEE.

International Electrotechnical Commission. (2003). *International Standard IEC 61131-3: Programmable controllers – Part 3: Programming languages*. Second ed. Geneva: International Electrotechnical Commission.

International Electrotechnical Commission. (2008). *International Standard IEC 62424, Specification for Representation of process control engineering requests in P&IDs*. Geneva: International Electrotechnical Commission.

International Electrotechnical Commission. (2012). *International Standard IEC 61499-1: Function blocks – Part 1: Architecture*. Second ed. Geneva: International Electrotechnical Commission.

Koza, J.R. (1992). *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. Cambridge: MIT press.

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

López-Ibáñez, M., Dubois-Lacoste, J., Stützle T. & Birattari, M. (2011). *The irace package, Iterated Race for Automatic Algorithm Configuration.* Technical Report TR/IRIDIA/2011-004, IRIDIA, Université libre de Bruxelles, Belgium.

Mitchell, M., Holland, J.H. & Forrest, S. (1994). When will a genetic algorithm outperform hill climbing? *Advances in Neural Information Processing Systems*, vol. 6, pp. 51–58. Morgan Kaufmann, San Mateo, CA.

nxtControl – nxtSTUDIO. (2014). Retrieved 14.12.2014 from http://www.nxtcontrol.com/en/engineering/

Patil, S., Vyatkin, V. & Sorouri, M. (2012). Formal verification of intelligent mechatronic systems with decentralized control logic. *17th IEEE Conference on Emerging Technologies & Factory Automation (ETFA'12)*, Krakow, Poland, pp. 1–7. IEEE.

Peltola, J., Sierla, S., Aarnio, P. & Koskinen, K. (2013). Industrial evaluation of functional Model-Based Testing for process control applications using CAEX. *18th IEEE Conference on Emerging Technologies & Factory Automation (ETFA'13)*, Cagliary, Italy, pp. 1–8. IEEE.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W.E. (1991). *Object-oriented modeling and design*, vol. 199, no. 1. Englewood Cliffs, NJ: Prentice-hall.

Schaffer, J.D. & Eshelman, L.J. (1991). On crossover as an evolutionary viable strategy. *4th International Conference on Genetic Algorithms*, pp. 61–68. San Mateo, CA, Morgan Kaufmann.

Srinivas, N. & Deb K. (1994). Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, vol. 2, no. 3, pp. 221–248.

Thramboulidis, K. (2006). IEC 61499 in factory automation. *Advances in Computer, Information, and Systems Sciences, and Engineering*, pp. 115–124. Springer Netherlands.

von Styp, S. & Yu, L. (2013). Symbolic Model-Based Testing for Industrial Automation Software. *Hardware and Software: Verification and Testing*, pp. 78–94). Springer International Publishing.

Vyatkin, V. & Chouinard, J. (2008). On Comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations. *6th IEEE International Conference on Industrial Informatics (INDIN'08)*, Daejeon, Korea, pp. 289–294.

Zhou, A., Qu, B. Y., Li, H., Zhao, S. Z., Suganthan, P. N. & Zhang, Q. (2011). Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 32–49. Elsevier.

Zoitl, A. & Vyatkin, V. (2009). IEC 61499 Architecture for Distributed Automation: the 'Glass Half Full' View. *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 7–23. IEEE.

APPENDIX 1 EXAMPLE OF JAVA CODE PREPARED FOR EVOSUITE EXECUTION

In this appendix, the code generated from the composite FB `my_sensor2` from the PnP system is presented. The interface of the FB and the network of FBs inside are shown in Fig. 14 (the screenshots were made in *FBDK*).

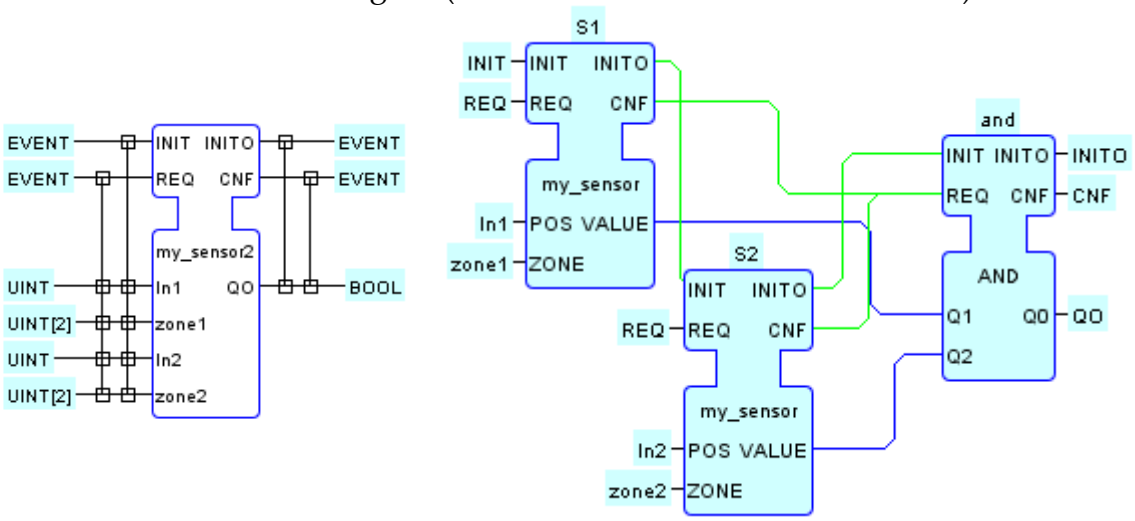

FIGURE 14 The interface (left) and the FB network (right) of the composite FB `my_sensor2`.

```
package fb.rt.pnp;

import fb.rt.*;
import fb.rt.net.*;
import fb.datatype.*;

public class my_sensor2__Composite {
    private final my_sensor2 instance = new my_sensor2();

    public void event_INIT(int In1_, int zone1_0, int
zone1_1, int In2_, int zone2_0, int zone2_1) {
```

```
        instance.In1.value = Math.abs(In1_);
        ((UINT) instance.zone1.value[0]).value =
Math.abs(zone1_0);
        ((UINT) instance.zone1.value[1]).value =
Math.abs(zone1_1);
        instance.In2.value = Math.abs(In2_);
        ((UINT) instance.zone2.value[0]).value =
Math.abs(zone2_0);
        ((UINT) instance.zone2.value[1]).value =
Math.abs(zone2_1);
        instance.INIT.serviceEvent(instance);
    }

    public void event_REQ(int In1_, int zone1_0, int
zone1_1, int In2_, int zone2_0, int zone2_1) {
        instance.In1.value = Math.abs(In1_);
        ((UINT) instance.zone1.value[0]).value =
Math.abs(zone1_0);
        ((UINT) instance.zone1.value[1]).value =
Math.abs(zone1_1);
        instance.In2.value = Math.abs(In2_);
        ((UINT) instance.zone2.value[0]).value =
Math.abs(zone2_0);
        ((UINT) instance.zone2.value[1]).value =
Math.abs(zone2_1);
        instance.REQ.serviceEvent(instance);
    }

    private class my_sensor2 extends FBInstance {
        public UINT In1 = new UINT();
        public ARRAY zone1 = new ARRAY(new UINT(),2);
        public UINT In2 = new UINT();
        public ARRAY zone2 = new ARRAY(new UINT(),2);
        public BOOL QO = new BOOL();
        public EventOutput INIT = new EventOutput();
        public EventOutput REQ = new EventOutput();
        public EventOutput INITO = new EventOutput();
        public EventOutput CNF = new EventOutput();
        protected my_sensor S1 = new my_sensor() ;
        protected my_sensor S2 = new my_sensor() ;
        protected AND and = new AND() ;

        public my_sensor2() {
            super();
            INIT.connectTo(S1.INIT);
            S1.INITO.connectTo(S2.INIT);
            REQ.connectTo(S1.REQ);
            S2.INITO.connectTo(and.INIT);
            S1.CNF.connectTo(and.REQ);
            S2.CNF.connectTo(and.REQ);
```

```
            and.INITO.connectTo(INITO);
            and.CNF.connectTo(CNF);
            REQ.connectTo(S2.REQ);
            S1.connectIVNoException("POS",In1);
            S1.connectIVNoException("ZONE",zone1);
            S2.connectIVNoException("POS",In2);
            S2.connectIVNoException("ZONE",zone2);

and.connectIVNoException("Q1",S1.ovNamedNoException("VALUE"
));

and.connectIVNoException("Q2",S2.ovNamedNoException("VALUE"
));
            QO = (BOOL) and.ovNamedNoException("Q0");
        }
    }

    private class my_sensor extends FBInstance {
        public UINT POS = new UINT();
        public ARRAY ZONE = new ARRAY(new UINT(),2);
        public BOOL VALUE = new BOOL();
        public EventServer INIT = new EventInput(this);
        public EventServer REQ = new EventInput(this);
        public EventOutput INITO = new EventOutput();
        public EventOutput CNF = new EventOutput();

        public ANY ovNamed(String s) throws
FBRManagementException {
            if ("VALUE".equals(s)) return VALUE;
            return super.ovNamed(s);
        }

        public void connectIV(String ivName, ANY newIV)
            throws FBRManagementException {
            if ("POS".equals(ivName)) connect_POS((UINT)
newIV);
            else if ("ZONE".equals(ivName))
connect_ZONE((ARRAY) newIV);
            else super.connectIV(ivName, newIV);
        }

        public void connect_POS(UINT newIV) {
            POS = newIV;
        }

        public void connect_ZONE(ARRAY newIV) {
            ZONE = newIV;
        }

        private static final int index_START = 0;
```

```
private void state_START() {
    eccState = index_START;
}

private static final int index_INIT = 1;

private void state_INIT() {
    eccState = index_INIT;
    alg_INIT();
    INITO.serviceEvent(this);
    state_START();
    transition_my_sensor_0();
}

private static final int index_REQ = 2;

private void state_REQ() {
    eccState = index_REQ;
    alg_REQ();
    CNF.serviceEvent(this);
    state_START();
    transition_my_sensor_1();
}

public my_sensor() {
    super();
}

public void serviceEvent(EventServer e) {
    if (e == INIT) service_INIT();
    else if (e == REQ) service_REQ();
}

public void service_INIT() {
    if ((eccState == index_START)) {
        state_INIT();
        transition_my_sensor_2();
    }
}

public void service_REQ() {
    if ((eccState == index_START)) {
        state_REQ();
        transition_my_sensor_3();
    }
}

public void alg_INIT() {
    VALUE.value = false;
```

```
        }

        public void alg_REQ() {
            if ((POS.value >= ((UINT) ZONE.value[0]).value)
& (POS.value <= ((UINT) ZONE.value[1]).value)) {
                VALUE.value = true;
            } else {
                VALUE.value = false;
            }

        }
    }

    private void transition_my_sensor_0() {
    }

    private void transition_my_sensor_1() {
    }

    private void transition_my_sensor_2() {
    }

    private void transition_my_sensor_3() {
    }

    private class AND extends FBInstance {
        public BOOL Q1 = new BOOL();
        public BOOL Q2 = new BOOL();
        public BOOL Q0 = new BOOL();
        public EventServer INIT = new EventInput(this);
        public EventServer REQ = new EventInput(this);
        public EventOutput INITO = new EventOutput();
        public EventOutput CNF = new EventOutput();

        public ANY ovNamed(String s) throws
FBRManagementException {
            if ("Q0".equals(s)) return Q0;
            return super.ovNamed(s);
        }

        public void connectIV(String ivName, ANY newIV)
            throws FBRManagementException {
            if ("Q1".equals(ivName)) connect_Q1((BOOL)
newIV);
            else if ("Q2".equals(ivName)) connect_Q2((BOOL)
newIV);
            else super.connectIV(ivName, newIV);
        }

        public void connect_Q1(BOOL newIV) {
```

```
    Q1 = newIV;
}

public void connect_Q2(BOOL newIV) {
    Q2 = newIV;
}

private static final int index_START = 0;

private void state_START() {
    eccState = index_START;
}

private static final int index_INIT = 1;

private void state_INIT() {
    eccState = index_INIT;
    alg_INIT();
    INITO.serviceEvent(this);
    state_START();
    transition_AND_0();
}

private static final int index_REQ = 2;

private void state_REQ() {
    eccState = index_REQ;
    alg_REQ();
    CNF.serviceEvent(this);
    state_START();
    transition_AND_1();
}

public AND() {
    super();
}

public void serviceEvent(EventServer e) {
    if (e == INIT) service_INIT();
    else if (e == REQ) service_REQ();
}

public void service_INIT() {
    if ((eccState == index_START)) {
        state_INIT();
        transition_AND_2();
    }
}

public void service_REQ() {
```

```
            if ((eccState == index_START)) {
                state_REQ();
                transition_AND_3();
            }
        }

        public void alg_INIT() {
            Q0.value = Q1.value & Q2.value;
        }

        public void alg_REQ() {
            Q0.value = Q1.value & Q2.value;
        }
    }

    private void transition_AND_0() {
    }

    private void transition_AND_1() {
    }

    private void transition_AND_2() {
    }

    private void transition_AND_3() {
    }
}
```

APPENDIX 2 EXAMPLE OF A TEST SUITE PRODUCED BY EVOSUITE

```
package fb.rt.pnp;

import static org.junit.Assert.*;
import org.junit.Test;
import fb.rt.pnp.my_sensor2__Composite;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.evosuite.runtime.EvoSuiteFile;
import org.junit.runner.RunWith;

public class my_sensor2__Composite_ESTest {
  //Test case number: 0
  @Test
  public void test0()  throws Throwable  {
      my_sensor2__Composite my_sensor2__Composite0 = new
my_sensor2__Composite();
      my_sensor2__Composite0.event_REQ((-2823), (-2823), (-
2823), 1462, (-2823), (-1026));
  }

  //Test case number: 1
  @Test
  public void test1()  throws Throwable  {
      my_sensor2__Composite my_sensor2__Composite0 = new
my_sensor2__Composite();
      my_sensor2__Composite0.event_INIT(1462, (-2823), (-
2823), 1462, (-1026), (-2823));
  }
}
```