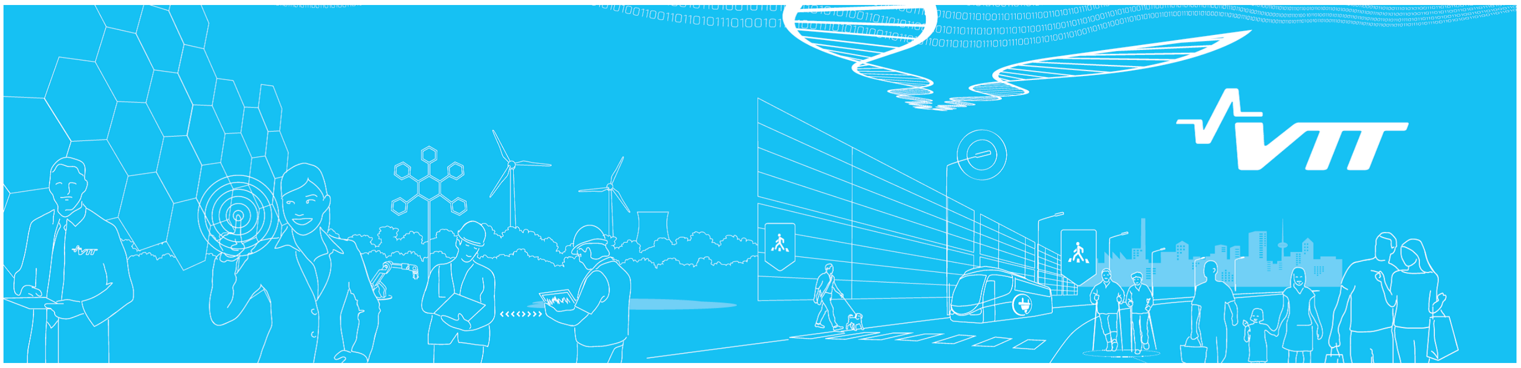
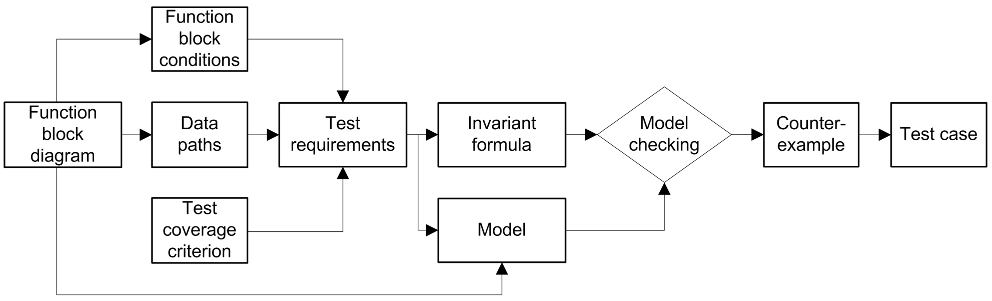


**RESEARCH REPORT**

VTT-R-04004-15



# Supporting structure-based test design using model checking

Authors: Jussi Lahtinen

Confidentiality: Public

| | |
|---|---|
| **Report's title**<br>Supporting structure-based test design using model checking | |
| **Customer, contact person, address**<br>Valtion ydinjätehuoltorahasto VYR | **Order reference**<br>SAFIR 5/2015 |
| **Project name**<br>Integrated safety assessment and justification of nuclear power plant automation | **Project number/Short name**<br>102392/SAUNA |
| **Author(s)**<br>Jussi Lahtinen | **Pages**<br>19 |
| **Keywords**<br>structure-based testing, function block diagram, model checking, mutation analysis | **Report identification code**<br>VTT-R-04004-15 |

**Summary**

Nuclear domain safety systems are commonly designed using function block diagrams that are automatically translated into software code. These safety systems need to be rigorously verified. One of the verification techniques required by standards and nuclear regulators is structure-based testing. Structure-based testing of automatically generated code is not effective in detecting defects in function block diagrams. Hence, several approaches for structure-based testing on the level of the function block diagrams have recently emerged. We have defined three structure-based test criteria for function block diagrams, and developed an automatic technique for designing test cases according to these criteria that uses model checking to generate the test cases. Unlike other similar test criteria, the developed criteria especially focus on the time-dependent aspects of the test requirements. We have tested our technique on fictitious function block diagrams, and a set of vendor-specific real-world industrial function block diagrams. The fault detection capability of the method is analysed using mutation analysis. The results suggest that the developed technique is scalable to most nuclear domain safety systems. The average fault detection capability of the generated tests ranged from 90 % to 95 % in our experiments.

| | | |
|---|---|---|
| **Confidentiality** | Public | |
| Espoo 19.1.2016 | | |
| **Written by** | **Reviewed by** | **Accepted by** |
| Jussi Lahtinen,<br>Research Scientist | Antti Pakonen,<br>Research Scientist | Riikka Virkkunen,<br>Head of Research Area |

**VTT's contact address**
VTT Technical Research Centre of Finland Ltd., P.O. Box 1000, FI-02044 VTT, FINLAND

**Distribution (customer and VTT)**
SAFIR2018 RG1 members, electronic copy
VTT archive, 1 copy

## Contents

# 1. Introduction

Digital instrumentation & control (I&C) systems are becoming more and more common in safety-critical environments such as in nuclear power plants. Nuclear domain regulators (see e.g., [1]) require that these systems are adequately verified using simulation, formal methods, and many variations of testing, including structure-based testing.

The application software of nuclear domain safety systems is commonly designed using function block diagrams (FBD). FBD as defined in the IEC standard 61131-3 [2] is a commonly used graphical programming language for programmable logic controllers, in which the design consists of inputs, outputs, and a set of simple elementary function blocks such as AND, OR, or timer function blocks, and the connections between these components. In the nuclear domain, the IEC 61131-3 standard is not commonly applied. Instead vendor-specific variants of the FBD language are typically used. In this report we use the term function block diagram, and the abbreviation FBD, to refer to the design diagrams, and not the design language itself.

In the design process, FBDs are typically converted into software code using an automatic code generator. Applying structure-based testing to automatically generated code is undesirable as the test cases can become non-intuitive and difficult to understand. Structure-based testing of automatically generated code is also not effective in detecting function block level defects in FBDs [3]. One alternative to this is to determine the structure-based tests on the level of the FBD.

Several structure-based test criteria for FBDs have recently emerged. These coverage criteria are based on interpreting the system as a data flow diagram, and generating a set of test requirements that the tests have to fulfil. However, the existing approaches are not based on formal definitions, and do not consider possible time delays associated with data paths through the FBD. In this paper, we define three new structure-based test criteria customised for FBD systems operating in constant length time intervals. The criteria are based on the work of Jee et al. [4, 5]. Our criteria extend the methodology by Jee et al. with more exact definitions of when an input affects an output. We also take the time dimension into account in our definitions. The developed methodology focuses only on Boolean valued signals, and the parts of the design concerning analogue signals are left out of test coverage calculations.

The developed coverage criteria offer a good basis for planning structure-based tests. However, on large system designs that contain memories, timers and feedback loops, it can be overwhelmingly difficult to design the test cases manually. Therefore, an automatic technique for test design is needed. We have applied a formal method called model checking [6] to assist in the test design process. In model checking, a model of the system is written and the system requirements are formalised in a suitable language, e.g. state invariants, or temporal logic. A model checking tool then analyses the model against the temporal logic clause in a way that takes all possible system behaviours into account. If it is possible to violate the specification in the model, the model checking tool gives a concrete counter-example as output that demonstrates on variable-level how the violation might occur. This ability to produce concrete counter-examples can also be exploited to generate test cases. The classic way of this is to take the negation of a system requirement and formalise that in temporal logic. When the resulting formula is analysed using the model checking tool a counter-example will be produced that is according to the original requirement. The inputs and expected outputs of the test case can then be read from the counter-example.

We have developed an automatic method for test generation that uses model checking to generate the concrete test cases. We primarily employ the k-induction algorithm [7, 8] provided by the model checker NuSMV 2.5.4 [9]. In that approach the state invariants are proved using induction, and the base step and the induction step of the proof are basically reduced to bounded model checking problems.

We have tested our technique on several fictitious FBDs, and a set of vendor-specific real-world industrial FBDs. In order to analyse the fault detection capability of the method, we have also performed mutation analysis on the case study systems. The results suggest that the developed technique is feasible for most systems. The average fault detection capability of the generated tests was 95 % when the most rigorous test criterion was used for test generation.

## 2. Related work

The idea of using model checking for test generation originally came from Callahan et. al [10] and Engels et. al [11]. The general test generation idea has since been adapted in a variety of applications, see e.g., [12] for an extensive survey.

Applying formal methods to generate structure-based tests for FBDs is a newer research subject. To the best of our knowledge, this line of work started after Jee et al. developed [4, 5] the first structure-based coverage criteria for FBDs: basic coverage (BC), input condition coverage (ICC), and complex condition coverage (CCC). In [13], the authors describe a technique for generating test cases automatically based on these criteria. The technique is based on using a Satisfiability Modulo Theories (SMT) solver. In their experiments they show that the technique substantially outperforms manually created tests, and could detect 82 % of previously known faults in an industrial case study system, whereas manually generated tests only detected 35 % of the faults.

Another coverage criterion called FB-Path Complete Condition Test Coverage (FPCC) for FBDs is introduced in [14] with the aim of detecting function mutation errors. An automatic tool is introduced in the paper that translates the PLCOpen XML form FBDs directly to models for the UPPAAL model checker. The UPPAAL tool is used to generate the test cases.

In [15], an automatic test generation technique for FBDs is also described. In this paper, two coverage metrics called MC/DC4d and Propagation Toggle Coverage (PTC) are described. MC/DC4d is based on the commonly used Modified Condition/Decision Coverage (MC/DC) [16] criterion. The developed tool uses an SMT solver to generate the tests. The authors have also experimented with mutation analysis to confirm the ability of the technique to detect defects.

The work by Enoiu et al. [17] also discusses FBDs in the context of test generation using model checking. They have defined their own coverage criteria for FBDs, and used the model checker UPPAAL to generate a test set. This work is continued in [18]. In addition to the previous criteria, the MC/DC criterion has also been applied to the systems and the technique is evaluated in several industrial programs. The results of this work show that their technique is efficient and scales well for most of the programs.

In this paper we have defined our own coverage metrics that are based on the definitions by

Jee et al. [4, 5], and applied the model checker NuSMV to generate test cases. Like other authors, we have also applied the MC/DC principles in these definitions. However, our metrics are more formally defined, and focus on the time dimension in a way that has not been done in other papers. We also present a model checker assisted way of composing the input-output conditions required in the technique. An early alternative version of our technique and its implementation are explained in more detail in a conference paper [19].

# 3. Test coverage criteria

In this Chapter, we give definitions for three coverage metrics that we call modified basic coverage (MBC), modified input condition coverage (MICC) and modified complex condition coverage (MCCC). The criteria are based on interpreting the FBD as a data flow graph, and then creating test requirements for the data paths of that graph. The test requirements state that the input of a data path must individually affect the output of the path. In section 3.1, we first define these input-output conditions for individual function blocks and for data paths. In section 3.2, we then use these definitions and introduce the test coverage metrics. Finally, in section 3.3 we demonstrate the definitions using a small example system.

## 3.1 Definitions

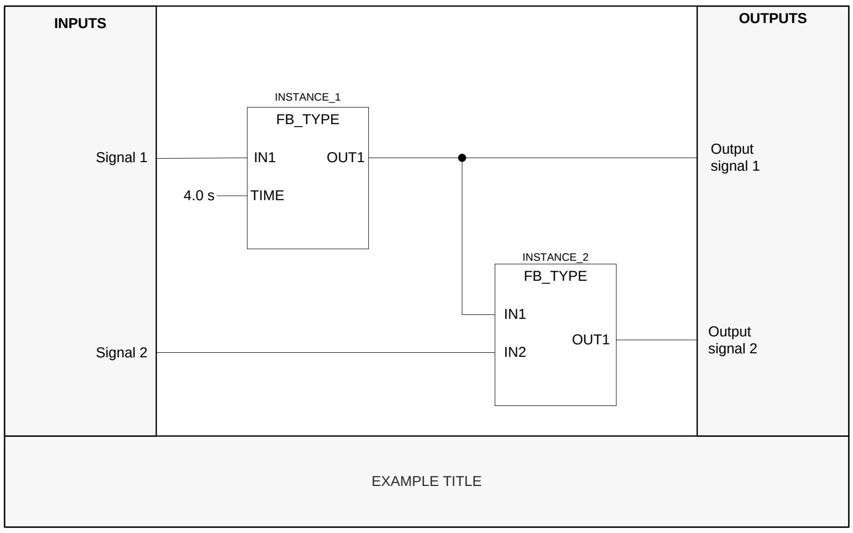


*Figure 1. An example of a function block diagram (FBD)*

A function block diagram is built by connecting function block instances to each other by drawing lines between input and output gates of the function blocks. It is also possible to create higher level hierarchical function block structures by assembling instances of function blocks to form new composed components. An example of a FBD following the graphical notations used in this paper is illustrated in Fig. 1.

A function block is defined as $FB = \langle I, O, IV, P \rangle$ where $I$ is the set of input edges, $O$ is the set of output edges, $IV$ is the set of internal variables, and $P$ is the set of parameters that the function block has. We define an edge as a connection between two function blocks, or a function block and an input/output. An edge $e$ at time point $t$ is denoted $e^t$.

We define a Function Block Condition (FBC) as the logical condition under which the output edge $e_o$ of a function block is affected by the value at the input edge $e_i$. We base this definition on the widely used coverage criterion Modified Condition/Decision Coverage (MC/DC). One requirement of the MC/DC requirement is that it must be shown that each input individually affects the outputs. We interpret this so that a Boolean input of a system is shown to affect a Boolean output if negating the input value also results in a negated output value, while the values of all other inputs are fixed. In this methodology we only focus on edges that have Boolean values, and only use paths that consist only of Boolean edges (Boolean paths). The parts of the design concerning analogue signals are left out of test coverage calculations. In case analogue valued edges exist within a path, only the Boolean valued fragment of that path is considered as a path.

An input of a function block may affect an output instantaneously, or with a delay (e.g. in the case of the DELAY function block). We define a function block condition delay (FBCD) as the smallest delay at which the input can individually affect the output. For example, the AND function block has a FBCD value of 0 since the inputs immediately affect the output value, i.e. $FBCD_{AND}(\langle e_i, e_o \rangle) = 0$. If the input can not be shown to affect the output at any delay value, an FBCD for the input-output pair does not exist.

In order to define the FBC formally, we must first define a timed variant of the FBC named Timed Function Block Condition (TFBC) as a condition under which the output edge $e_o$ at time $t_2$ is affected by an input edge $e_i$ at time $t_1$, denoted as: $TFBC_{FB}(\langle e_i^{t_1}, e_o^{t_2} \rangle)$.

Formally the TFBC can be defined as follows. An evaluation of an edge $e$ at time point $t$ is denoted as $v(e^t)$. An assignment of an edge $e$ to the value *True* at time point $t$ is denoted $v(e^t \leftarrow True)$. TFBC is a formula of the input edges $I$ of the function block excluding $e_i$ at time point $t_1$. Correspondingly, the TFBC formula is true in a set of evaluations of the input edges denoted as $C$. The value of the output edge $e_o$ at time point $t_2$ is determined by a function $f_{e_o^{t_2}}(c, v(e_i^{t_1}))$, where $c$ denotes the evaluations of the input edges of the function block at time points ranging from 1 to $t_2$ $e_1^1, e_1^2, \ldots, e_1^{t_2}, \ldots, e_k^1, e_k^2, \ldots, e_k^{t_2}$, excluding $e_i^{t_1}$. The output edge $e_o$ at time $t_2$ is affected by an input edge $e_i$ at time $t_1$ when the set of evaluations $C$ is such that:

$$c \in C \iff f_{e_o^{t_2}}(c, v(e_i^{t_1} \leftarrow True)) = \neg f_{e_o^{t_2}}(c, v(e_i^{t_1} \leftarrow False)).$$

Intuitively, this means that the evaluation of the output is of opposite value when the input is flipped. Finally, the function block condition (FBC) of an input-output pair ($e_i$, $e_o$) can then be formally defined as:

$$FBC_{FB}(\langle e_i, e_o, t \rangle) = \begin{cases} TFBC_{FB}(\langle e_i^{t-FBCD_{FB}(\langle e_i, e_o \rangle)}, e_o^t \rangle) & \text{If } FBCD_{FB}(\langle e_i, e_o \rangle) \\ & \text{exists.} \\ FALSE & \text{Otherwise} \end{cases}$$

In the equation $t$ denotes a non-specific reference point of time. The function block condition formula is not dependent on the specific value of $t$.

Next we define data paths of the FBD, and a condition called data path condition (DPC) under which the input of the data path affects the output. A data path of a FBD is defined as a finite sequence $\langle e_1^{t_1}, e_2^{t_2} \ldots e_n^{t_n} \rangle$ of edges where all the edges succeed one another on the FBD. The time parameters of a path shall be such that $t_{j+1} = t_j + FBCD_{FB}(\langle e_j, e_{j+1} \rangle)$ to accommodate for the delays induced by the individual function blocks. We define a Boolean data path as a fragment of a data path in which all the edges carry Boolean data.

A data path condition (DPC) is a condition related to a Boolean data path under which the input value affects the output. It can be composed as the conjunction of the function block conditions on that path.

## 3.2 Modified test coverage criteria

Jee et al. [4, 5] introduced three coverage metrics: basic coverage (BC), input condition coverage (ICC), and complex condition coverage (CCC) for FBDs. Here we define slightly modified versions of these metrics that are based on the methodology of section 3.1.

The modified basic coverage (MBC) criterion is met when each DPC corresponding to a Boolean path is fulfilled by some test case at some time point.

The modified input condition coverage (MICC) criterion is met when for each Boolean path there is i) a test case in which the DPC of that path is fulfilled, and the **first edge of that path is true**, and ii) a test case in which the DPC of that path is fulfilled, and the **first edge of that path is false**.

The modified complex coverage condition (MCCC) criterion requires that **for all edges within a Boolean path** there is a test case in which i) the DPC of that path is fulfilled, and the edge is true, and ii) the DPC of that path is fulfilled, and the edge is false.

## 3.3 Example system

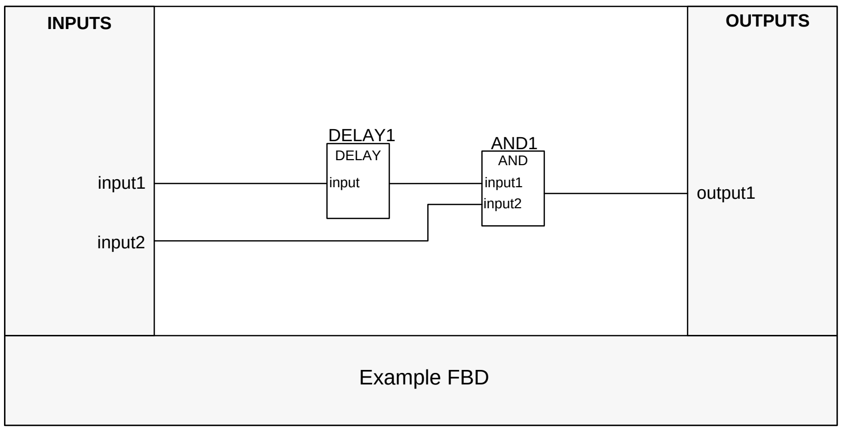


*Figure 2. An example of a function block diagram (FBD)*

An FBD we use as a running example in Fig. 2 has two function blocks: a DELAY function block and an AND gate. The FBCs for these function blocks can be found in Table 1. An input of the AND block affects the output whenever the other input is true, and there is no delay involved. The input of the DELAY function block always affects the output, but this happens with a delay of one processing cycle.

The example system has two Boolean paths: from *input*1 to *output*1, and from *input*2 to *output*1. The path from *input*1 to *output*1 has the following data path condition:

$$DPC(input1, output1, t) = FBC_{DELAY1}(\langle input1, DELAY1.output1, t-1\rangle)\wedge$$
$$FBC_{AND1}(\langle DELAY1.output1, output1, t\rangle) = v(TRUE^{t-1}) \wedge v(input2^{t}).$$

*Table 1. The function block conditions required for the analysis of the example system*

| FB | FBC | FBCD |
|---|---|---|
| AND | $FBC_{AND}(\langle input1, output1, t\rangle) = v(input2^t)$ | 0 |
| | $FBC_{AND}(\langle input2, output1, t\rangle) = v(input1^t)$ | 0 |
| DELAY | $FBC_{DELAY}(\langle input1, output1, t\rangle) = v(TRUE^{t-1})$ | 1 |

The second path from *input*2 to *output*1 has the following data path condition:

$DPC(input2, output1, t) = FBC_{AND1}(\langle input2, output1, t\rangle) = v(input1^t)$.

The MBC criterion is met when both of these Boolean formulas evaluate to true at some point in a test case, i.e. there are two test requirements. The MICC criterion has additional requirements on the first edge of the path, and results in four test requirements:

1. $v(input1^{t-1}) \wedge v(TRUE^{t-1}) \wedge v(input2^t)$
2. $\neg v(input1^{t-1}) \wedge v(TRUE^{t-1}) \wedge v(input2^t)$
3. $v(input2^t) \wedge v(input1^t)$
4. $\neg v(input2^t) \wedge v(input1^t)$

Finally, the MCCC condition has additional requirements for each edge within the paths resulting in 10 test requirements:

1. $v(input1^{t-1}) \wedge v(TRUE^{t-1}) \wedge v(input2^t)$
2. $\neg v(input1^{t-1}) \wedge v(TRUE^{t-1}) \wedge v(input2^t)$
3. $v(DELAY1.output1^t) \wedge v(TRUE^{t-1}) \wedge v(input2^t)$
4. $\neg v(DELAY1.output11^t) \wedge v(TRUE^{t-1}) \wedge v(input2^t)$
5. $v(AND1.output1^t) \wedge v(TRUE^{t-1}) \wedge v(input2^t)$
6. $\neg v(AND1.output1^t) \wedge v(TRUE^{t-1}) \wedge v(input2^t)$
7. $v(input2^t) \wedge v(input1^t)$
8. $\neg v(input2^t) \wedge v(input1^t)$
9. $v(AND1.output1^t) \wedge v(input1^t)$
10. $\neg v(AND1.output1^t) \wedge v(input1^t)$

# 4. Test set generation

## 4.1 Using model checking for generating test cases

Our technique for using model checking to generate test cases for FBDs is illustrated in Fig. 3. The prerequisites for the technique are that the function block conditions have been defined for each function block type, and that a suitable model of the FBD exists. A methodology for modelling FBDs already exists; see e.g. [20]. Once the initial information has been acquired, the Boolean data paths of the system are identified, and a set of test requirements is created.

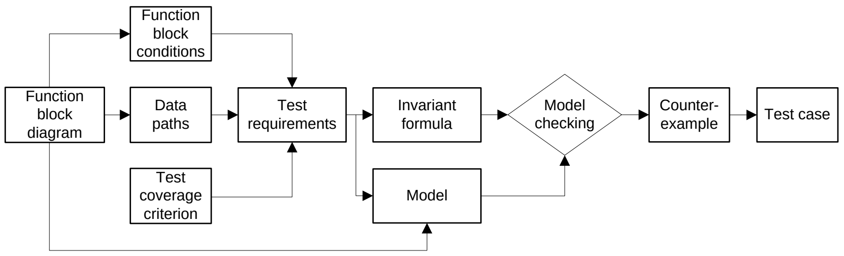


*Figure 3. Test generation using model checking*

This is straight-forward work and can be done automatically if the system design is in computer readable form.

The model is then augmented so that for each test requirement a new Boolean variable is added to the model. The variable is true whenever the test requirement is true in the model. Our methodology assumes that the system is operating cyclically in constant length time intervals. A previous time point refers to the previous operating cycle of the system.

There is also need to to divide certain test requirements into several parts. Each test requirement is related to a data path of the system. That data path may involve function blocks with delayed function block conditions (e.g. an input affects an output at a later time point). Such function blocks divide the data path into two parts: the data path before the function block, and the data path after that function block. Therefore, in these cases the test requirement also needs to refer to signal values at different time points. The test requirement is divided into parts according to these path fragments so that each part always refers to a single time point of the system. Additional variables are added tracking the values of the parts at previous time points.

In order to create a test case that fulfills a given test requirement, we create a state invariant formula stating the negation of that test requirement. The model checking tool can be used to evaluate the invariant formula against the system model. If a path exists to a state in which the test requirement is fulfilled, it is given as a counter-example. The system inputs used for a test case and the expected outputs of the system can be read from the counter-example.

## 4.2 Test set generation for the example system

We now demonstrate the test generation technique using the same running example that was defined in section 3.3, see Fig. 1. The NuSMV model code for the example system is shown in Listing 4.1. In the model code, a separate module is modelled for both function block types AND, and DELAY. Instances of the function blocks are then created in the main module. The main module also declares the input variables of the system, and defines the output of the system.

In the example case, if the MBC test criterion is followed, two test requirements need to be added to the model. The lines that need to be added to the main module are shown in Listing 4.2. There is a variable for both test requirements: *TR_1* and *TR_2*. *TR_1* consists of two parts ( *DPC_1_0* and *DCP_1_1*). Note that the variables *DPC_1_0* and *DCP_1_1* get their values based on the FBCs defined in Table 1, i.e. *DPC_1_0* is true when input1 affects the output of the DELAY, and *DPC_1_1* is true when the output of the DELAY affects the output of the AND gate.

Because of the delay on the path, *TR_1* is true whenever *DPC_1_1* is true and *DPC_1_0* was true at the previous time step. The previous value of *DPC_1_0* is captured with the variable *prev_1_DPC_1_0*.

In order to find a test case that fulfills test requirement 1 an invariant is added on line 11 in Listing 4.2.

```
MODULE main
VAR
  input1 : boolean;
  input2 : boolean;
  DELAY1 : DELAY(input1);
  AND1 : AND_2(DELAY1.output1, input2);
DEFINE
  output1 := AND1.output1;
MODULE AND_2(input1,  input2)
DEFINE
  output1 := input1 & input2;
MODULE DELAY(input1)
VAR
  prev : boolean;
DEFINE
  output1 := prev;
ASSIGN
  init(prev) := FALSE;
  next(prev) := input1;
```

*Listing 4.1. NuSMV model code for the example system*

```
 1 VAR
 2 prev_1_DPC_1_0 : boolean;
 3 DEFINE
 4   DPC_1_0 := TRUE;
 5   DPC_1_1 := (input2);
 6   TR_1 := prev_1_DPC_1_0 & DPC_1_1;
 7   TR_2 := DELAY1.output1;
 8 ASSIGN
 9   init(prev_1_DPC_1_0) := FALSE;
10   next(prev_1_DPC_1_0) := DPC_1_0 ;
11 INVARSPEC ( ! TR_1 );
```

*Listing 4.2. NuSMV model code needed for incorporating the test requirements*

When model checking is run on the generated model, a counter-example is received. The counter-example corresponds to a test case that is depicted as a timing diagram in Fig. 4. *TR_1* is satisfied at the second time step as the previous value of *DPC_1_0* is true, and *input2* is true.

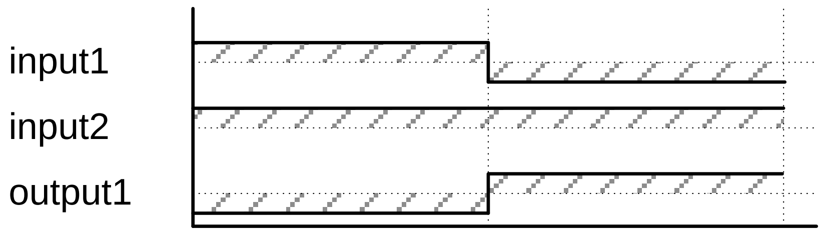


*Figure 4. Generated test case for the example system*

**Algorithm 1** Test set generation algorithm

```
1:  procedure GenerateTestSet(TRs, FBD)
2:      Unchecked ← TRs                                  ▷ Covered test requirements
3:      Infeasible ← ∅                                   ▷ Infeasible test requirements
4:      Tests ← ∅                                        ▷ Resulting test set
5:      while |Unchecked| > 0 do
6:          CurrentTR ← Unchecked.pop()
7:          [testfound, ce] = runMC(FBD, TRs, CurrentTR)
8:          if not testfound then
9:              Infeasible ← Infeasible ∪ CurrentTR
10:         else                                         ▷ Test found
11:             Tests ← Tests ∪ ce
12:             satisfiedTRs ← parseCounterExample(ce, TRs)
13:             for t ∈ satisfiedTRs do
14:                 if t ∈ Unchecked then
15:                     Unchecked.remove(t)
16:                 end if
17:             end for
18:         end if
19:     end while
20: end procedure
```

## 4.3 Test set generation algorithm

In order to generate a compact test set that fulfills all feasible test requirements we follow a simple algorithm presented in Algorithm 1. In the procedure *GenerateTestSet* we assume that a set of test requirements *TRs* has been calculated and that the system *FBD* has been modelled. The procedure makes calls to another function *runMC* creates the invariant formula based on the currently examined test requirement, incorporates the test requirements as definitions in the model, and performs model checking on the model, and returns two elements: a Boolean variable *testfound* that expresses whether a suitable test case could be found, and the counter-example file *ce*, if one exists. The function *parseCounterExample* has the counter-example *ce*, and list of test requirements as input, and parses through the counter-example. It returns a list of test requirements that are true at some point within the counter-example.

The algorithm simply selects a test requirement from the set of unchecked test requirements, and tries to find a test case for it. If a test case is found we parse through the test to see all the other test requirements that were satisfied in that test. If a test case can not be found, the current test requirement is added to the set of infeasible test requirements. The process is continued until all test requirements are either satisfied by a test case, or determined infeasible.

We implemented the algorithm in a prototype tool using Python. The tool automatically deduces the structure of the FBD based on annotations, and calculates the data paths and test requirements of the system. It then augments the model by adding model code for the test requirements and runs the test generation algorithm. The tool uses the k-induction algorithm with a small bound to model check the invariant specification. If the k-induction algorithm does not find a counter-example or a proof of the invariant within that bound we check the invariant using a BDD-based approach which is guaranteed to give a result. Also, after the test set has been generated the tool checks if any of the test cases are redundant (test requirements satisfied by it can be satisfied by other test cases).

## 5. Efficient generation of the function block conditions

It can be difficult to compose the function block conditions by hand. We devised a semi-automatic technique for creating the FBCs. In this technique we create a model checking model in which there are two instances of the same function block type, and no other function blocks. In addition to these two instances there is a variable in the model called *trigger* that can become true at any one time point, and after that is always false. One input-output pair is then selected for examination. The inputs of the two function blocks are connected to variables that decide their value non-deterministically at every time point. The function block instances always receive the same inputs, except the input under examination. For this input one of the function blocks receives the result of applying the logical XOR operator with the value of the input and the variable *trigger* as operands. This results in a model in which both function blocks always receive the same input values except at a single time point the value of one input is different. The described configuration is illustrated in Fig. 5.

This framework allows us to work out the function block conditions iteratively. We start by assuming that the function block condition delay is 0, and the function block condition is false. Then we try to verify the LTL specification:

```
G (trigger -> (FBC <-> (instance1.output1 <-> !instance2.output1))),
```

where FBC is the assumed function block condition. The formula states that whenever the trigger variable is true and the function block condition is true, the outputs of the two function block instances are of opposite value. If the result is false we can manually refine the FBC based on the counter-example and check the refined specification again until the formula evaluates as true. If we deduce that the input does not affect the output (FBC is false), we increase the assumed function condition delay by one. The checked formula is also changed accordingly by adding the temporal operator `Y` (yesterday):

```
G ( Y trigger ->
(FBC <-> (instance1.output1 <-> !instance2.output1)))
```

The process can be continued until a non-false FBC is found, or it is reasonable to assume that the input does not affect the output. Note that in standard function blocks an input typically can affect the output somehow. Complex vendor-specific function blocks, however, may be built in such a way that an input may not affect all the outputs of the function block. For example, a

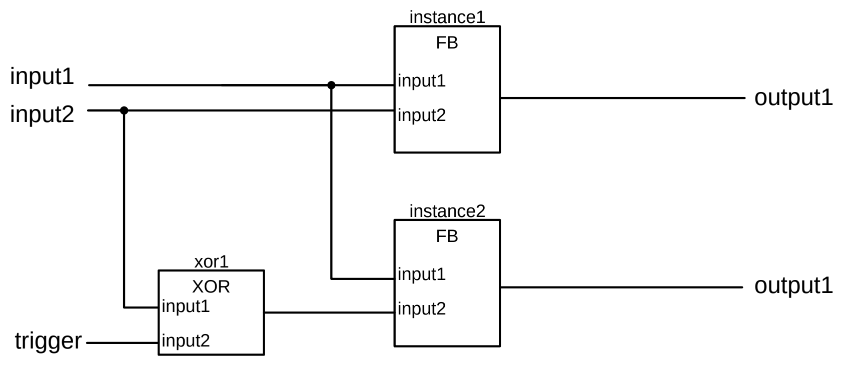


*Figure 5. Configuration for determining FBCs using model checking*

single large function block may be defined that consists of several partitions, and the inputs and outputs of a single partition are isolated from other partitions.

## 6. Experiment

We performed mutation analysis to evaluate the fault detection effectiveness of the test sets. In mutation analysis, variations or mutants of the target system are created. These mutations represent alternative erroneous designs of the system. A test set is then generated for the original system according to one of the test coverage metrics. The test set should be such that by executing the test set it is possible to distinguish between the original design and the erroneous mutant. It is checked whether running the test cases on a mutant causes behavioral changes on the outputs. If it does, the design error it represents could be observed by runnung the test set, and it is said that the mutant is killed. The quality of a test set is determined by the percentage of killed mutants.

The FBDs used in this experiment are listed in Table 2 accompanied with some data on their structure. The FBDs E1 – E9 are fictitious but realistic FBDs. The FBDs I0 – I10 are based on actual industrial designs.

We have created four different types of mutants for these FBDs: NOT-mutants, AND-OR -mutants, TIME-mutants, and Flip-Flop(FF) -mutants. NOT-mutants are created by negating a single boolean edge of the FBD. AND-OR -mutants are created by replacing a single AND function block with an OR block and vice versa. TIME-mutants are created by replacing a timer block (TON, TOF or PULSE) with another timer block with the same parameters. The FF-mutants are created by replacing a set priority flip-flop with a reset priority flip-flop and vice versa. Each mutant had only a single change when compared to the original FBD. Behaviourally equivalent mutants with respect to the original FBD were not used in the analysis.

## 7. Results

In this Chapter we present data from the test generation process for the example FBDs in Table 2, and the results of the mutation analysis experiment. The tests were generated on a PC with Intel Core i7-4600U processor and 8 GB of RAM. For model checking, NuSMV version 2.5.4 was used.

Using the developed test generation technique, we generated test sets for the example systems based on the three coverage criteria (BC, MICC and MCCC). Information related to the test generation process is presented in Table 3, Table 4, and Table 5. For each FBD, the tables show the number of test requirements, infeasible test requirements, test cases, and model checker runs. In addition, the time needed for test generation is shown.

From the test generation data we can see that in many cases the industrial FBDs used in the experiment result in a rather large number of test requirements. This is due to the fact that these FBDs contain vendor-specific function blocks with additional signal status inputs and outputs. This subsequently causes a large number of data paths and test requirements. The test requirements of the industrial FBDs are also mainly infeasible. The vendor-specific features are such that an input of a function block often can not influence an output causing all data paths flowing through that input-output pair to become infeasible. Since the FBCs of

*Table 2. Function block diagrams used as input for test generation*

| | FBD data | | | | Mutants | | | |
|---|---|---|---|---|---|---|---|---|
| ID | Inputs | Outputs | Function blocks | Data paths | NOT | AND-OR | TIME | FF |
| E1 | 3 | 1 | 6 | 3 | 7 | 2 | 4 | 0 |
| E2 | 2 | 1 | 4 | 2 | 4 | 1 | 2 | 0 |
| E3 | 1 | 1 | 1 | 1 | 2 | 0 | 2 | 0 |
| E4 | 1 | 4 | 14 | 15 | 20 | 3 | 8 | 0 |
| E5 | 4 | 4 | 12 | 18 | 21 | 4 | 2 | 1 |
| E6 | 1 | 1 | 6 | 3 | 9 | 2 | 3 | 0 |
| E7 | 1 | 2 | 5 | 6 | 9 | 2 | 2 | 0 |
| E8 | 3 | 2 | 8 | 15 | 15 | 4 | 0 | 0 |
| E9 | 2 | 5 | 10 | 10 | 17 | 2 | 2 | 0 |
| I0 | 14 | 4 | 10 | 488 | 35 | 5 | 6 | 1 |
| I1 | 14 | 14 | 14 | 1194 | 42 | 2 | 12 | 0 |
| I2 | 2 | 8 | 6 | 20 | 17 | 0 | 10 | 0 |
| I3 | 4 | 6 | 3 | 16 | 14 | 0 | 4 | 1 |
| I4 | 8 | 6 | 6 | 67 | 19 | 0 | 4 | 1 |
| I5 | 2 | 10 | 5 | 20 | 20 | 0 | 10 | 0 |
| I6 | 16 | 18 | 20 | 5692 | 74 | 6 | 14 | 1 |
| I7 | 24 | 6 | 24 | 6772 | 82 | 11 | 6 | 3 |
| I8 | 6 | 10 | 10 | 86 | 30 | 4 | 0 | 0 |
| I9 | 16 | 2 | 10 | 265 | 30 | 3 | 2 | 0 |
| I10 | 6 | 6 | 7 | 182 | 20 | 1 | 6 | 0 |

*Table 3. Test generation data for the MBC test criterion*

| FBD Name | Test reqs | Infeasible | Test cases | MC runs | Elapsed time |
|---|---|---|---|---|---|
| E1 | 3 | 0 | 1 | 2 | 0.73 s |
| E2 | 2 | 0 | 2 | 2 | 0.38 s |
| E3 | 1 | 0 | 1 | 1 | 0.36 s |
| E4 | 15 | 6 | 3 | 11 | 3.92 s |
| E5 | 18 | 0 | 2 | 7 | 1.76 s |
| E6 | 3 | 0 | 1 | 1 | 0.32 s |
| E7 | 6 | 1 | 1 | 4 | 1.35 s |
| E8 | 15 | 0 | 3 | 3 | 0.96 s |
| E9 | 10 | 0 | 1 | 3 | 1.05 s |
| I0 | 488 | 450 | 5 | 11 | 62.5 s |
| I1 | 1194 | 1037 | 21 | 26 | 180.6 s |
| I2 | 20 | 9 | 2 | 3 | 8.02 s |
| I3 | 16 | 4 | 1 | 3 | 1.05 s |
| I4 | 67 | 34 | 6 | 8 | 30.04 s |
| I5 | 20 | 5 | 1 | 2 | 1.56 s |
| I6 | 5692 | 5481 | 17 | 65 | 7004 s |
| I7 | 6772 | 6536 | 11 | 36 | 432.3 s |
| I8 | 86 | 52 | 3 | 4 | 0.67 s |
| I9 | 265 | 219 | 7 | 9 | 11.9 s |
| I10 | 182 | 137 | 5 | 6 | 14.73 s |

*Table 4. Test generation data for the MICC test criterion*

| FBD Name | Test reqs | Infeasible | Test cases | MC runs | Elapsed time |
|---|---|---|---|---|---|
| E1 | 6 | 0 | 3 | 5 | 1.40 s |
| E2 | 4 | 0 | 3 | 3 | 0.99 s |
| E3 | 2 | 0 | 2 | 2 | 0.63 s |
| E4 | 30 | 12 | 7 | 12 | 10.64 s |
| E5 | 36 | 0 | 8 | 17 | 7.57 s |
| E6 | 6 | 2 | 2 | 4 | 1.16 s |
| E7 | 12 | 4 | 2 | 4 | 2.28 s |
| E8 | 30 | 3 | 5 | 13 | 3.48 s |
| E9 | 20 | 2 | 4 | 10 | 3.14 s |
| I0 | 976 | 904 | 21 | 39 | 188.83 s |
| I1 | 2388 | 2074 | 51 | 59 | 419.6 s |
| I2 | 40 | 18 | 9 | 11 | 28.43 s |
| I3 | 32 | 8 | 7 | 9 | 3.17 s |
| I4 | 134 | 68 | 17 | 20 | 74.44 s |
| I5 | 40 | 10 | 2 | 4 | 2.81 s |
| I6 | 11348 | 11046 | 51 | 229 | 12065 s |
| I7 | 13544 | 13132 | 56 | 182 | 1428 s |
| I8 | 172 | 104 | 9 | 9 | 1.39 s |
| I9 | 530 | 459 | 16 | 37 | 49.91 s |
| I10 | 364 | 274 | 12 | 16 | 37.70 s |

*Table 5. Test generation data for the MCCC test criterion*

| FBD Name | Test reqs | Infeasible | Test cases | MC runs | Elapsed time |
|---|---|---|---|---|---|
| E1 | 20 | 0 | 3 | 5 | 1.46 s |
| E2 | 12 | 0 | 3 | 3 | 0.67 s |
| E3 | 4 | 0 | 2 | 2 | 0.76 s |
| E4 | 152 | 72 | 7 | 84 | 29.60 s |
| E5 | 178 | 0 | 7 | 17 | 10.39 s |
| E6 | 26 | 11 | 2 | 13 | 4.54 s |
| E7 | 46 | 18 | 2 | 21 | 5.52 s |
| E8 | 148 | 16 | 5 | 26 | 6.95 s |
| E9 | 104 | 15 | 4 | 23 | 5.85 s |
| I0 | 5952 | 5583 | 20 | 72 | 1613 s |
| I1 | 14488 | 12818 | 50 | 60 | 571.2 s |
| I2 | 128 | 64 | 9 | 11 | 29.7 s |
| I3 | 64 | 16 | 7 | 16 | 3.07 s |
| I4 | 510 | 282 | 18 | 21 | 79.63 s |
| I5 | 80 | 20 | 2 | 4 | 2.94 s |
| I6 | 88688 | 86869 | 51 | 1274 | 48315 s |
| I7 | 100968 | 98206 | 60 | 785 | 9257 s |
| I8 | 528 | 352 | 9 | 9 | 1.47 s |
| I9 | 2706 | 2408 | 17 | 110 | 145.2 s |
| I10 | 1725 | 1356 | 15 | 17 | 43.32 s |

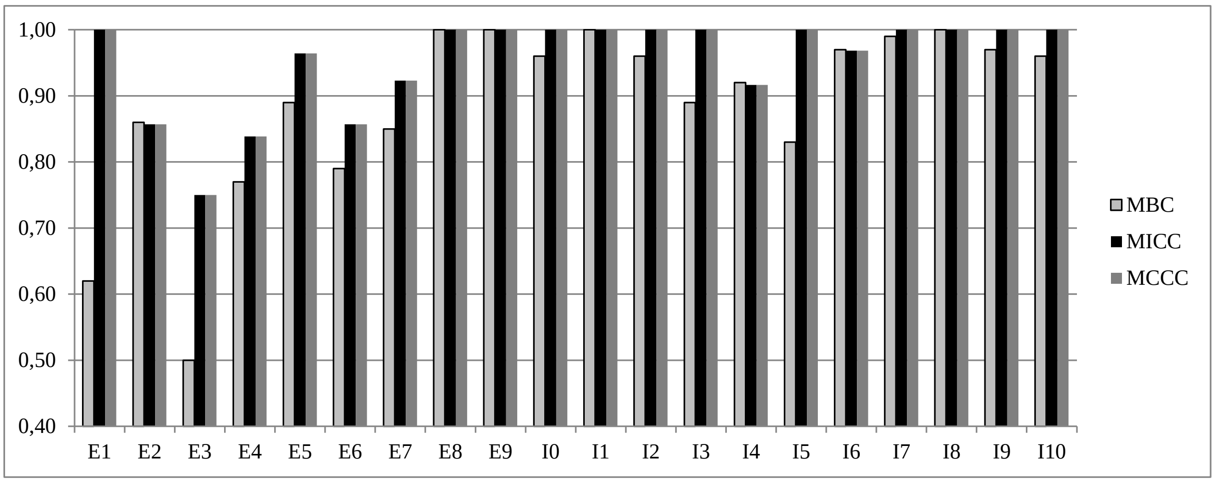


*Figure 6. Mutant kill rates for test sets based on MBC, MICC and MCCC*

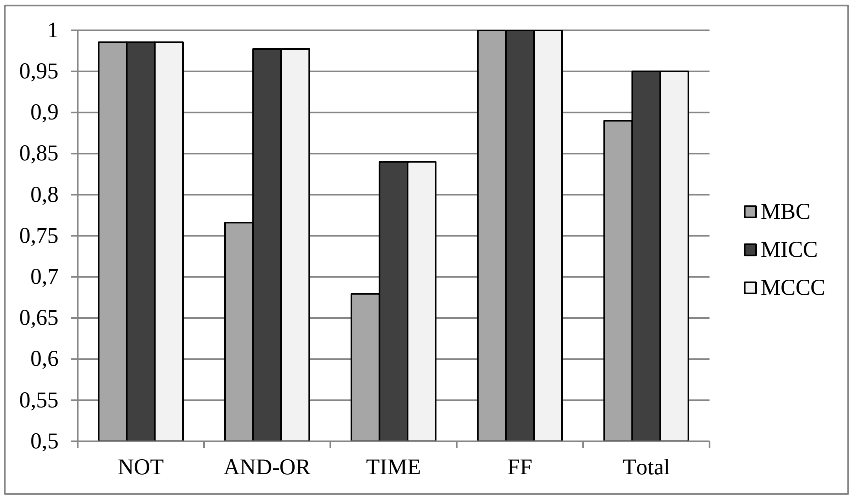


*Figure 7. Average kill rates of the different mutant variants by test sets based on MBC, MICC and MCCC*

these pairs are simply “FALSE”, these infeasible data paths can be identified in pre-processing before model checking is run, making the test generation process much quicker.

The complex FBDs that result in many test requirements, require more model checker executions than the simpler FBDs. It also takes more time to run the model checker against the more complex FBDs. Consequently, some FBDs (I6 and I7 especially) result in excessively long test generation times.

The mutant kill rates of the generated test sets are illustrated in Fig. 6. The kill rates are generally quite good, reaching 100 % on many FBDs, but on some FBDs the kill rates remain lower. We can see that the test sets generated based on MBC have a weaker mutant kill rates on almost all FBDs. In every FBD the kill rates of MICC based test sets and MCCC based test sets were identical.

Fig. 7 illustrates the average kill rates of the different mutant variants. The kill rates of the TIME-mutants (68 %, 84 %, 84 %) are quite low while the kill rates for NOT-, AND-OR, and FF-mutants are over 97 % for MICC and MCCC.

# 8. Conclusions

In this work, we have defined three structure-based test criteria for FBDs, and developed an automatic technique for designing test cases according to these criteria. Unlike other similar test criteria, the developed criteria focus on delayed input-output dependencies of function blocks. The DELAY function block is one example where such a dependence is relevant but similar examples exist especially among vendor-specific variations of timer function blocks.

We have tested our technique on fictitious FBDs, and a set of vendor-specific real-world industrial FBDs. The tests suggest that the developed technique is scalable to most nuclear domain safety systems. However, on some complex FBDs the time required to generate the tests becomes quite infeasible. In these cases, it could be useful to partition the FBD into smaller subsystems, or otherwise divide the test generation task into smaller sub-tasks.

The results also showed that a vast majority of test requirements were infeasible in the industrial FBDs. This is primarily due to vendor-specific features of the function blocks. We also note that in some cases infeasible test requirements may be indicators of potential defects in the design, and analysis of the infeasible requirements could be a beneficial bug finding technique. We leave this for future work.

The fault detection capability of the method was analysed using mutation analysis. The average fault detection capability of the generated tests was 90 % when the MBC test criterion was used and 95 % when the more rigorous test criteria (MICC, MCCC) were used for test generation. The fault detection capability of the tests generated using the MICC criterion was identical to tests generated using the MCCC criterion. This suggests that it may not be sensible to use the MCCC criterion since the benefits when compared to MICC may be small when compared to the additional computational effort required.

We have not yet addressed the test oracle problem, i.e. the problem of distinguishing the desired and correct behaviour of a system from incorrect behaviours given an input for that system. In the test generation technique we have developed, a set of test cases is created based on a model of that system's design. The generated test sequences, however, can be such that they do not correspond to the desired functionality of the system. This is because the design used for test generation, or the model of that design may be incorrect, resulting in incorrect test sequences. In the ideal case the test generation process should be coupled with a computer-based test oracle (a system capable of distinguishing between desired and incorrect test sequences) that would ensure that the tests always correspond to desired behaviour. Unfortunately, creating such a test oracle for an arbitrary system is very difficult. In practice a human test oracle who manually analyses the generated test cases can be used. In the mutation analysis experiment, the original reference models are assumed correct, so that the generated test cases always represent desired system behaviour.

Threats to *internal validity* of this work might come from errors in the implementation code, the model checking model, and the function block conditions. To reduce possible errors in these components, the implementation was tested on many FBDs, and intermediate products such as data paths and test requirements were manually reviewed. The methodology of this paper also relies on the fact that the system is cyclically run on constant length intervals so that the system behaviour corresponds to the discrete time model checking model. A threat to *external validity* is whether the system can be adequately described in the used modelling language.

Designs including complex mathematical functions cannot be exactly modelled. Also, analogue variable ranges have to be discretised for the model checking tool.

## References

[1] USNRC. Software Unit Testing for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, Regulatory Guide 1.171, 1997.

[2] IEC. *IEC 61131-3 (2013): International Standard for Programmable Controllers — Part 3: Programming Languages*. 1993.

[3] Mahdi Sarabi. Evaluation of structural testing effectiveness in industrial model-driven software development. Master's thesis, Mälardalen University, June 2012.

[4] Eunkyoung Jee, Junbeom Yoo, Sung Deok Cha, and Doohwan Bae. A data flow-based structural testing technique for FBD programs. *Information & Software Technology*, 51(7):1131–1139, 2009.

[5] Eunkyoung Jee, Suin Kim, Sungdeok Cha, and Insup Lee. Automated test coverage measurement for reactor protection system software implemented in function block diagram. In *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'10, pages 223–236, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[7] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, pages 108–125, 2000.

[8] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[9] FBK-IRST, Carnegie Mellon University, University of Genova and University of Trento. NuSMV model checker v.2.5.4, 2012.

[10] John Callahan, Francis Schneider, Steve Easterbrook, et al. Automated software testing using model-checking. In *Proceedings 1996 SPIN workshop*, volume 353. Citeseer, 1996.

[11] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer Berlin Heidelberg, 1997.

[12] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[13] Eunkyoung Jee, Donghwan Shin, Sungdeok Cha, Jang-Soo Lee, and Doo-Hwan Bae. Automated test case generation for fbd programs implementing reactor protection system software. *Software Testing, Verification and Reliability*, 24(8):608–628, 2014.

[14] Yi-Chen Wu and Chin-Feng Fan. Automatic test case generation for structural testing of function block diagrams. *Information and Software Technology*, 56(10):1360 – 1376, 2014.

[15] K. Maruchi, H. Shin, and M. Sakai. MC/DC-like structural coverage criteria for function block diagrams. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 253–259, March 2014.

[16] IEC. *ISO/IEC 29119-4 (2013): Software and systems engineering — Software testing — Part 4: Test techniques*. 2013.

[17] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. Model-based test suite generation for function block diagrams using the UPPAAL model checker. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 158–167. IEEE, 2013.

[18] Eduard P. Enoiu, Adnan Čaušević, ThomasJ. Ostrand, ElaineJ. Weyuker, Daniel Sundmark, and Paul Pettersson. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2014.

[19] Jussi Lahtinen. Automatic test set generation for function block based systems using model checking. In *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*, pages 216–225, Sept 2014.

[20] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela. A toolset for model checking of PLC software. In *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–6, September 2013.