Automated Test Generation using Model-Checking: An Industrial Evaluation

Eduard P. Enoiu[1], Adnan Čaušević[1], Thomas J. Ostrand[3], Elaine J. Weyuker[1], Daniel Sundmark[1 2], and Paul Pettersson[1]

[1] Mälardalen University, Västerås, Sweden.
[2] Swedish Institute of Computer Science
[3] Software Engineering Research Consultant

Abstract. In software development, testers often focus on functional testing to validate implemented programs against their specifications. In safety critical software development, testers are also required to show that tests exercise, or cover, the structure and logic of the implementation. To achieve different types of logic coverage, various program artifacts such as decisions and conditions are required to be exercised during testing. Use of model-checking for structural test generation has been proposed by several researchers. The limited application to models used in practice and the state-space explosion can, however, impact model-checking and hence the process of deriving tests for logic coverage. Thus, there is a need to validate these approaches against relevant industrial systems such that more knowledge is built on how to efficiently use them in practice. In this paper, we present a tool-supported approach to handle software written in the Function Block Diagram language such that logic coverage criteria can be formalized and used by a model-checker to automatically generate tests. To this end, we conducted a study based on industrial use-case scenarios from Bombardier Transportation AB, showing how our toolbox COMPLETETEST can be applied to generate tests in software systems used in the safety-critical domain. To evaluate the approach, we applied the toolbox to 157 programs and found that it is efficient in terms of time required to generate tests that satisfy logic coverage and scales well for most of the programs.

1 Introduction

Advances in model-checking tools and technology in the last decade have made it a pragmatically usable technique for test case generation from finite-state models [13]. There have been a number of approaches used for defining logic coverage using model checkers, e.g., [7, 24, 25], however, these techniques are not directly applicable to real-world programs of critical systems. When industrial software systems are being tested, there is still the issue of potential combinatorial explosion of the state space which thereby limits the application to models used in practice.

Many industrial application domains use safety-critical software to implement the behavior of programmable logic controllers (PLCs). One of the programming languages defined by the *International Electrotechnical Commission* (IEC) for PLCs is the *Function Block Diagram* (FBD). Programs developed in FBD are automatically transformed into program code, which is compiled into machine code by using specific engineering tools provided by PLC vendors. The motivation for using FBDs as a preferred language arises because it is the standard in many industrial software systems, such as in the railway domain. Such systems typically require a certain degree of certification [8], such as some level of logic coverage which must be demonstrated on the developed software. Although all software should aspire to correctness, safety critical software is generally held to a higher standard than other types of systems, which should be reflected in their testing. However, there is no commonly accepted level of test thoroughness for safety-critical software. In this paper, we show how to efficiently generate test cases that achieve several levels of coverage, including MC/DC and decision coverage. It should be noted that the generated tests are not intended to replace requirement-based test design at the FBD program level, but to complement it with a structural perspective.

In our previous work we proposed the use of logical coverage for FBD programs [12] and defined a model-based test generation method based on the UPPAAL tool. While this approach is promising, there is a need to validate it using realistic programs of critical systems. To this end, we conduct an experimental evaluation using 157 programs of a train control system, written in the FBD language. We develop a toolbox, named COMPLETETEST[4], suitable for transforming an FBD program to a formal representation of both its functional and timing behavior. This is done by implementing an automatic model–to–model transformation from FBDs to timed automata. Timed automata, introduced by Alur and Dill [3], were chosen because there is an already existing formal semantics and tool support for simulation and model-checking using UPPAAL [22]. The transformation accurately reflects the data-flow characteristics of the FBD language by constructing a complete behavioral model which assumes a *read-execute-write* program semantics. The translation method consists of four separate steps. The first three steps involve mapping all the interface elements and the existing timing annotations. The final step produces a formal behavior for every standard component in the FBD program. These steps are independent of timed automata and therefore are generic in the sense that they could also be used when translating an FBD program to a different target language. The toolbox uses a test generation technique based on model-checking, tailored for logic coverage of FBD programs. A generated test consists of a sequence of input vectors. As the main purpose of the tool at present is to generate test cases that satisfy coverage criteria, the tool does not generate expected outputs. Expected outputs can be provided manually to the toolbox by a human tester.

The paper is organized as follows. Section 2 provides an overview of PLC software, the IEC 61131-3 standard, timed automata and logic coverage. Section 3 describes the transformation scheme into timed automata. Section 4 and Section 5 present the test case generation method required for logic coverage criteria. In Section 6, we describe the tool box used for testing FBD software and demonstrate its application by showing relevant user scenarios. In Section 7, we evaluate the toolbox on industrial programs in terms of its efficiency and usability. Section 8 describes related work. Section 9 presents our conclusions.

[4] COMPLETETEST is available at `http://www.completetest.org/`.

2 Preliminaries

This paper describes a toolbox for generating tests that cover the logical structure of FBD programs, by transforming them first to networks of timed automata. Our technique will be illustrated throughout this paper using a complete, small, but non-trivial FBD program that exhibits many of the features of FBDs. We show how this program can be translated into a timed automaton and used to illustrate the approach, the toolbox evaluation and its practical implications. In this section, we provide some background details on FBD programs, timed automata and logical coverage.

2.1 Programmable Logic Controllers

PLCs are widely used in real-time software for many types of software systems including nuclear plants and train systems. A PLC is an integrated embedded system that contains a processor, a memory, and a communication bus. The semantics of a program running on a PLC has the following representative characteristics:

- programs execute in a cyclic loop where every cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs).
- Input and output channels correspond to sensors and actuators respectively.

The language can be specified on an implementable subset of timed automata [3]. Dierks [10] proposed a new class of automata suitable for PLCs and this definition is the basis for implementing a model-to-model transformation for PLC software.

FBD, a PLC programming language standardized by IEC 61131-3, is very popular in industry because of its graphical notations and its data flow nature [23]. Blocks in an FBD program form the basis for a structured and hierarchical application. They may be supplied by the manufacturer, defined by the user, or predefined in a library. An application generator is utilized to automatically transform each block to a C compliant program with its own thread of execution. A block cannot be recursive as it cannot call itself [27]. However, blocks may have multiple instances within a program.

Although our description is not limited to a particular PLC software development style for FBD programs, it is exemplified by a generic PLC control application compliant with the IEC 61131-3 standard. A PLC periodically scans an FBD application program, which is loaded into the application memory. As an example of the FBD generic model, we consider first the hierarchical structure of a PLC and the functional integration. An FBD control program is considered as a hierarchical application. The FBD program is created as a composition of interconnected blocks, which may have intra-program data flow communication. When activated, a program consumes one set of input data and then executes to completion. The code is used on the specific PLC and is the actual application code from the IEC 61131-3 compliant FBD program.

The IEC 61131-3 standard proposes a hierarchical software architecture for structuring and running any FBD program. This architecture specifies the syntax and semantics of a unified control software based on a PLC configuration, resource allocation, task control, program definition, function and function block repository, and program code [23, 27].

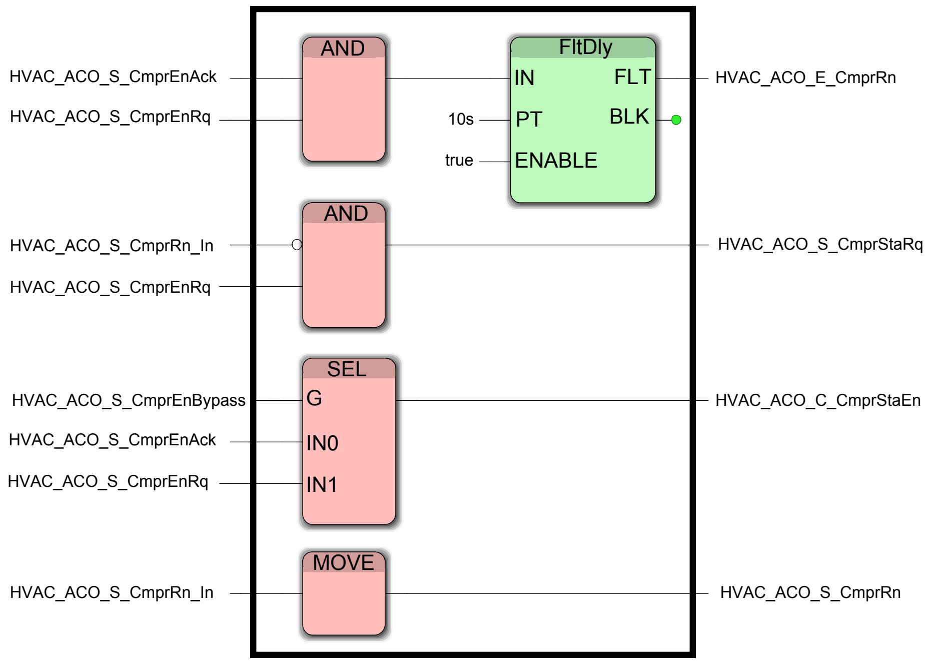


Fig. 1. Running Example: Compressor Start Enable program showing the graphical nature of the language.

The systems we are studying contain a particular type of blocks called *PLC timers*. These timers are output instructions that provide the same functions as timing relays and are used to activate or deactivate a device after a preset interval of time. There are two different timer blocks (i) On-delay Timer (TON) and (ii) Off-delay Timer (TOF). A timer block keeps track of the number of times its input is either true and false and outputs different signals based on these counters. In practice many other time configurations can be derived from these basic timers. In order to study how to generate test cases using a model checker for these types of FBD programs, we use a formal representation that can cope with timers and timing information.

2.2 The Compressor Start Enable Program

The translation scheme, test generation, and logic coverage will be illustrated by translation of a complete, small, but typical FBD program that includes many of the FBD features. Figure 1 contains this FBD program for which we will ultimately generate test cases. It was developed by an engineer from Bombardier Transportation responsible for developing train control software in Västerås, Sweden.

The train is made up of motorized cars and intermediate trailer cars with pantographs. These cars are combined to create a fixed eight car train, each with its own complete software control system that applies regulation to a heating and/or air conditioning system. The task of the train operating the ventilation compressor mode is imposed by the controller FBD program depicted in Figure 1. The program requests permission to start the ventilation compressor from the auxiliary load control. When granted, it will forward the command to the ventilation controller. The Compressor Start Enable will request permission to start the ventilation compressor. When granted, the signals are forwarded to the ventilation controller.

The request will time out (`HVAC_ACO_E_CmprRn`) when the compressor start signal is acknowledged (`CmprEnAck`) and required (`HVAC_ACO_S_CmprEnRq`) provided the clock is greater than or equal to ten seconds. Additionally, the ventilation should be active (`HVAC_ACO_S_CmprRn`) when the compressor is running (`HVAC_ACO_S_CmprRn_In`). The ventilation request is started (`HVAC_ACO_S_CmprStaRq`) when the compressor is enabled (`HVAC_ACO_S_CmprEnRq`) and the compressor is not running (`CmprRn_In`). When the external supply (`CmprEnBypass`) is not available, the compressor should be enabled (`CmprStaEn`) only when the compressor is allowed to start from auxiliary load control (`HVAC_ACO_S_CmprEnAck`). If the external supply is available, then the compressor is enabled.

The program consists of basic functions (e.g., AND, SEL, MOVE) and function blocks (e.g., FltDly). In Figure 1, AND is a function. In contrast, FltDly is a function block because it maintains an internal state and produces outputs based on this state and inputs. Recall that in order to express timing constraints within one component, standard PLC timers are used. The timers in a PLC are operated by an internally generated clock that originates in the processor module. Consider the following PLC timer `FltDly` defined as a tuple:

$FltDly = \langle FltDly_1, (IN, PT, ENABLE, FLT, BLK), B_t \rangle$,

where $FltDly1$ is the name identifier, IN, PT, $ENABLE$, BLK, and FLT are the set of ports and parameters in `Port`, and B_t is the behavior description. This timer component is an attempt to specify its interface and behavior. From a semantic point of view, FBD programs are a special case of deterministic reactive systems. We use more informative notations to denote the actual behavior. In the following section we present several such notations to describe how FBD programs can be handled by the UPPAAL model checker.

2.3 Networks of Timed Automata

A timed automaton is a standard finite-state automaton extended with a collection of real-valued clocks. The model was introduced by Alur and Dill [3] and has gained in popularity as a suitable model for real-time systems. We consider model checking algorithms that perform analysis to check for a reachability property of the form $\exists \Diamond \beta$. $\exists$ is the existential quantifier, $\Diamond$ is the temporal operator meaning eventually, and β is a formula designed to capture the requirements of a particular type of logic coverage. The reachability property states that there exists a path σ through the states of the timed automaton such that β eventually holds. The property is presented to the model checker, which then attempts to find an actual path that satisfies the property. A path σ that satisfies the reachability property can be converted into a test case that satisfies the desired coverage. We use a timed modal logic to specify properties. The logic may be seen as properties that can be expressed as logical formulae in the Timed Computational Tree Logic (TCTL) [2].

An example of a network of timed automata modeled in UPPAAL is shown in Figure 2. The network consists of an automaton of a lamp and an automaton of a user. A network of timed automata in this case can be written as $Lamp \parallel User$. The user operates the lamp by pressing the on/off switch. By pressing the switch, the lamp can be in three possible locations: Off, Dim and Bright. The automaton of the lamp starts at the Switched Off location and contains one clock x. If the user presses the light switch, the lamp switches to Dim and the clock is reset, by the update assignment $x := 0$. When Dim, the lamp remains on as long as the clock is less than or equal to five time units (i.e., invariant $x <= 5$). A state of the automaton Lamp depends on its current location and on the current values of its clocks. If the user presses the light switch before three time units, then the lamp switches to location Bright. In this location, the lamp automaton stays ON until the user presses the light switch again. Both automata synchronize via the actions press! and press? i.e., by sending via channel press! and receiving using press?. Based on the states of the Lamp automaton, one can denote traces starting from the initial state as a sequence of alternating transitions $\sigma = (Off, 0) \xrightarrow{press} (Dim, 0) \xrightarrow{delay} (Dim, 2) \xrightarrow{press} (Bright, 2)$.

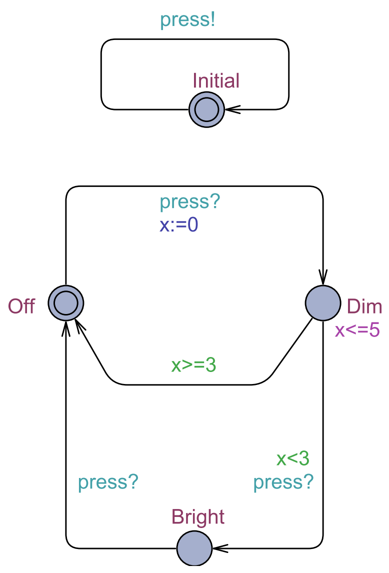


Fig. 2. Example of a network of timed automata.

We provide a brief summary of the notation and concepts in Appendix A, for readers unfamiliar with timed automata theory. Further information can be found in [1].

2.4 Logic-based Coverage Criteria

Coverage criteria are a code-based means of assessing the thoroughness of test cases. They are normally used at the unit test level to check that various aspects of the code structure have been exercised by the test cases. Out of the many criteria that have been defined and studied, we have implemented three logic-based criteria that measure the thoroughness of test coverage of the control flow structure of FBD programs.

FBD program flow is controlled by atomic Boolean expressions called *conditions*, and by *decisions* made up of conditions combined with Boolean operators (NOT, AND, OR, XOR, IMPLIES, EQUIV). A condition can be a single boolean variable, an arithmetic or character comparison with a Boolean value (e.g., $A > B$ or $str1 == str2$), or a call to a function with a Boolean value, but does not contain any Boolean operators. The test generation tool uses the UPPAAL model checker to generate test cases that satisfy three types of logic coverage: *decision coverage DC* (also known as predicate coverage), *condition coverage CC* (also known as clause coverage) and *Modified condition decision coverage (MC/DC)*.

A set of tests satisfies decision coverage if running the test cases causes each decision in the FBD program to have the value *true* at least once and the value *false* at least once. Note that for any individual predicate, the true and false values might occur under a single test case or under two different test cases. In general a single test case will exercise more than one decision, and it is possible, but certainly not required, that all decisions in a program might have both values exercised by a single test case. In the context of traditional sequential programming languages, decision coverage is usually referred to as *branch coverage*.

Condition coverage requires test cases that cause each individual condition to be exercised at least once with value true and once with value false. A set of test cases might satisfy either condition coverage or decision coverage, or both of them. Modified condition decision coverage captures the idea that the value of a decision can be controlled by the value of each of its conditions independently of the values of all other conditions. This means that for each individual condition c in a decision, there are sets of values of all the other conditions so that the decision's value differs for the two possible values of the condition c.

For MC/DC each individual condition in each decision should be shown to be able to determine the outcome of the decision during testing. MC/DC is a stronger requirement than CC; any test set that satisfies MC/DC must also satisfy CC. For most non-trivial decisions, MC/DC is also more strict than DC, even though there are decisions for which a MC/DC-satisfactory test set does not satisfy DC. CC, DC and MC/DC, as well as other logic criteria, are defined and exemplified in [9, 5].

3 Translation

The translation scheme will be illustrated on the running example. After translation, the UPPAAL model checker can be applied to test that the program satisfies the required logic coverage on the FBD program. The translation is performed starting from signals which are translated into global variables shared by the corresponding blocks. Additionally, FBD blocks are mapped to input/output behavior (e.g., functional and timing) between signals. This may be done by using predefined UPPAAL operators, as in the case of *basic blocks* (e.g., `AND, SEL, MOVE`), or by capturing the functionality of more *complex blocks* (e.g., `FltDly`) from their description.

In practice the timed behavior of an FBD program is defined as a network of timed automata, extended with data input and output variables. We first perform an automatic transformation of the FBD program to timed automata that obeys the *read-execute-write* semantics of the FBD program, hence preserving the semantics of FBDs without altering its structure. Next, we specify the execution of each block, and construct a complete timed automata model by the parallel composition of local behaviors.

```
1  plc = plcSupervision();
2
3  readinput1= input_HVAC_ACO_S_CmprEnRq();
4  readinput2= input_HVAC_ACO_S_CmprRn_In();
5  readinput3= input_HVAC_ACO_S_CmprEnAck();
6  readinput4= input_HVAC_ACO_S_CmprEnBypass();
7
8  block1= Function_AND1();
9  block2= Function_AND2();
10 block3= Function_MOVE1();
11 block4= Function_SEL1();
12 block5= Function_FltDly1();
13
14 writeoutput1= output_HVAC_ACO_C_CmprStaEn();
15 writeoutput2= output_HVAC_ACO_S_CmprStaRq();
16 writeoutput3= output_HVAC_ACO_S_CmprRn();
17 writeoutput4= output_HVAC_ACO_E_CmprRn();
18
19 system plc, readinput1, readinput2, readinput3,
20 readinput4, block1, block2, block3, block4, block5,
21 writeoutput1, writeoutput2, writeoutput3,
22 writeoutput4;
```

Fig. 3. Interface elements created from structure and behavioral elements from the Compressor Start Enable.

3.1 FBD Structure

For illustration, we start with the translation of the Compressor Start Enable Program. For each block, a timed automaton is defined for the program description. Templates of components are included and we list the composed timed automata network representing the FBD program as

$$AND_1 \parallel AND_2 \parallel SEL_1 \parallel MOVE_1 \parallel FltDly_1$$

The top-level structure of the UPPAAL model is shown in Figure 3 and represents a parallel composition of several processes corresponding to inputs (lines 3-6), outputs (lines 14-17), and blocks (lines 8-12).

An input named in the program `HVAC_ACO_S_CmprEnAck` will be automatically translated into a timed automata template named `input_HVAC_ACO_S_CmprEnAck()`.

When an input signal in FBD has a name, the name is preserved during translation. However, it is often the case that signals in FBDs are not named (e.g., in the Compressor Start Enable program there are simply "*wires*" connecting two blocks). In such a case, the name given to the signal corresponds to the name of the block which produces the signal. For example, the output signal produced by `Function_AND2()` will correspond to an UPPAAL variable named `bool AND2` as shown in Figure 4.

```
chan execute,write, read;

// variable definition for the FBD program
// US_INT OR BOOL VAR_INPUT

// input variable definition
bool HVAC_ACO_S_CmprEnRq;
bool HVAC_ACO_S_CmprRn_In;
bool HVAC_ACO_S_CmprEnAck;
bool HVAC_ACO_S_CmprEnBypass;

// output variable definition;
bool HVAC_ACO_C_CmprStaEn;
bool HVAC_ACO_S_CmprStaRq;
bool HVAC_ACO_S_CmprRn;
bool HVAC_ACO_E_CmprRn;

// internal intermediate variables
bool AND1;
bool AND2;
bool MOVE1;
bool SEL1;
bool FltDly1;
clock BLK;
bool ENABLE;
const int PT=10;
```

Fig. 4. Input, Output, and Internal Signals translated for the Compressor Start Enable Program.

Several Boolean and integer variables are used for recording information in the UPPAAL model and are shown in Figure 4: `read`, `execute` and `write` synchronization channels are used to hard code the execution of the program, the `BLK` clock variable is used to keep track of the elapsed time in FltDly, other variables (e.g., bool `HVAC_ACO_S_CmprEnRq`) are used for recording the inputs generated by the input automaton, `PT` represents the fault delay, while `ENABLE` records the compressor enable.

3.2 Cycle Scan and Triggering

A block in an FBD has an *interface*, consisting of a name identifier, input and output ports, and a list of parameters. The interface is used to access the block behavior. When the block is activated the *behavior* is started using the values read on the input ports. When the behavior ends, i.e., when the block implementation terminates its execution, the output ports are updated. The behavior of a block is typically implemented by a code fragment that updates local variables. In addition, the program contains a clock variable for modeling a delay between the cycles. We show in Figure 5 how a

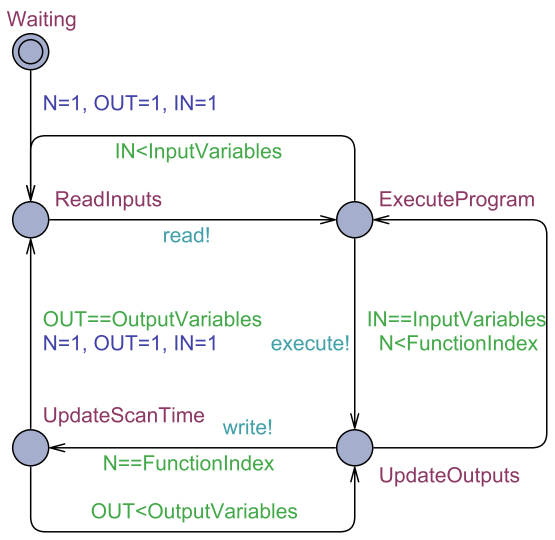


Fig. 5. Timed Automaton of a Program Cycle Scan and Execution Order.

cycle starts when the automaton enters the `ReadInputs` node and ends its computation in `UpdateScanTime` node. For an FBD program, the execute operation of each block is extended according to connections and `IN` and `OUT` variables corresponding to the program inputs and outputs. A program composition is a set of interconnected blocks closed under a specific execution order. The execution order `N` is automatically defined according to the general rules included in the IEC 61131-3 standard. This predetermined order directly dictates the data dependency. Using the program cycle requires deterministic program execution, by restricting the underlying timed transition system. The program is executed in a loop and the computation follows the *run-to completion* semantics. We use the notion of precedence to describe such dependencies on the convention of reading such FBD programs in a top-to-bottom, left-to-right fashion. To show an example of a program cycle scan as shown in Figure 5 different actions are executed:

– `read(IN)` for reading variables from `IN`.
– `write(OUT)` for writing variables onto output ports.
– When the execution order holds, the ports are updated by `read(IN)`, and `write(OUT)`.

For the Compressor Start Enable program, the *execution order* is `AND1`, `FltDly1`, `AND2`, `SEL1`, and `MOVE1`. For each block we assign a precedence priority to the corresponding timed automaton. A counter is created in this step to represent the execution priority of a block. In this way we ensure that block are executed one by one. After the last block is evaluated, the counter is reset to repeat the scan cycle.

3.3 Translation of basic blocks

Simple FBD blocks are translated into predefined UPPAAL operators. In particular:

– The `Logical Operator` blocks are translated using the logical UPPAAL operators `and`, `not`, `or`.
– The `Arithmetic Operator` blocks are translated using the arithmetic UPPAAL operators `+, =, - , /, *`.
– The `Comparison blocks` are translated using the relational operators UPPAAL `<, >, <=, >=, =`.
– The `Selection` blocks are translated using `if-then-else` statements.

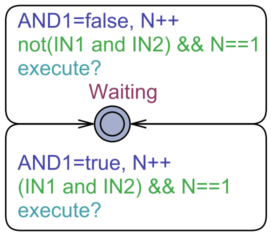


Fig. 6. An automaton showing the AND logical block.

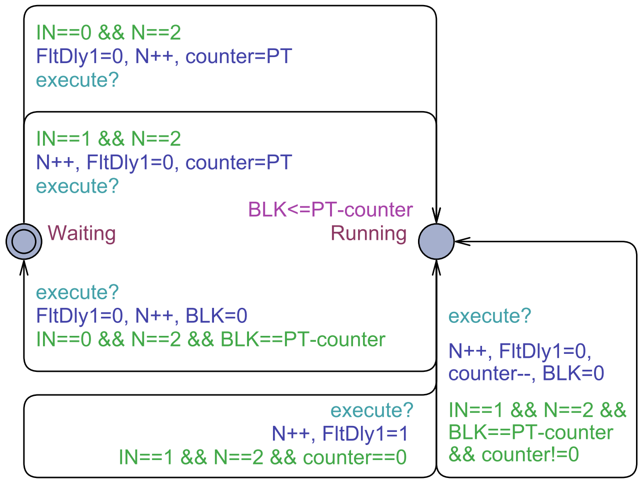


Fig. 7. A Timed Automaton showing a FltDly timer block.

The behavior mapped onto a basic block is modeled as an UPPAAL automaton as shown in Figure 6 for an `AND` logical block. The execution of the translated FBD program is determined in terms of the execution order `N`. A block is therefore initially in location `Waiting`, and after performing the read action it starts executing until its internal computation is done. After completing the write action, which forwards data from the output ports via connections, the block becomes `Waiting` again.

The parallel processes translated for the basic blocks for the Compressor Start Enable program are the following:

- `plcSupervision` The automaton in Figure 5 controls the valid structural information for the other automata. The structure of the FBD program is restricted to reading inputs, execution of the components, and the writing of the outputs.
- `input_name` This automaton non-deterministically generates valid input sequences for the translated blocks. Valid sequences are restricted to Boolean and Integer values.
- `block_AND1 and block_AND2` The automaton in Figure 6 encodes a Boolean AND function by reading the input values and returning a true or false value for the next automaton.
- `block_SEL1` Selects one of two inputs depending on the value of a Boolean input. Then the translation would be:
  `SEL1= if(G=true) then SEL1=IN1 else SEL1=IN0`.
- `block_MOVE1` This automaton is a memory function when we turn on the input port.
- `output_name` The output startup-mode automaton checks the current value received from the function automaton. It also updates the values of the variables `OUT` and `IN`.

More stateful blocks are translated into UPPAAL automata. In particular:

- The `Bistable` blocks (e.g., `SR` and `RS` latches) are elements whose output depends not only on the current inputs, but also on previous inputs and outputs. These blocks can be implemented using logical, relational UPPAAL operators and `if-then-else` statements.
- The `Edge Detection` blocks are translated using UPPAAL expressions involving Boolean operators.
- The `Counters` blocks are translated by the use of UPPAAL `++` increment and `--` decrement operators.
- The `Timer` blocks are translated as a special automaton that is initially in location `Waiting`. After reading its inputs, it starts executing in location `Running` until its internal computation is done. After computing the on-delay timer, it forwards data to output ports and the block becomes `Waiting` again. One example of a `Timer` block from the Compressor Start Enable program is shown in Figure 7. The `FltDly` automaton counts time-based intervals when the input is true and activates its output after a preset interval of ten seconds. The cycle scan interacts with the timer block via the `execute?` action. The timer sets the output `FltDly1` to true if `IN` variable is true at least as long as the time `PT` and `ENABLE` are set to true.

4 Testing Function Block Diagram Software using the UPPAAL Model-Checker

In this section, we describe an approach to automatically generating tests for FBD programs. Logic coverage criteria are used to define what test cases are needed and we use a model-checker to generate test traces. In addition, the methodology presented in this paper is tailored for FBD programs, and is composed of the following steps, mirrored in Figure 8:

1. *Model Transformation* To test an FBD program we map it to a finite state system suitable for model checking. In order to cope with timing constraints we have chosen to map FBD programs to timed automata.
2. *Logic Coverage Annotation* We annotate the transformed model such that a condition describing a single test case can be formulated. This is a property expressible as a reachability property used in most model checkers.
3. *Test Case Generation* We now use the model-checker to generate test traces. To provide a good level of practicality to our work, we use a specific model-checker called UPPAAL which uses timed automata as the input modeling language [5]. The verification language supports reachability properties. In order to generate test cases for logic coverage of FBD programs using UPPAAL, we make use of UPPAAL's ability to generate test traces witnessing a submitted reachability property [15]. Currently UPPAAL supports three options for *diagnostic trace generation*: some trace leading to a goal state, the shortest trace with the minimum number of transitions, and fastest trace with the shortest time delay.

While UPPAAL is a viable tool for model checking, it is not directly tailored to test case generation in practice. We demonstrate how to work around this by automatically generating traces for logic coverage of the control flow of FBD programs described in timed automata and how we transform these traces to actual test cases. We discuss these steps in further detail in the following sections.

As a result of the translation described in Section 3.3, we consider that the FBD program is given as a closed network of timed automata as shown in Figure 9. This model contains two sub-networks, one modeling the `FBD Program` and the other one modeling its `Input` and `Output` Model. In addition, we consider a completely unconstrained input environment that allows all possible interactions between the timed automata network elements. In this way the cycle scan is used to control the FBD program via `read!`, `execute!`, and `write!` actions. This corresponds to synchronization actions implemented in UPPAAL as a hand-shaking synchronization: two automata take a transition at the same time, one will have an a! and the other an a?, a being the synchronization channel.

Let us assume the generic timed automata network of the Compressor Start Enable program together with its cycle scan (`plcSupervision()`) and Input/Output models shown in Figure 9. A trace produced by the model checker for a given reachability property defines the set of actions executed on the Compressor Start Enable program which in our case is considered the system model *sys*. An example of a diagnostic trace has the following form:

$$(sys_0) \xrightarrow{a_1} (sys_1) \xrightarrow{a_2} \ldots \xrightarrow{a_n} (sys_n),$$

where (sys_k) are states of the FBD program and PLC supervision with input environment constraints, respectively, and a_k are either internal synchronization actions, time-delays or `read!`, `execute!`, and `write!` global synchronizations. For FBD programs,

[5] The UPPAAL tool is available at `http://www.uppaal.org`.

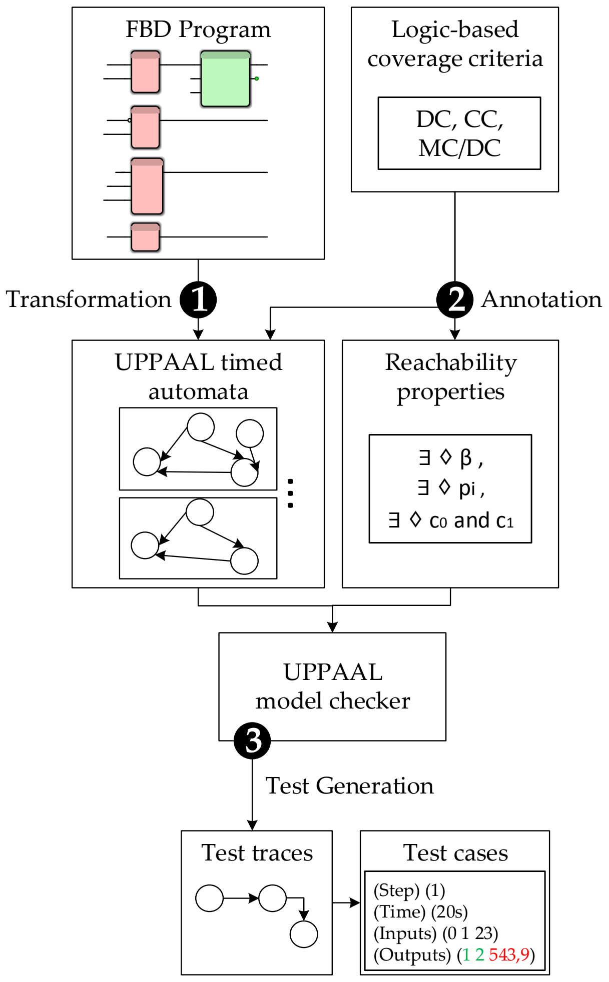


Fig. 8. Testing Methodology Roadmap

the sequence represents only the global synchronizations shown in Figure 9. Test cases are obtained by extracting from the test path the observable actions *read!*, *execute!*, and *write!*. Obviously all the test obligations cannot be satisfied by a single test case. By using a scan cycle we allow the test to be implemented as one or more paths separated by resets. To introduce resets in the model, we annotate the cycle scan with a reset transition leading to the initial `ReadInputs` location. On this transition all variables and parameters (excluding encoded internal variables) are reset to their default value. This reset is hardcoded into the PLC supervision for any modeled FBD program in UPPAAL, being an atomic communication between all timed automata.

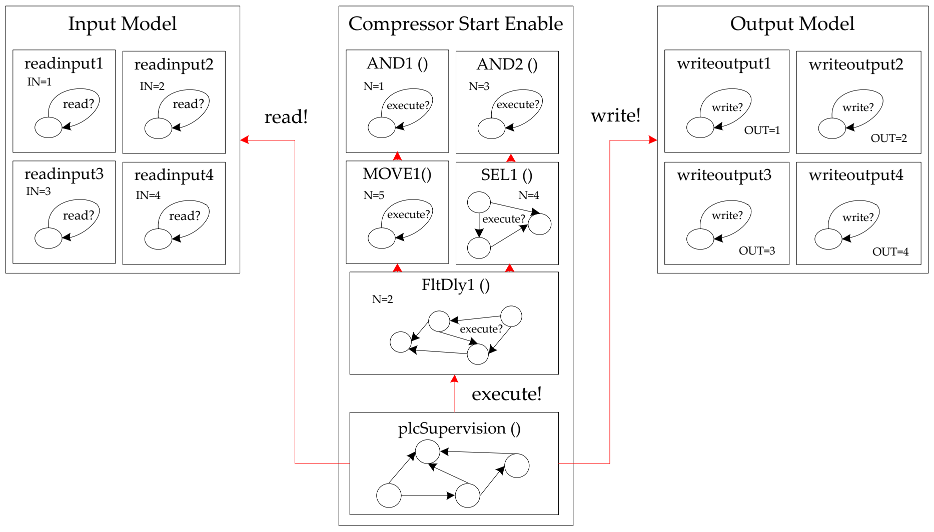


Fig. 9. Timed Automata Network of the Compressor Start Enable Program.

5 Analyzing Logic Coverage

The basic approach to generating test cases for logic coverage using model-checking is to define a test as a finite execution path. By characterizing a logic coverage criterion as a temporal logic property, model-checking can be used to produce a path for the test obligation.

Ammann et al. [4] argued that criteria such as logic coverage that have constraints involving more than one test trace cannot be handled in this way. The core problem is that each execution is characterized by a temporal formula, and test obligations span multiple runs of the model checker. This means that to ensure model-checking of MC/DC test obligations one should satisfy constraints on multiple runs of the model-checker. However, an FBD program has an implicit control loop, so a reset transition can occur in the program without modifying the transformed timed automata in any way. This reset transition restores the program to its initial state, making it possible to handle test obligations over multiple program executions as a single execution path containing subpaths separated by resets.

By using a translated FBD program, we use logic coverage to directly annotate both the model and the temporal logic property to be checked. We propose the annotation with auxiliary data variables and transitions in such a way that a set of paths can be used as a finite test sequence. In addition, we propose to describe the temporal logic properties as logic expressions satisfying certain logic coverage criteria. Informally, our approach is based on the idea that to get logic coverage of a specific program, it would be sufficient to (i) annotate the conditions and decisions in the FBD program, (ii) formulate a reachability property for logic coverage, and (iii) find a path from the initial state to the end of the FBD program. To apply the criteria, necessary properties for the integration of logic coverage need to be fulfilled.

For each criterion, model checking allows the generation of paths for logical predicates showing test obligations satisfaction. To do so, conditions and decisions have to be formulated as temporal logic formulae. Hessel et al. [16] proposed one way to apply coverage criteria to specifications described in timed automata. We extend this approach to apply it to the conditions and decisions in an FBD program.

Decisions in an FBD program are blocks that can be evaluated to a Boolean value, i.e., true or false. Decisions can be identified from the instrumentation points in the FBD program (e.g., AND block). Let $\{d_i\}$ be the set of decisions in an FBD program and $\{c_{ij}\}$ be the set of conditions in d_i.

DC requires every d_i to evaluate to true and false, and is described by the following two test obligations:

$$o_1 = d_i$$

$$o_2 = \neg d_i$$

These obligations guarantee that each decision d_i evaluates to both true and false, not necessarily along the same execution path.

CC requires two test obligations for each clause c_{ij} in a decision d_i, such that c_{ij} evaluates to both true and false:

$$o_1 = c_{ij}$$

$$o_2 = \neg c_{ij}$$

MC/DC imposes two requirements for test cases. First, for each condition c_{ij} in a decision d_i, test cases must show that c_{ij} determines the value of decision d_i, and second, c_{ij} has to evaluate to true and false. As shown in [5], a condition c_{ij} determines a decision d_i if there is an assignment of values to all the variables in d_i except c_{ij} such that the value of d_i is different for the two values of c_{ij}. This requirement is met if the following logical expression is satisfied [6]:

$$d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)}$$

Combining the two requirements for MC/DC coverage, we have the following two test obligations:

$$o_1 = c_{ij} \wedge (d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)})$$

$$o_2 = \neg c_{ij} \wedge (d_{i(c_{ij},true)} \oplus d_{i(c_{ij},false)}).$$

For generating tests for DC, CC, and MC/DC we represent the test obligations over a set of variables monitoring the decisions and conditions as a reachability property. This approach is implemented in the toolbox by automatically creating a temporal logic property used by the model checker to produce tests.

6 Overview of the Toolbox

In this section we outline some of the main aspects of the toolbox, including the user interface and the architecture. We also present several technical solutions used in its implementation to fully support the complexity required for model-checking while at the same time presenting a clean and minimal user interface.

[6] $d_{i(c_{ij},v)}$ denotes d_i with c_{ij} replaced with v.

6.1 User Interface

The main goal for the design of the user interface was to meet the exact needs of an industrial end user. Although there is a possibility for fine tuning the configuration parameters of the underlying UPPAAL model-checker, most of them are set to default values, making the toolbox immediately ready for use upon startup. Figure 10 depicts menu options for the toolbox, listing chosen default values for the parameters and the coverage criteria.

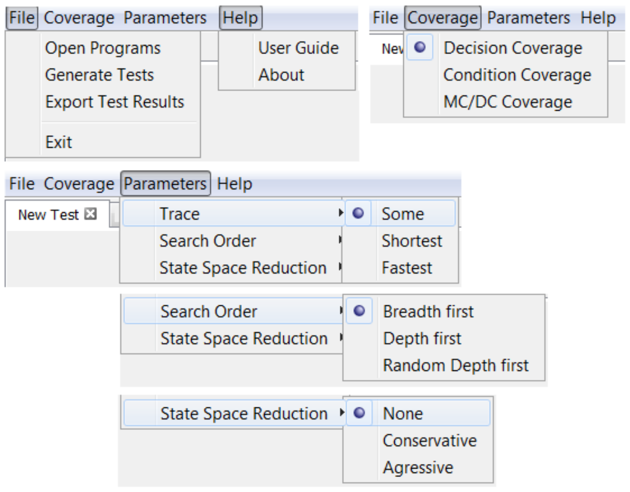
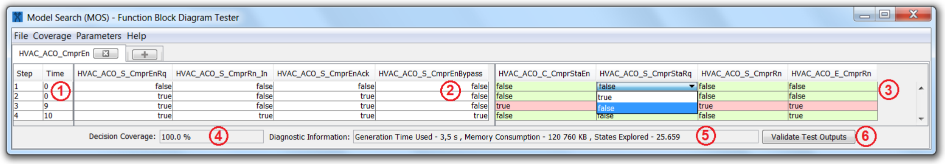


Fig. 10. User Menu of the Toolbox


Model Search (MOS) - Function Block Diagram Tester
File Coverage Parameters Help
HVAC_ACO_CmprEn
1
2
3
4
5
6

Step	Time	HVAC_ACO_S_CmprEnRq	HVAC_ACO_S_CmprRn_In	HVAC_ACO_S_CmprEnAck	HVAC_ACO_S_CmprEnBypass	HVAC_ACO_C_CmprStaEn	HVAC_ACO_S_CmprStaRq	HVAC_ACO_S_CmprRn	HVAC_ACO_E_CmprRn
1	0	false	false	false	false	false	false	false	false
2	0	true	false	true	false	false	[illegible]	false	false
3	9	true	false	true	false	true	[illegible]	true	true
4	10	true	true	true	true	false	false	false	true

Decision Coverage: 100.0 %
Diagnostic Information: Generation Time Used - 3,5 s , Memory Consumption - 120 760 KB , States Explored - 25.659
Validate Test Outputs

Fig. 11. Graphical Interface of the Toolbox

Use-Case Scenario 1: Basic Test Generation

A very basic use-case scenario to get started with the toolbox would consist of:

1. Opening an FBD Progam XML file (File → Open FBD Programs)
2. Generating tests (File → Generate Tests)

These actions cause the tool to attempt to generate a set of test cases that cover all of the decisions. The attempt continues until either all decisions have been covered, or the tool has run for 10 minutes even if there are decisions still uncovered. We found that pragmatically, when the toolbox is applied to FBD programs produced at Bombardier Transportation AB, the model checker has been able to generate tests in 0.05 to 133 seconds. Figure 11 depicts an output of the toolbox for this use-case scenario executed on our running example (as defined in Section 2). The figure shows several types of information presented to the user in a table with the test data (points 1,2,3 in the figure), and a set of additional information and actions (points 4, 5 and 6 in the figure). The numbered points in the figure are:

1. Steps and Timing information regarding when the specific test data is provided to the running FBD program.
2. Generated test input data needed to achieve a maximum coverage of the given program.
3. Editable area of the test output data, where the user can provide expected outputs for a specific set of test inputs based on a defined behavior in the requirement. To maintain efficient use of space in the toolbox, expected values for test outputs are provided in the form of a drop-down selection list for boolean values (true/false) or as a text field for other non boolean values (integers, doubles, etc.).
4. Percentage of the logic coverage achieved by using generated tests.
5. Diagnostic information with respect to the time spent on generating tests, memory usage and size of the state space.
6. Optional action to compare expected values with computed ones. Invoking the ”Validate Test Items” button causes the entries in section 3 of the test data table to be colored with green where the expected value matches the computed one, and with red where there is a mismatch. Any subsequent updates to the expected values will automatically update the coloring of that entry.

Use-Case Scenario 2: Selecting A Logic Coverage Criterion

Tests generated using *Use-Case Scenario 1* aim at achieving maximum decision coverage. If a user would like to use a logic coverage measurement other than the default decision coverage (DC), this can be selected from the ”Coverage” menu. Table 1 presents test inputs for the running example when the toolbox is using both decision and condition coverage. Since the running example includes a timer function block (`FltDly`), achieving maximum decision coverage is possible only if we provide test inputs for a certain number of time units. In the running example, the `FltDly` function block expects an input value to be *true* for at least 10 seconds. This is why inputs to the program are set to *true* in steps 2, 3 and 4 for decision coverage representing the state of the system at *time=0*, *time=9* and *time=10*. Since there are no observable changes in the way the system behaves between *time=1* and *time=8*, the toolbox does not display those test steps. For an industrial user, this minimization of test steps is very important, because it saves manual effort in providing expected output values for the system under test.

Table 1. Test inputs generated for Decision Coverage (DC) and Condition Coverage (CC) on the running example. In order for decisions to achieve a certain state, test inputs have to be provided for several time units due to the usage of a timer.

Logic Coverage Criteria	Step	Time	HVAC_ACO_S_CmprEnRq	HVAC_ACO_S_CmprRn_In	HVAC_ACO_S_CmprEnAck	HVAC_ACO_S_CmprEnBypass
Decision Coverage (DC)	1	0	false	false	false	false
	2	0	true	false	true	false
	3	9	true	false	true	false
	4	10	true	true	true	true
Condition Coverage (CC)	1	0	false	false	false	false
	2	0	true	true	true	true

Use-Case Scenario 3: Changing Configuration Parameters

In addition to the basic use-case scenario of the toolbox, a user can perform various configuration changes to the way tests are obtained. This is done by modifying the model-checker's settings in the "Parameters" menu of the tool-box. For example, the user can set the search algorithm to be "Breadth First", and/or set the output trace of the model-checker to a "Fastest" one, etc.

Use-Case Scenario 4: Fault Detection in FBD Programs

This example compares the expected values and computed values produced by the program. We created a typical fault in the Compressor Start Enable program, by removing the negated input for the AND block corresponding to the compressor running (`HVAC_ACO_S_CmprRn_In`). Then we generated tests that satisfy DC for both the original program (assumed to be correct) and the faulty program, as shown in Table 2 (only three signals are shown because these are the inputs that affect the output). For the original program we observe that the specification described in Section 2.2 agrees with the actual output and therefore in all cases (step 1-4) the output is green. Now by examining the output of the faulty program, the user can determine that the ventilation request is not started (`HVAC_ACO_S_CmprStaRq`) when the compressor is enabled (`HVAC_ACO_S_CmprEnRq`) and the compressor is not running (`HVAC_ACO _S_CmprRn_In`), revealing a bug in the program.

Table 2. Manual fault discovery by checking the output (no negated input signal for the AND block in Compressor Start Enable Program). When generating tests with DC for a faulty program, the Compressor Start Request signal will indicate an erroneous false status when the Compressor is not running and there is a request for enabling the compressor.

Step (number of tests)	Time	HVAC_ACO_S_CmprEnRq	HVAC_ACO_S_CmprRn_In	HVAC_ACO_S_CmprStaRq
1 (Original Program)	0	false	false	🟩 false
2	0	true	false	🟩 true
3	9	true	false	🟩 true
4	10	true	true	🟩 false
1 (Faulty Program)	0	false	false	🟩 false
2	0	true	false	🟥 true
3	9	true	false	🟥 true
4	10	true	true	🟥 false

Use-Case Scenario 5: Exporting Test Results

As a final use-case scenario of the toolbox, a user can export the resulting tests in a comma separated values (CSV) format by selecting ”File → Export Test Results”. In this case, all the information from the test data table is saved, including both computed and expected (i.e., user provided) output values. Such data could be used outside of the toolbox for creation of a custom test report.

6.2 Toolbox Architecture

An overview of the toolbox architecture is presented in Figure 12. The actual toolbox was developed as a Java Swing application using the NetBeans integrated development environment and following a modular approach in the design of the toolbox architecture. This resulted in the following modules being part of the toolbox:

FBD Import Editor
This module is used for validating whether the structure of a provided XML file represents a valid PLCOpenXML file containing an FBD Program.

Translation Plugin
Once the FBD Import Editor module has been executed, the PLCOpenXML file

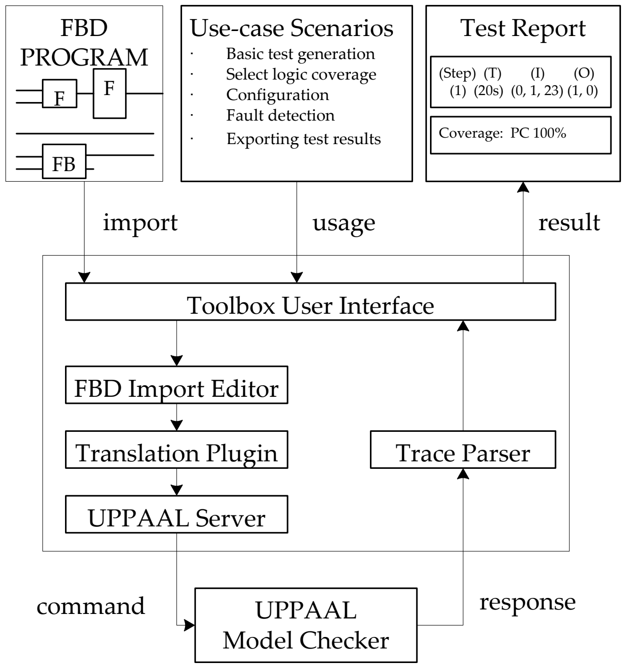


Fig. 12. Overview of the Toolbox Architecture.

containing the FBD Program is translated into an XML-format accepted by the UPPAAL model checker. This translation is carried out by following the rules of translation defined in Section 3.3.

UPPAAL Server

The UPPAAL Server module is used for external invoking of the UPPAAL model checker. UPPAAL provides support for formal verification using a client-server architecture, allowing the toolbox to connect as a client to the model checker and verify properties against the model.

Trace Parser

The Trace Parser toolbox module collects diagnostic trace output from the UPPAAL model checker and parses this output into a JavaCC structure corresponding to a set of inputs and outputs for a given model. This parsing mechanism is further explained in Section 6.5.

User Interface

The function of the user interface is to provide a way for the user to communicate with the tool including: (1) the selection of which FBD program to import and generate tests for, (2) the selection of the coverage criterion to be used for test generation, (3) the presentation of generated test inputs, and (4) the determination of correctness of the result produced for each generated test by comparing the actual test output with the expected output (as provided manually by the tool user).

6.3 PLCOpen XML Standard

The PLCOpen XML interchange format for PLC applications is the base for the model translation to timed automata. PLCOpen is a vendor independent standard aiming to provide a common programming interface for the use of the IEC 61131-3 standard. In the toolbox, the XML file used as input for the translation to timed automata is in accordance to the PLCOpen standard defining the FBD programming language. Figure 13 depicts an example of a PLCOpen XML file corresponding to the Compressor Enable Program. The program consists of specific XML elements consisting of the program name (lines 5), the interface information (lines 6-20), and the block specification for `AND` and `FltDly` (lines 22-53). The XML scheme is mainly storing program information such the *identifier* for blocks and *dependencies*. As shown in Figure 13, `localId` indicates the identifier of a block, and every `refLocalId` in the `connection` tag represents the dependency identifier for the connection to a certain block or input variables. This structural format is used in the implemented translation from FBD to timed automata.

6.4 Implemented Model Translation

We define a translation inside the toolbox, which consists of the formal definition of the FBD language. A program consists of the following elements: composite programs, basic blocks, library blocks, connections, ports, and timing constraints.

The toolbox considers that each modeling element, except for the composite programs, has a set of ports through which it can exchange data. Ports are associated by a set of data types, which are used for data representation, e.g., integer with a specific range. A Port is associated with the same type of data as the associated internal variable.

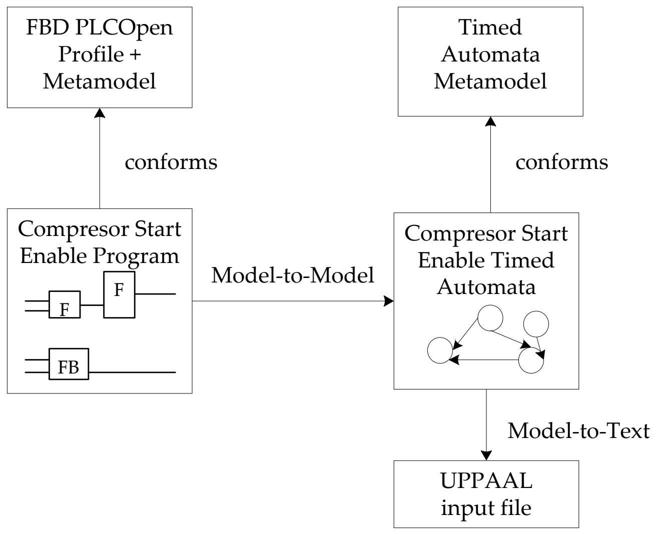
For an FBD program the read-execute-write semantics means that input ports may only be accessed at the beginning of each computation, and output ports are only written at the end of the computation. Therefore, the behavior is augmented with an external interface. The interface of a block consists of ports and the execution order information. An input port has an associated variable holding the current data values. The internal computation of a block starts with reading all input ports. This internal data is used together with the behavioral model during execution, before writing the variables to the output ports.

We have developed the model transformation shown in Figure 14. In order to simplify the semantics of an FBD program, we focus on the PLCOpen language constructs relevant to functional and timing modeling elements.

The PLCOpen language is implemented as an XML profile that provides the ability to describe FBD programs using this profile. The PLCOpen language provides both structural and graphical information needed for implementing the actual translation. The toolbox generates PLCOpen files in an XML format. As shown in Figure 14, we introduce the timed automata as the interface between the FBD program and the UPPAAL input model. The Compressor Start Enable Program conforms to the PLCOpen

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="www.plcopen.org/xml/tc6.xsd">
<types><dataTypes/>
<pous>
  <pou name="HVAC_ACO_CmprEn" pouType="fBlock">
    <interface>
      <inputVars retain="false">
       <variable name="HVAC_ACO_S_CmprEnRq">
         <type><BOOL/></type>
       </variable>
       <variable name="HVAC_ACO_SCmprRnIn">
         <type><BOOL/></type>
       </variable>
      </inputVars>
      <outputVars retain="false">
       <variable name="HVAC_ACO_C_CmprStaEn">
         <type><BOOL/></type>
       </variable>
      </outputVars>
    </interface>
    <body><FBD>
      <block typeName="AND" localId="11">
        <inputVariables>
         <variable formalParameter="IN1"
                           negated="true">
           <connection refLocalId="14"></connection>
         </variable>
         <variable formalParameter="IN2"
                            hidden="true">
           <connection refLocalId="13"></connection>
         </variable>
        </inputVariables>
        <inOutVariables/>
        <outputVariables>
         <variable formalParameter="OUT"
                            hidden="true">
         </variable>
        </outputVariables>
      </block>
      <block typeName="FltDly" localId="66">
        <inputVariables>
          <variable formalParameter="IN">
            <connection refLocalId="18"
                   formalParameter="OUT">
            </connection>
          </variable>
          <variable formalParameter="PT">
            <connection refLocalId="21"/>
          </variable>
          <variable formalParameter="ENABLE">
            <connection refLocalId="22"/>
          </variable>
        </inputVariables>
        <inOutVariables/>
        <outputVariables>
          <variable formalParameter="FLT"></variable>
          <variable formalParameter="BLK"></variable>
        </outputVariables>
      </block>
    </FBD></body>
  </pou>
</pous>
</types>
<instances><configurations/></instances>
</project>
```

Fig. 13. PLCOpen XML format for the Compresor Enable Program



FBD PLCOpen Profile + Metamodel
conforms
Timed Automata Metamodel
conforms
Compresor Start Enable Program
F
F
FB
Model-to-Model
Compresor Start Enable Timed Automata
Model-to-Text
UPPAAL input file

Fig. 14. Model Export from an FBD Program to UPPAAl Model Checker.

profile and meta-model. The structural translation described in Section 3.3 maps an FBD program into timed automata. The structure of the timed automata model is the basis of the model to text transformation into the UPPAAL input model.

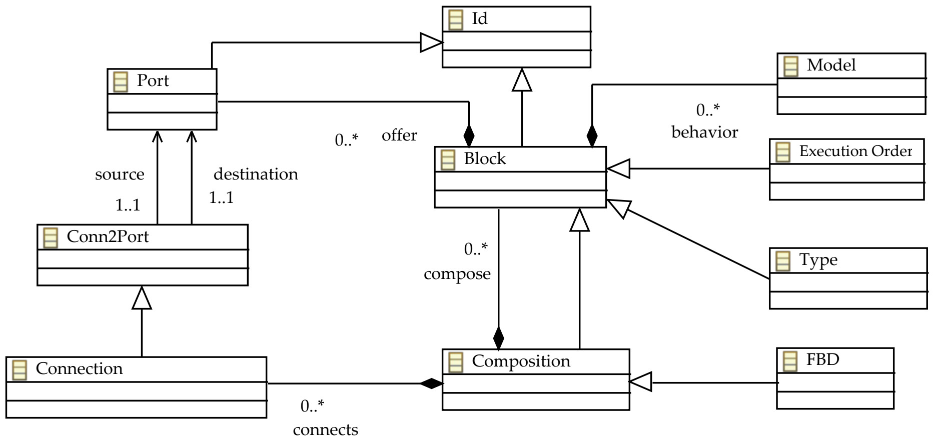


Fig. 15. Class Diagram representing the meta-model elements of the Function Block Diagram.

The modeling elements of an FBD program used in the translation are described in Figure 15. These elements represent the structure of the model, the behavior, and the timing information. The meta-model elements provide concepts used in component based design. A Block element can be translated with Type, ExecutionOrder and Model elements. Blocks can be composed using connections and ports. Furthermore, a Block element can have a behavioral description as a Model element. The model provided after the translation represents the model annotated with triggering and timing information with assumed functionality.

6.5 Dynamic Traces - JavaCC - Test Cases

```
State
(
plc.ExecuteProgram readinput1.Process
readinput2._id10 readinput3.Process
readinput4._id12  and1.Update
and2._id18 fltdly1.Waiting
sel1._id15  move1._id16
writeoutput1._id5 writeoutput2._id6
writeoutput3._id7 writeoutput4._id8
)

fltdly1.ET<=0 steps=1 HVAC_ACO_S_CmprEnRq=1
HVAC_ACO_S_CmprRn_In=1 HVAC_ACO_S_CmprEnAck=0
HVAC_ACO_S_CmprEnBypass=1 HVAC_ACO_C_CmprStaEn=0
HVAC_ACO_S_CmprStaRq=0 HVAC_ACO_S_CmprRn=0
HVAC_ACO_E_CmprRn=0

AND1=0 AND2=0 FltDly1=0 SEL1=0 MOVE1=0

N=1 IN=5 OUT=1

decisions[0]=0 decisions[1]=0 decisions[2]=0
decisions[3]=0 decisions[4]=0 decisions[5]=0
decisions[6]=0 decisions[7]=0 decisions[8]=0
decisions[9]=0

fltdly1.counter=0 move1.firstTime=0
move1.RS_local=0 move1.decision=0

Transitions:
 plc.ExecuteProgram->plc.UpdateOutputs
{ IN == InputVariables, execute!, 1 }

and1.Update->and1.Update
{ !(HVAC_ACO_S_CmprEnAck && HVAC_ACO_S_CmprEnRq) &&
 N == 1, execute?, AND1 := 0, N++, decisions[0] := 1
}
```

Fig. 16. An excerpt of a trace in response to a command to UPPAAL for the Compressor Enable Program.

UPPAAL model-checking tool is mainly used for the verification of a certain property of a model, resulting in a affirmative or a negative response. However, it is also possible to obtain a full trace used in the process of verifying that property on a model. An excerpt of such a trace for the running example is shown in Figure 16. To interpret dynamic traces generated by UPPAAL, a grammar file was created for JavaCC[7] parser generator. The trace starts with the initial state and is followed by pairs of transitions and states, i.e. the state can be reached from the previous state via the transition. A state in the trace contains locations (lines 3-9), clocks (line 12), internal variables (lines 12-20), decisions and conditions (lines 22-25) in the same order as they appear in the UPPAAL input file. The trace parsing using JavaCC is the process of analyzing the trace, transforming the trace into a state machine, extracting the necessary information (i.e., values of the input and output variables, clock valuation) needed for testing of an FBD program. In the end tests are merged based on the program cycle scan as one or more test cases separated by resets.

7 Experimental Evaluation and Discussions

Our goal in this section is to evaluate the toolbox on industrial FBD programs and to acquire experience regarding its efficiency and usability. We therefore conduct a set of analyses using programs developed by Bombardier Transportation AB in Sweden. The system has been in development for more than two years and uses processes influenced by safety-critical requirements and regulations including the EN 50128 standard [8] which requires different logic coverage levels (e.g., DC and MC/DC). In 2014 its source code was made up of more than 350.000 lines of C code generated from FBD programs. The development teams use both automated and manual testing from unit testing through system testing.

Table 3. Information about the 157 subject programs.

	Blocks	Inputs	Outputs	Decisions
Maximum per Program	32	15	29	196
Average per Program	6.9	2.7	5.9	30

We investigate the following questions regarding the tool's performance:

- *Q1, Efficiency:* What is the time required for the tool to generate tests that satisfy the DC, CC and MC/DC logic coverage criteria?
- *Q2, Coverage:* How close does the tool come to generating tests that achieve 100% coverage of each of the criteria?

The industrial system studied in this paper is the TCMS (Train Control and Management System), developed by Bombardier Transportation AB engineers, which has been deployed to the field. In this research we, have used all TCMS programs written in the FBD standard language resulting in a total of 157 artifacts. Each of the programs is sizable and representative of industrial programs used in the train system's development. Information regarding the size of the system and number of blocks is provided in Table 3.

[7] The JavaCC™is available at `https://javacc.java.net/`.

For each program, the tool generated a model version in UPPAAL. Then, for each implementation of a program, the toolbox:

– *Generated test input vectors for three different coverage criteria.* We used a reachability-based approach for generation of tests aimed at satisfying DC, CC and MC/DC. If the model checker is able to find a path to satisfy a reachability property, given that such a path exists, then the approach is guaranteed to generate a test suite that achieves maximum possible coverage of the program.

Table 4. Average, median, minimum, and maximum generation times for 123 of the 157 programs.

	CC	MC/DC	DC
Average Generation Time (s)	1.53	4.54	1.93
Median Generation Time (s)	0.27	0.51	0.34
Minimum Generation Time (s)	0.05	0.06	0.06
Maximum Generation Time (s)	35.37	133.60	72.125

Hence, if the model checker succeeds in finding paths to satisfy all the reachability properties for a given criterion, then the method will achieve 100% coverage for that criterion. We have used the UPPAAL model checker in our experiments. Our reachability-based test generation approach produces one test for each coverage criterion as our goal is to assess the coverage and efficiency of the toolbox in terms of time to generate tests. To generate the tests, the tool uses the random-depth first search algorithm provided by the UPPAAL model checker. The tool terminates the generation by determining the coverage requirements satisfied by each test.

– *Assessed efficiency of each test based on coverage, and collected complexity measures for each program.* We measured the generation time for each program and determined the number of test requirements for each coverage criterion.

To answer Q1 and Q2, the tool generate tests aimed at achieving maximum logic coverage. Since we are using a model checker for generating tests, the toolbox simply produces the maximum achievable coverage with a proof that uncovered test obligations are not coverable. For 123 of the 157 programs (78%) the tool provided tests that covered 100% of the required entities for each of the three coverage criteria. Table 4 gives the performance figures in terms of time needed to generate the tests. The generation time for MC/DC averaged approximately twice as long as for DC. The results are summarized as boxplots in Figure 17 with the kernel density distribution of the generation time shown in Figure 18. The kernel densities estimates for the generation time for DC (red), CC (green) and MC/DC (blue) are plotted on the same graph. It is quite clear on the graph that the distribution of generation times is more variable for MC/DC. It is also worth noting that the generation time modes (i.e., most frequent values in the generation time data set) of a distribution are close to each other for all criteria. We can observe that a few outliers caused the average generation time to greatly exceed the median generation time for all coverage criteria.

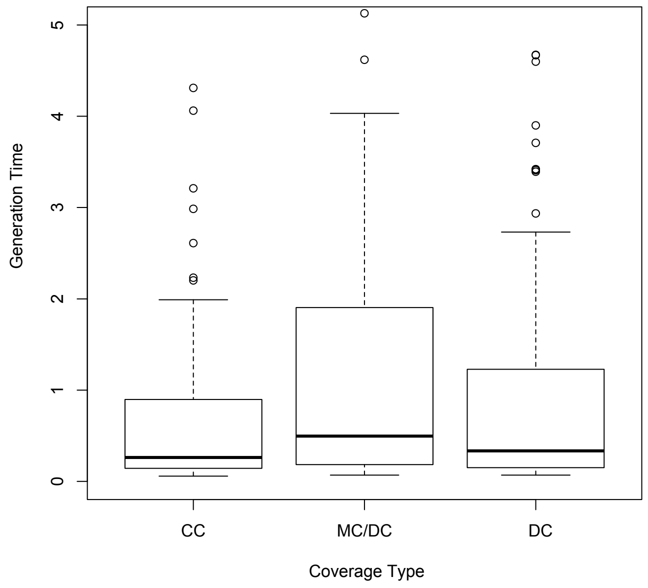


Fig. 17. Experimental results: Generation Time Distributions.

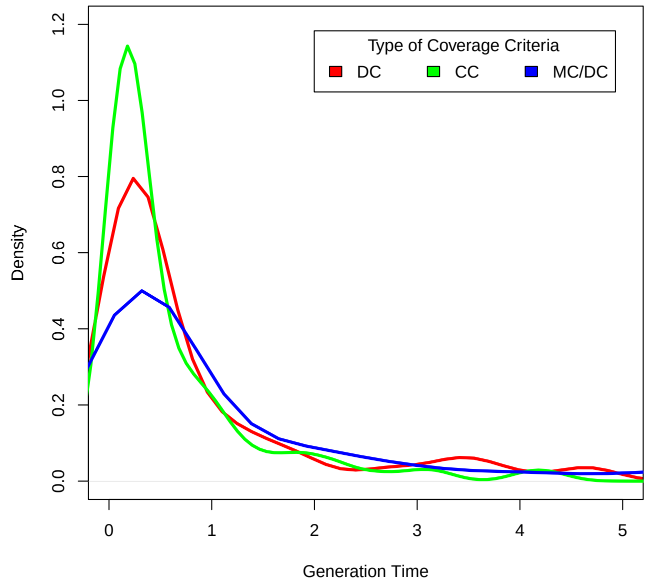


Fig. 18. Generation Time Distribution by Coverage Criteria.

For 34 of the 157 programs, the tool did not terminate after running for a substantial period of time. After discussions with engineers from Bombardier Transportation AB regarding the needed time for a tester to provide a set of tests for a desired coverage, we concluded that 10 minutes was a reasonable cut-off point for the model checker to terminate its search. Recall, however, that the aim of these experiments was not to provide measures of test effectiveness in the sense of bug-finding, but instead to evaluate the applicability of using a model checking technique for test generation and its success in meeting coverage requirements. We wanted to work with a realistic cut-off time that could be used in practice if this approach is to be adopted. Therefore, in each case a run of the model checker was terminated after 10 minutes.

Table 5. Achieved coverage for all Programs.

Case	1	2
Percentage of all Programs	78%	22%
Average DC Achieved	100%	82%
Average CC Achieved	100%	88%
Average MC/DC Achieved	100%	65%

As noted above, for 22% of the programs in this study, the tool did not generate the required test suite in an acceptable period of time. To determine the circumstances under which the toolbox does or does not successfully generate test suites that satisfy one of the logic coverage criteria (Q2) we collected the average number of decisions for both the case when the model checker finishes its execution (Case 1) and the case when we forcefully terminated the tool because the running time reached 10 minutes (Case 2). Table 5 provides information about these two cases. Case 1 consists of the 78% for which the tool generated tests achieving 100% DC, CC and MC/DC. The number of decisions for Case 1 ranged from 1 to 22 with the average being 5. In contrast, for Case 2, the set of programs for which the tool exceeds the allocated time before generating a test set satisfying the coverage criterion, the decisions ranged from 12 to 196 with the average being 38. This indicates that as the number of decisions increases, the performance deteriorates and the cost of using the tool may become prohibitive. This factor contributes to a scalability issue which results in longer test sequences, especially when generating tests for MC/DC.

It is important to note that during model-checking the reachability-based generation used by the toolbox is guided to achieve a desired coverage, and not to minimize or optimize the test. A generated test may not be the minimal way to satisfy the coverage criterion. However, a generated test might be able to satisfy more than one test obligation. From the point of view of limiting the number of tests generated, we note that our approach would perform better than other approaches including trap property generation [25, 14], which can lead to a large number of duplicate tests because these properties are derived by using the model-checker's ability to generate counterexamples.

Engineers from Bombardier Transportation AB indicated that their certification process involves achieving a minimum of 80% DC for all programs. For 78% of the programs in this study, the tool automatically generated tests achieving 100% DC, CC and MC/DC. For the other 22% of the programs, the results were less satisfactory. The data about the achieved coverage is shown in Table 5. As can be seen from this data, the tool generated tests with 82% DC on average. We conclude that we have provided evidence that this is a suitable tool for test generation tailored to FBD programs; it scaled well for most of the programs in this study and it is fully automated. There are, however, some drawbacks. Most importantly, for 22% of the programs, even though the tests generated for the coverage criteria achieved on average at least 65% coverage, we cannot determine whether the remaining test requirements are actually achievable, or if tests satisfying the requirements are longer that the search depth. This is an issue particularly for MC/DC where a fair number of test obligations were not satisfied.

From these experiments, it is clear that the toolbox can be sensitive to the number of decisions and as a consequence to the length of the tests required to achieve the desired coverage. In addition, the number of inputs considered during model checking is affecting the efficiency of the test generation technique. However, model checking does allow one to use a heuristic or meta-heuristic search technique [6, 26] to find the desired tests. We plan to investigate this approach in future work. In addition, the idea of combining symbolic execution or static analysis with model checking to achieve test generation has been proposed [20], and may allow more efficient model checking. Fraser et al. [13] noted that there is a lack of empirical evidence on how these model-checking techniques compare to each other in practice, making it hard to select an appropriate technique for a specific test purpose. We also plan to investigate how various approaches compare in future work.

8 Related Work

Model checkers have been used to produce test cases satisfying various criteria and for programs in a variety of formal languages [7, 17, 11]. Black et al. [4] discuss the problems encountered in using a model-checker for test case generation for full-predicate coverage. They present reasons why model-checking is not directly applicable for generating tests to satisfy logic coverage criteria. In our previous work [12], we overcome this issue by providing a way of generating test cases for logic criteria that are directly applicable to FBD programs. We found that model-checkers are an appropriate technique for automated test generation in terms of performance when used on real-world programs.

For data-flow programming languages such as FBD and Lustre, which describe the relationship between inputs and outputs instead of the control flow of the program, researchers proposed specific coverage metrics based on the structural aspects of the programs [19, 18, 21]. For Lustre, structural coverage metrics are based on the activation condition concept of the language that can be used when data travels from an input edge to an output edge. In addition, Whalen et al. [28] defined an alternative approach to measuring logic coverage for data flow programs called OMC/DC, a combination of MC/DC and an additional obligation to be satisfied such that faults will be observed through a variable monitored by the criteria.

9 Conclusion

In this paper we have shown how test case generation that aims to satisfy logic coverage on Function Block Diagrams can be solved as a model checking problem, by using model checking tools to automatically create traces that can be transformed into executable tests. We described a toolbox in which logic coverage criteria can be formalized and used by a model-checker to generate test cases. We carried out an extensive empirical study of the method by applying the toolbox to 157 real-world industrial programs developed at Bombardier Industries. The results showed that model checking is suitable for handling logic coverage for real-world FBD programs, and also revealed some potential limitations of the toolbox when used for test generation. The evaluation showed that the toolbox is efficient in terms of time required to generate tests that satisfy logic coverage and that it scales well for most of the programs. Our overall conclusion is that the model-checking approach provides a positive and useful addition to the testing process for FBD programs.

Acknowledgments

This research was supported by VINNOVA, the Swedish Governmental Agency for Innovation Systems within the ATAC project and The Knowledge Foundation (KKS) through the project 20130085 Testing of Critical System Characteristics (TOCSYC).

A Networks of Timed Automata

Let C be a finite set of reall-valued clocks and $B(C)$ the set of clock constraints, which are finite conjunctions of atomic guards of the form $x \bowtie n$, where $x \in C$, n is a natural number, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

A *timed automaton* (A) over actions $\mathscr{A}$, atomic propositions P and clocks C is a tuple $\langle N, l_0, E, I, V\rangle$ where N is a finite set of control locations, l_0 is the initial location, $E \subseteq N \times B(C) \times \mathscr{A} \times R^8 \times N$ is the set of edges. In the case of an edge $\langle l, g, a, r, l'\rangle \in E$, we write $l \xrightarrow{g,a,r} l'$ where the label g is a guard of the edge, r is the data- or clock reset assignments of the edge, and a is the action of the edge. $I : N \to B(C)$ is a function which for each control location assigns an invariant condition and $V : N \to 2^P$ is a function which for each control location provides a set of atomic propositions that are true in the location.

The semantics of A is defined in terms of a state transition system, where the state of A is defined as a pair (l, u), where l is a location and $u \in \mathbb{R}^C$ is a clock assignment in C. A state of A depends on its current location and on the current values of its clocks.

A state of an automaton A is defined as a pair (l, u), where l is a location, $u \in \mathbb{R}^C$ a clock assignment in C to a value in $\mathbb{R}_+$, with the initial state (l_0, u_0), where u_0 assigns all clocks in C to zero.

The semantics of A is given by the timed transition system $\langle S, s_0, E, V\rangle$, where S is the set of states of A, s_0 is the initial state (l_0, u_0), E is the transition relation defined as follows:

[8] R denotes the reset set i.e., assignments to manipulate clock- and data variables.

– $(l,u) \xrightarrow{d} (l, u \oplus d)$, where $u \oplus d$ is the result obtained by incrementing all clocks of the automata with the delay amount d such that for any $0 \leq d' \leq d$, the invariant of l holds.

– $(l,u) \xrightarrow{a} (l', u')$, corresponding to taking an edge $l \xrightarrow{g,a,r} l'$ for which the guard g is satisfied by u. The clock valuation u' of the target state is derived from reseting u according to updated r.

We denote by $T(A)$ all traces σ of A starting from the initial state (l_0, u_0) as a sequence of alternating transitions $\sigma = (l_0, u_0) \xrightarrow{a_1} (l_1, u_1) \xrightarrow{a_2} ... \xrightarrow{a_n} (l_n, u_n)$.

A network of timed automata $B_0 \parallel ... \parallel B_{n-1}$ is a parallel composition of n timed automata over C, $\mathscr{A}$ and synchronization functions (i.e., $a!$ is correlative with $a?$). We refer the reader to [1] for more information on the theory of timed automata. We consider a timed modal logic to specify properties. The logic may be seen as properties of A than can be expressed as logical formulae in the Timed Computational Tree Logic (TCTL) [2].

References

1. R. Alur. Timed Automata. In *Computer Aided Verification*, pages 688–688. Springer, 1999.
2. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in Dense Real-time. *Information and computation*, 104(1):2–34, 1993.
3. R. Alur and D. Dill. Automata for Modeling Real-time Systems. *Automata, languages and programming*, pages 322–335, 1990.
4. Paul Ammann, Paul E Black, and Wei Ding. Model Checkers in Software Testing. In *NIST-IR 6777, National Institute of Standards and Technology Report*, 2002.
5. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
6. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G Larsen, Paul Petterson, and Judi Romijn. Guiding and cost-optimality in uppaal. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 66–74, 2001.
7. Paul Black. Modeling and Marshaling: Making Tests from Model Checker Counterexamples. In *Proceedings of the 19th Digital Avionics Systems Conference*, volume 1, pages 1B3–1. IEEE, 2000.
8. CENELEC. 50128: Railway application–communications, signaling and processing systems–software for railway control and protection systems. *Standard Report*, 2001.
9. John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.
10. Henning Dierks. Plc-automata: a new class of implementable real-time automata. *Theoretical Computer Science*, 253(1):61–93, 2001.
11. Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. Model-based test suite generation for function block diagrams using the uppaal model checker. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 158–167. IEEE, 2013.
12. Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. Using logic coverage to improve testing function block diagrams. In *Testing Software and Systems*, pages 1–16. Springer, 2013.

13. Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with Model Checkers: a Survey. In *Journal on Software Testing, Verification and Reliability*, volume 19, pages 215–261. Wiley Online Library, 2009.
14. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Software EngineeringESEC/FSE99*, pages 146–162. Springer, 1999.
15. A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing Real-time Systems using UPPAAL. *Formal Methods and Testing*, pages 77–117, 2008.
16. Anders Hessel, Kim Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-Optimal Real-Time Test Case Generation Using UPPAAL. In *Lecture Notes in Computer Science, Formal Approaches to Software Testing*, pages 114–130. Springer Berlin Heidelberg, 2004.
17. Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic-Based Theory of Test Coverage and Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341. Springer, 2002.
18. E. Jee, S. Kim, S. Cha, and I. Lee. Automated Test Coverage Measurement for Reactor Protection System Software Implemented in Function Block Diagram. In *Journal on Computer Safety, Reliability, and Security*, pages 223–236. Springer, 2010.
19. E. Jee, J. Yoo, S. Cha, and D. Bae. A data flow-based structural testing technique for fbd programs. *Information and Software Technology*, 51(7):1131–1139, 2009.
20. Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.
21. A. Lakehal and I. Parissis. Lustructu: A Tool for the Automatic Coverage Assessment of Lustre Programs. In *International Symposium on Software Reliability Engineering*, pages 10–pp. IEEE, 2005.
22. K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.
23. M. Öhman, S. Johansson, and K.E. Årzén. Implementation Aspects of the PLC standard IEC 1131-3. In *Journal on Control Engineering Practice*, volume 6, pages 547–555. Elsevier, 1998.
24. S Rayadurgam and MPE Heimdahl. Generating MC/DC Adequate Test Sequences Through Model Checking. In *NASA Goddard Software Engineering Workshop Proceedings*, pages 91–96. IEEE, 2003.
25. Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based Test-case Generation using Model Checkers. In *Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings.*, pages 83–91. IEEE, 2001.
26. Kevin Seppi, Michael Jones, and Peter Lamborn. Guided model checking with a bayesian meta-heuristic. *Fundamenta Informaticae*, 70(1):111–126, 2006.
27. J. Thieme and H.M. Hanisch. Model-based Generation of Modular PLC Code using IEC61131 Function Blocks. In *Proceedings of the International Symposium on Industrial Electronics*, volume 1, pages 199–204. IEEE, 2002.
28. Michael Whalen, Gregory Gay, Dongjiang You, Mats P. E. Heimdahl, and Matt Staats. Observable Modified Condition/Decision Coverage. In *Proceedings of the International Conference on Software Engineering*, pages 102–111. IEEE, 2013.