2015

Model-Based Verification for SIMULINK Design

Victor Oke
Minnesota State University - Mankato

Model-Based Verification for SIMULINK Design

By

Victor Oke

Master's Thesis submitted in partial fulfillment of the requirements for the degree of Masters of Science in Engineering.

Department of Electrical and Computer Engineering and Technology

Minnesota State University, Mankato
Mankato, Minnesota

December 2015

Advisor: Dr. Nannan He

Model-Based Verification for SIMULINK Design

Victor Oke

This thesis has been examined and approved by the following members of the student's committee.

Dr. Nannan He, Advisor

Dr. Vincent Winstead

Dr. Muhammad Khaliq

Table of Contents

List of Figures

Acknowledgements

My sincere appreciation goes to my advisor, Dr. Nannan He, for her patience, understanding, encouragements and the confidence she has in me to make this thesis a reality. She really assists me in the research and always willing to listen to my concerns. I also want to express my gratitude to Dr. Vincent Winstead for his incessant support. His advice for me on this thesis and throughout the length of my masters program is indeed valuable. I can't do without thanking all my professors in Electrical Engineering department for their supports.

My wife Leslie is just so wholesome; her impacts on my success cannot be underestimated. The work wouldn't have been accomplished without the support of my family and friends. Thanks to everyone.

To the God almighty, the alpha and omega, who in His infinite mercy supports me and guides me throughout this program, all the glory and honor be onto your holy name.

Chapter 1

1.1 Introduction

Software development engineering deals with the series of approaches that are based on the software modeling as the primary form of expression. Sometimes, these models are explicitly designed including the executable actions and the supporting codes are well written by hand. Codes can also be generated from these models, ranging from system skeletons to complete, deployable products. Until present, *White-Box(structural)* or *Code-Based* testing have been studied by many research groups. However, systems have become more sophisticated and code lines have incomparably grown. Diving into details in program codes can be too cumbersome for testers because it requires lots of efforts, money and time, that is the reason why testers generally adopt *Black-Box(functional)* testing [19] rather than *White-Box* testing [22].

Model-based Testing (MBT) is the automatic generation of software test procedure, using the models of system requirements and behavior. Automatic support is required for good functionality of MBT and this is probably the most popular software testing [24] used for verification and validation techniques for modeling software under test (SUT) [23]. Model-based Testing can either be black-box or grey-box considering the level of abstraction of the model from which the behavior of SUT is observed. Black-box MBT consists of high level of abstraction representing the SUT behavior, while the grey-box MBT describes the model with details of the design information. White-box testing is not usually considered MBT [29].

Also with the increasing demand in software products, customers expect more reliable, efficient and a quality software product that contains advanced features and functionality. The competition between many companies forces the manufacturer to deliver the product with certain

prerequisites within a short period of time. This necessitates a short period of testing time. There comes the need for test automation. Automation of testing not only reduces the effort and time but also the cost incurred as testing needs to be done regressively when meeting tight project schedules. The focus of this thesis is on the evaluation of software testing methods and the requirements related to these testing methods. The requirements relating to the methods of testing are in the domain of embedded real-time systems.

1.2 Scope

The scope of this thesis covers the design level, testing level and the verification level. Since all these levels of design result in comprehensive and confidence design, all hands are put together to go through all the three levels of design, although more emphasis are laid on design and the verification level and these are also in the user interface level of application. The design is accomplished by the use of Simulink module and the verification is being done by bounded model checker for C (CBMC), Gene-Auto [30] is used as the testing tool. Some suggestions for future design are also included.

1.2 Method

Solid background knowledge was developed after a thorough literature study on model based design and testing. The design in this thesis stems from the challenges associated with an industrial application model applied to automatic water tank control system. From the requirements which must be met to get this water tank in place, knowledge of ladder logic was applied. The ladder logic design was accomplished and the testing was executed but the verification was almost impossible except for the timeliness properties which were done using Uppaal [17] and this led to using Simulink-based design modeling to achieve proper verification.

The water tank model will be designed in Matlab/Simulink and the properties will be verified with Bounded Model Checker for C (CBMC). CBMC only understands C code and therefore cannot run the Simulink model without compilation. Therefore Gene-auto will be introduced. Gene-Auto automatically converts the Simulink model to its equivalent C code [13] and this also serves as a test tool because during the conversion, any mismatch in the design will result in conversion failure and the C code will not be generated until this error is fixed. After a successful generation of C code, CBMC [26] will be used to run the code for verification. Assertion follows to assert the properties of the model.

In addition, automatic generation of miter model will be developed, this allows automatic injection of mutant [15] in the miter model and the behavior (output) of individual mutated miter model will be compared using the same inputs (Test cases) and then verified by using the above procedures.

1.3 Related Work

Our work on Simulink model analysis is related to available methods in the literature. For instance, code generated by Simulink was automatically validated using a decision procedure, (Strichman and Ryabtsev [13]). This was done against some verification conditions which were extracted from the model. Most authors touch a small fragment of Simulink model and only discuss the approximate behavior of the model [5]. In contrast, our work extends these existing results to a deeper level. Specifically, the precise behavior of the model was explicitly analyzed.

Our work is also related to previous work on the generation of test vectors with the use of software model checkers [25]. CBMC or other similar techniques have been reportedly used by

some papers for generating high –coverage test suites and our work is closely related to these implementations. Mutant injection in a model is also part of our work in this thesis and this has equally been touched in various dimensions by some other authors [15]. Some papers described single mutations [1] while combination of faults has been considered by others [4] in mutant models. Bounded model checking method for estimating coverage was described by GroBe et al. [7] which explained implementing the flip mutation at a given cycle and the verification is done by model checker. Impact of equivalent mutations was described by Schuler et al. [16] and the means of detecting such mutations was also discussed.

1.4 Structure

The thesis work was first introduced in chapter 1 where we discussed the scope of the thesis, the method used to execute our work and the related work. We also went ahead to discuss Model-based design, Model-based testing and the testing procedures. Ladder logic, Simulink and the relationship between the two will be discussed in chapter 2. Chapter 3 describes the thesis methodology where the whole work procedures are executed and the system properties are verified. Some observations are drawn and suggestions for future designs are also discussed. Then, the last part provides concluding remarks to all the work done.

Chapter 2

2.1 Model-Based Design

Model-Based Design is a visual and math-based method of describing complex control design problems. Its usefulness is not limited to industrial applications but also motion control, aerospace and automotive applications. This happens to be the efficient methodology applied in embedded system design and it consists of four ordered steps of the development process: (1) plant modeling, (2) analyzing and synthesizing a controller for the plant, (3) simultaneous simulation of the plant and that of the controller, (4) deploying the controller [27]. Model-Based Design is a more cost effective and time-saving approach in the development of dynamic systems, not limited to control systems but also signal processing and communication systems.

Traditional design methodology consists of complex structures and extensive software code development, but the Model based design touchstone is completely different. Designers use continuous-time and discrete-time building blocks to formulate a model with advanced functional characteristics. The formulated models with the corresponding simulation support tools can result in rapid prototyping and enhance software testing, software/hardware validation and the verification process. The Model-Based Design development process starts from requirements analysis to design and implementation, followed by testing and verification. Some of the advantages of Model-based Design that makes it a more efficient approach are that common design environments are used across project teams, designs are directly linked to the requirements, early stage error detection and correction, software codes and design documentation are automatically generated, algorithms are refined through multi-domain simulation. In addition, the test suites are reusable. More information regarding Model-based design for embedded system is discussed in [28].

2.2 Model-Based Testing (MBT)

Model Based Testing is a software testing method in which test cases are derived entirely or partially from a behavioral model that describes the System Under Test, (SUT). Most model-based testing inherits the complexity of the domain, or more particularly of the related domain models, the basic model is abstract and tries to describe the system in whole or in less detailed mode. The generated test case from this model is as abstract as the original model and this is called the Abstract Test Suites (ATS). Since this is abstract, it is not potentially executable but can provide an Executable Test Suite (ETS) that perfectly runs the SUT. There is no particular best method to create test cases because many methods have been developed to generate this from the models and fundamentally software testing is often heuristic based and experimental. Most of the time the package is created, namely Test Requirements, which includes the test stop conditions and information with regard to the SUT part which should be tested.

Test requirements are usually a result of merging the whole test configuration related to the design decision.

2.2.1 Importance of MBT

The main benefit of MBT is to generate a wide range of test cases in short span of time. Even though modeling takes a considerable amount of time, it will always be less than deriving the test cases manually. MBT allows us to test every module of the system at each stage in order to detect which part does not satisfy the design specification before the whole design is coupled or put into operation. This approach saves time. It allows one to figure out design errors, inconsistencies or uncontrollable failures in design. Again, Model Based Testing is related to

how well the automation can be implemented. Therefore, the models that are formal, well-defined functional interpretations, or machine readable models can in principle originate test cases automatically. These models are commonly translated to State Transition Systems (STS) or Finite State Automata (FSA). These STS or FSA show the feasible configuration of SUT. Thus, to generate a test case STS/FSA is ought to find an executable path. An arbitrary feasible execution path works as a test case. This technique is only possible if the model is Deterministic Finite Automaton, FDA, or if it is reducible to a FDA. Based on the designed model, test cases are usually generated and prioritization of the test cases will be needed to structure the testing process and reduce test effort. Standard test generation criterion such as boundary value analysis, equivalence class partitioning and cycle coverage are generally used. But the underlying factor is the selection of any criteria that covers all the requirements.

2.3 Quality Control Techniques in MBT

Approaches to quality control techniques in Model-based design are *Validation* and *Verification*.

- Validation is just the measure of correctness or completeness of how a design specification or requirement is being implemented. This is actually done to detect gross error in the system. This validation is incomplete of course, but this is not very important in this context as compared to the usual refinement-to-code context. With Model-Based Testing, if some errors remain in the model, they are very likely to be detected when the generated tests are run against the system under test.
- Verification is the way of scrutinizing the consistency of a system with respect to specified design requirements. The two steps in verification are *Testing* and *Formal Verification/Model Checking.*

2.4 Testing

This is used to validate control system programs and subsequently detect errors. It can still be understood as the process of systematically evaluating a system by observing its execution. The main advantage of testing is its scalability, which means that it can handle millions of line of code but this does not guarantee correctness because there is possibility of a system being successfully tested and still contain errors. For the purpose of this thesis, Gene-auto is used as the testing tool. This converts the designed SIMULINK model to its equivalent C-code.

Meaning of Testing

Testing can be interpreted in several ways depending one's point of view however general understanding of testing is described below, some of which is contained in Beizer's testing levels [20];

- Testing can be understood as tester's confidence booster. At least if all detected failures have been removed from a system, a tester will have confidence on the correctness of the system. It is understood that testing does not guarantee the absence of faults.
- Testing can also be described as getting the variation between the actual and expected behaviors of the system under test which brings about possibilities of detecting functional failures.
- Testing is actually detecting failures and not the cause of the failures. This clarifies the difference between testing and debugging because only debugging finds cause of a particular failure. It does not prove absence of faults.

2.5 Definition of Terms Related to Testing

There are several notations used in testing procedures and few of these are described below. They are commonly used throughout the thesis.

- **Test Case**: This is a set of input stimuli to be introduced into a system and the expected behavior of the system under which a tester will determine whether the system under test satisfies requirements or works correctly. Tester can also find problems in the requirements or design during this process. It is good practice for testers to test one thing at a time in order to ensure that test cases are not complicated or overlapped. In test cases, both the positive scenarios and the negative scenario must be covered, and these must be accurate, traceable, repeatable and be reusable if necessary.
- **Abstract Test Case**: This gives an idea of the test case structure and the information about satisfied coverage criteria, it is actually made up of abstract information about the set of input and output where the concrete information like parameter values or function names are missing. This is usually the first step in test case generation and it cannot be directly used on the actual system under test because of its high abstraction level and lack of concrete information about the SUT and its environment.
- **Concrete Test Case**: This comprises of the present of abstract test case together with the missing concrete information which gives it sufficient details to be actively executed and correctly communicate with the system under test (SUT).
- **Test Suite:** This is set of detailed test cases that show some specific set of behaviors and contains some information about the configuration to be used during testing. The executable test suite is usually interfaced with the SUT through the test harness.
- **Test Harness:** This brings about automation of test data under varying conditions and monitoring its outputs with test execution engine and test script repository. It is also

called Automated Test Framework. Test harness plays very important roles in testing procedure such as automating the test process, executing test suites of test cases, generating corresponding test results and ensuring that subsequent test runs are exact duplicates of the previous ones. Description of the elements of a typical test case is presented in the table 1.

T. C. ID	Test case ID
T. C. Summary	The test case summary or objective
T. S. ID	Test suite ID for this test case
RRID	This is the related requirement ID to which the test case can be traced
Prerequisites	The combination of preconditions that must be met before test execution
Test procedure	All the steps taken to execute the test
Test Data	All the parameters needed to conduct the test
Expected Result	The expected test output
Actual Result	The actual test output generated after test execution
Test Status	The conclusion to the test output either successful or failed
Remarks	Any suggestion or comments related to the test conducts
Created By	Name of the author that derived the test case
D.O.C	The actual date the test case was created
Execution Date	The actual date the test was executed
Executed By	The name of the actual person conducted the test
Test Environment	The kind of software or hardware used to carry out the test procedures

Table 1: Elements of Test Case

- **Test Oracle:** This is used to determine the status of test whether it failed or passed. It handles this by comparing what it knows to be the actual behaviors of the system, to the behaviors of the SUT for a specific test-case input at a particular time. No test is able to detect a failure without this.
- **Debugging:** This is the process by which a fault causing a particular failure in a given time is being located.

  Whenever a fault is detected in a system, it is understandable that not all inputs fed into the system caused the incorrect output which is known as failure and relating a particular failure with a corresponding fault will often be very difficult. And so, these concepts are better analyzed by fault/failure model which states that for a failure to be observed, the below three conditions must be observed;

  - **Reachability:** The location of the fault in the program must be reachable.
  - **Infection:** It must be confirmed that the state of the program in that particular location where the fault occur must be incorrect.
  - **Propagation:** The infected state of the program must propagate to cause incorrect output.

2.6 Test Techniques

Test techniques can be explained from the knowledge and the observability of the system under test. This can be conducted under black-box, white-box and gray-box testing.

Black-Box Testing: This refers to testing a system where the testers have no specific knowledge of the internal matters or the internal workings of the system. Testers only have knowledge of the possible input and expected output values but do not know how the program actually arrives at those output values. This is represented in figure 1. The source code and the architecture

knowledge of the system are not known, therefore the in-output functionality of the system are only allowed to be tested. Because of this, black-box testing is considered to be functional testing and is known as opaque box testing or closed box testing. The advantage of this is that tester can be non-technical since there is no need for the tester to have detailed functional knowledge of the system and that the test will be done from an end user's point of view, because the system must be accepted by the end user. The disadvantage of this is that since all possible inputs in a limited testing time will be difficult to identify, then writing test cases may be slow or difficult.

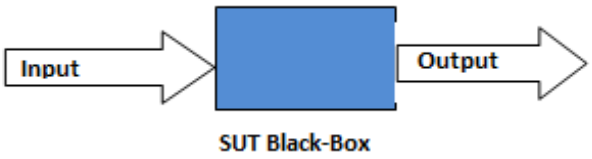


Figure 1: **SUT Black-Box**

White-Box Testing: This refers to testing a system with full knowledge of its internal matters and this can still be used to create tests because there is access to all source code and the architecture documents. With the access to this information, white-box testing is not restricted to the detection of failures, but bugs and vulnerabilities are also easily detected compared to the black box testing and we can be confident of getting more complete testing coverage since we precisely know what to test. This is also known as clear box testing. The advantage of this is the higher quality testing which is more thorough with the possibility of covering most paths because of the wide knowledge and information about the internal matters of the system. The main disadvantage of white-box testing is that highly skilled resources with thorough programming knowledge and implementation is required since text can be very complex and high effort is actually needed to scrutinize all aspect of the program.

Gray-Box Testing: Gray-box testing is actually the combination of both black-box and the white-box testing. It is testing a system with partial knowledge of the internal matters of the system. This knowledge is usually constrained to detailed design documents and architecture diagrams. This testing technique is used to design tests at white-box level and execute them at black-box level. Gray-box testing is commonly used for commercial model-based testing where the tester have the rich knowledge of the internal matters of the system at the design level but all these knowledge are not known or not accessible by any tester at the execution level.

2.7 Testing Levels and Testing Process

System development management can be explained from series of models but development knowledge from V-Model is found to be more pronounced because it's comprised of development stages at the top and the testing stages at the bottom. Figure 2 shows the importance of early execution test in system design because it is clearly understood that late execution test will definitely have an impact on the early development stages. The development stages range from the system requirements and specification, doing the requirement analysis, system being designed and arranged in modules, and implementation. The testing stages comprised of module testing for classes, integration testing which deals with components consisting of classes, system testing which integrate all components and the acceptance testing of the customer.

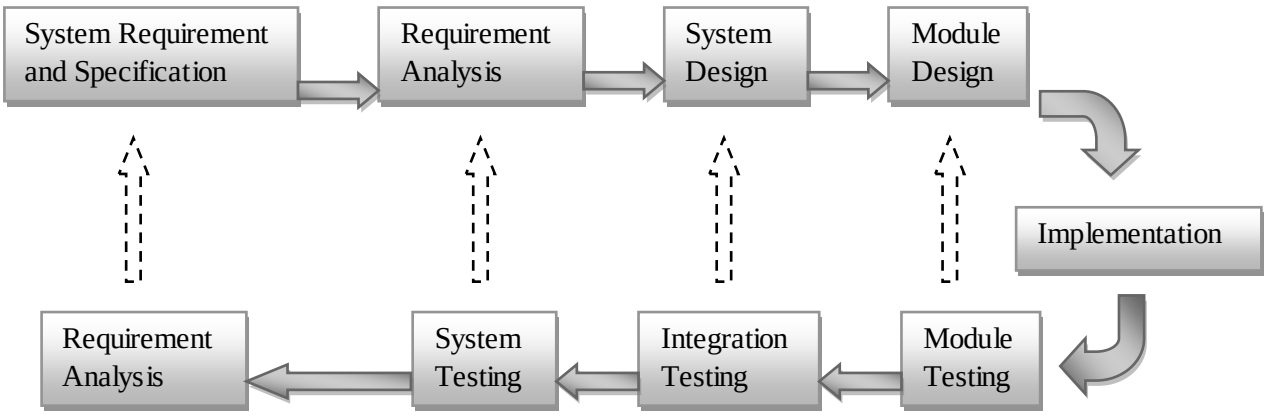


Figure 2: Model-Based Testing Level

Utilizing the style of user interface (UI) design could be a very good approach where the design application flow is explicitly described in the UI level. In this the models act as transformer between the source, which can be a UI specification, and the target, which can be a test automation script. The beauty of this is that it forms an easy process in the sense that the tester only has to provide the test automation parameters into the model, execute the test and finally analyze the results. The whole process is made possible with the availability of tools which can convert user interface specification into an application model, and create test cases from it. Figure 3 below described the Model-based testing process.

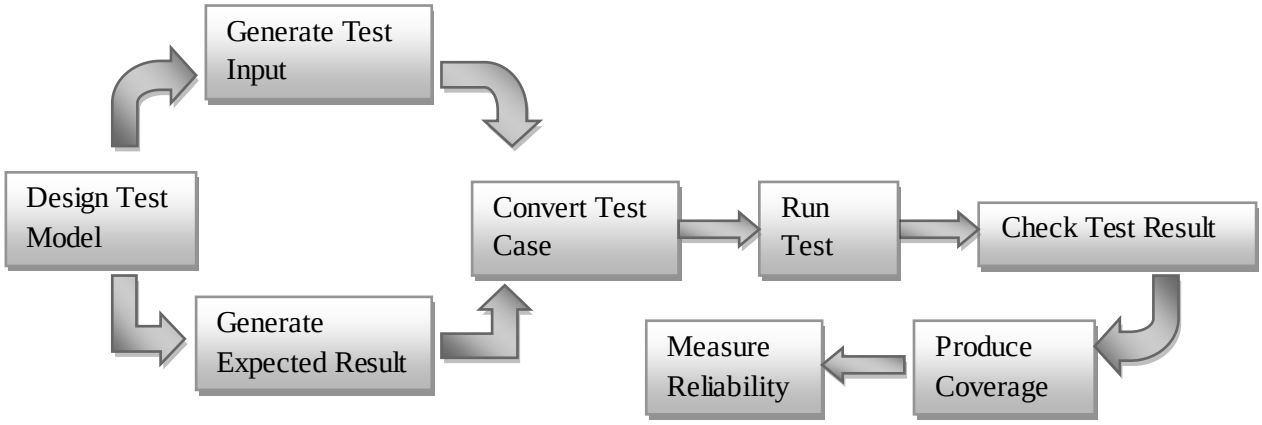


Figure 3: Model-Based Testing Process

2.8 Formal Verification/Model Checking

This actually ensures correctness and consistency, but not scalability. Basically, Model Checking was created as a method of assuring whether an attribute of a specification is acceptable in the model. The model of the SUT and the specific requirement to be examined in the model checker will be developed, and if this attribute is acceptable in the model, in as much as the attribute is under test to get proved, the model checker identifies instances and contradictions. An instance can be a path in the execution of the model where the attribute is satisfied, while contradiction is a path where the attribute failed. This particular path can be reused as a test case several times. The earlier verification is performed in a design the sooner the errors are detected and rectified. It is very important to verify the integrity of the designed model before deploying it on a target embedded controller for build and integration because of the cost and scarcity of physical prototypes. Verification of design integrity is usually achieved through simulation and coverage analysis. Numerical overflow is one of the indication of poor design integrity and this condition can easily be curbed with simulation by stress testing the minimum and maximum numerical values of the model. Another poor design integrity indicator is the unreachable logic which

means that part of the design is missing an important aspect during specification, implementation and the test creation. This cannot be easily detected through mere simulation but structural coverage can be best applied for the detection. To determine if the test passed or failed, model assertion is employed. This ensured that signal does not exceed its boundary during simulation or testing and it does this by stopping the execution when it's about to happen.

Two common verification tools;

- **CBMC**: Software bounded model checking for C. It automatically proves the correctness of C codes in bound but this does not verify the timeliness requirement. The original brain behind the Bounded model checker (BMC) was to develop an environment where the modeling system is associated with program traces that violate some specific requirements and clarifies satisfy resulting formulae. Application of CBMC is becoming very popular, this is not limited to automatic test generation for verification of circuits and microprocessor designs [14] but also many papers have actually applied BMC to formally verify finite systems and develop software verification [11]. Bounded Model Checker for C programs (CBMC) was used in this paper as automatic test generator, performing assertion and system verification. The key idea of CBMC is to work with low-level ANSI-C programs[12], discover array bounds, pointer constructs correctness, and user-provided assertions, also checks all other system safety properties [9]. The importance of CBMC in increasing the productivity of the entire software development process cannot be ignored, this enhancement has been achieved by reducing the cost of the testing phase. Since random testing does not provide enough confidence for proving the correctness of a compiled system because it solely relies on probability and finding semantically small faults with it is not guaranteed. Making up for this inefficiency

involves providing set of tests that covers 100% of the code. The use of CBMC was effectively experimented in this paper where we were able to verify the modules of an industrial water tank control system by generating set of test for individual function performed by each module and this provides 100% coverage of the code. If any part of the program violates the requirement, CBMC will return an error-trace which is an assignment to input variables. This error trace return is important in automatic test generation because a property has to be violated and this is achieved by inserting an assertion code that must be violated by the program using user-provided assertion. CBMC will generate an error trace return assigning violating value to the input variable.

- **UPPAAL**: This is a widely used model checker for real-time systems and it is modeled in timed automaton. A timed automaton is a non-deterministic finite state machine which uses a clock to express its timing properties. A clock can be set to zero and gradually increases its value linearly with time. At any instance, the value of a clock is equal to time elapsed since the last time it was reset. Timed automata comprised of control state, variables and the clocks. Here, transition is only possible when the associated timed constrain is satisfied and its guard expression evaluates to true in the system state. Execution of timed automata are infinite sequences of system states that fulfill the invariants which may be either the passing of time or running of transitions. UPPAAL explicitly verified the system timeliness properties by using the UPPAAL Timed Automata (UPTA) which is an extended version of timed automata, to specify a system as a network of timed automata consisting locations and transitions. Transitions between these locations describe the behavior of the system.

2.9 Ladder Logic

Concept of Programmable Logic Controller (PLC) originated from the knowledge of relay logic control system and Ladder logic happen to be the conventional programming language for the PLC because of the resemblance between the ladder logic programming diagram and that of relay logic control system. The brief introductions about Ladder logic programming can be better described by converting a simple switch program to a relay logic and finally to PLC ladder logic. Industrial Load control electrical circuit diagram can also be used to describe Ladder logic programs since this is basically the open or closed switch concept. It is understood by engineers or technicians that opened switch disconnects (break contact) the load from incoming current while the closed switch connects (make contact) the load to the incoming current, this is also known as an ON/OFF switch. The switch can be manually or automatically controlled depending on the application available. Figure 4 below represents a simple ladder logic diagram.

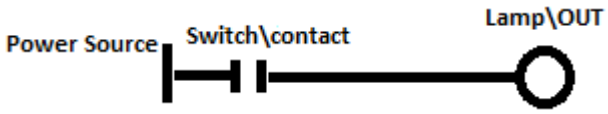


Figure 4: Simple ladder logic Diagram

The switch has to be connected before the power can flow to the lamp and light up the lamp. The horizontal line represents the flow of current. The switch used above is called a normally-open contact and there is also another one called the normally-close contact, this is shown in the figure 5. This can be extended to PLC application by connecting the switch to the PLC input, the lamp to the PLC output and the same program is run. Combination of contacts in various dimensions makes up the ladder logic program and most of these combinations are briefly described. This is called ladder logic because of its ladder nature and the combination of the contacts function as

logical operators. Normally open contact will be closed (connected) when it is activated and the normally close contact will be opened (disconnected) when activated.

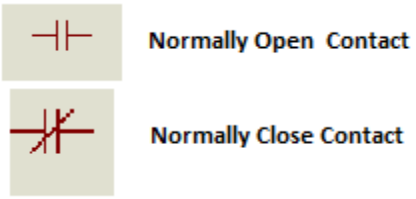


Figure 5: Normally close and Normally Open Contact

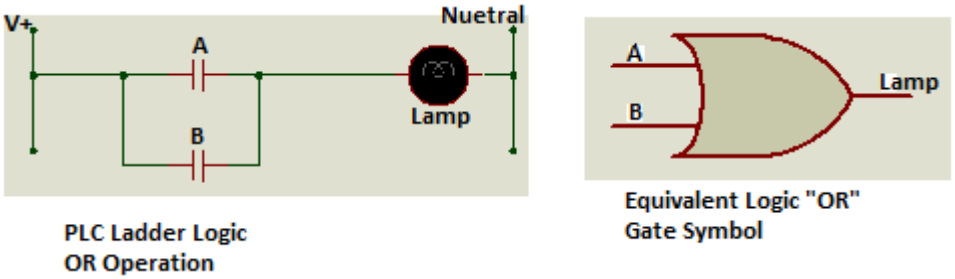


Figure 6: OR Symbol Equivalency

A	B	Lamp
OFF	OFF	OFF
OFF	ON	ON
ON	OFF	ON
ON	ON	ON

Table 2: Representing logic "OR"

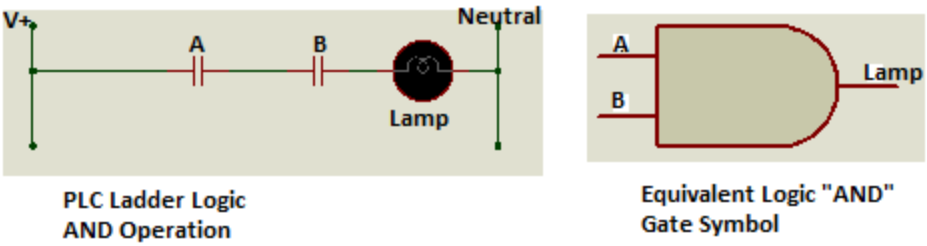


Figure 7: AND Symbol Equivalency

A	B	Lamp
OFF	OFF	OFF
OFF	ON	OFF
ON	OFF	OFF
ON	ON	ON

Table 3: Representing logic "AND"

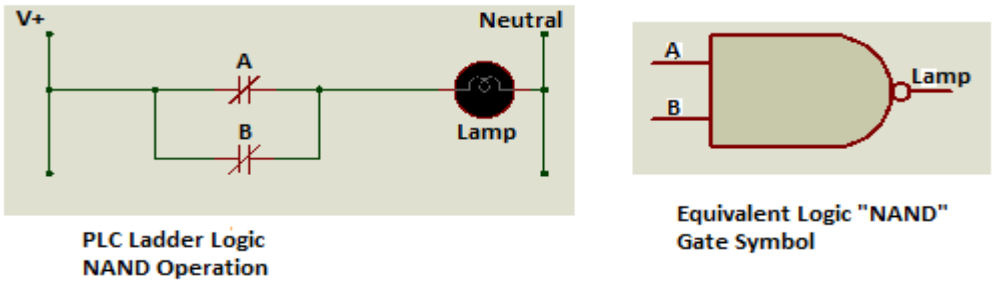


Figure 8: NAND Symbol Equivalency

A	B	Lamp
OFF	OFF	ON
OFF	ON	ON
ON	OFF	ON
ON	ON	OFF

Table 4: Representing logic "NAND"

In the figure 6 above, the lamp will turn ON when either of the contact A or B is activated, or when both are activated simultaneously. Table 2 above describes the OR operation. Figure 7 describes AND gate operation, the lamp turns ON only when both contact A and B are activated. If contact A is activated and contact B is not or either way, the lamp will not be turned ON. Table 3 also describes this scenario. In figure 8, both contact A and contact B need to be activated for the lamp to be OFF, but except that, any combination of contact A and B will definitely turn ON the lamp as described in table 4 above.

Knowledge of the PLC ladder logic programming and the possibility of relating it to logic gate operator equivalency helps a lot in this thesis to make the combination of the requirement analysis for modeling the industrial water tank control system, and these were transformed and explicitly modeled using Simulink.

2.10 Simulink

Simulink is an interactive graphical environment where model-based design for embedded system or dynamic system is been created and simulated [8]. This provides an easier and faster way to develop a model compared to text-based programming language (like C programming) because of the explicit details provided by graphical models and enhanced intellectual controls. A model represents a system which consists of collection of blocks. All these set of block

libraries in Simulink are used for modeling (combine, modify and generate output) and simulation (display signals) or to test some time-varying systems, they actually help in clarifying requirements analysis, validation and verification. Application of Simulink is not limited to the area of controls and communications but also very useful in image processing, video processing and signal processing. Simulink also use lines to transmit signals from block to block, this is usually done by transmitting the signal from the output terminal of one block to the input terminal of another. Figure 9 shows an example of basic Simulink model. Some examples of common Simulink blocks are listed below;

- **Sources:** These are used for signal generation and comprised of Signal generator, Step function, Random number, Ramp and Constant.
- **Linear and Connections:** This comprised of continuous-time system element, Connections and Linear blocks such as math Operations blocks (Add, Product, Gain, Sum), summing junctions, Signal routing (Mux and Demux) etc.
- **Nonlinear Operators:** These include Saturation, Transport Delay, Arbitrary functions etc.
- **Discrete:** This consists of discrete-time system elements such as State-Space, Transfer function, Integrator etc.
- **Sinks:** These are used to display or output signals. Examples are Scope, XY Graph etc.

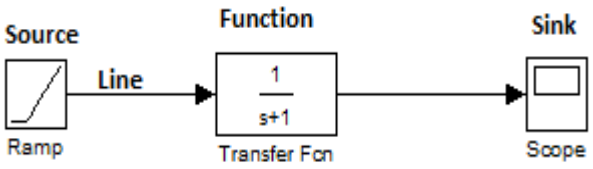


Figure 9: Simple Simulink model

Chapter 3

3.1 Methodology

This thesis work is done by carrying out research and executing the design on industrial water tank control system shown in figure11.This section describes the details of all the procedures taken to implement the model.

3.1.1 Overview

The work started from gathering the requirements to be taken into consideration for the design, followed by modeling of the industrial water tank control system with respect to these requirements using PLC ladder logic Programming. The content of the ladder logic is converted to its equivalent simulink model, then all the verifications and assertions were carried out. Figure describes the block diagram for the procedures.

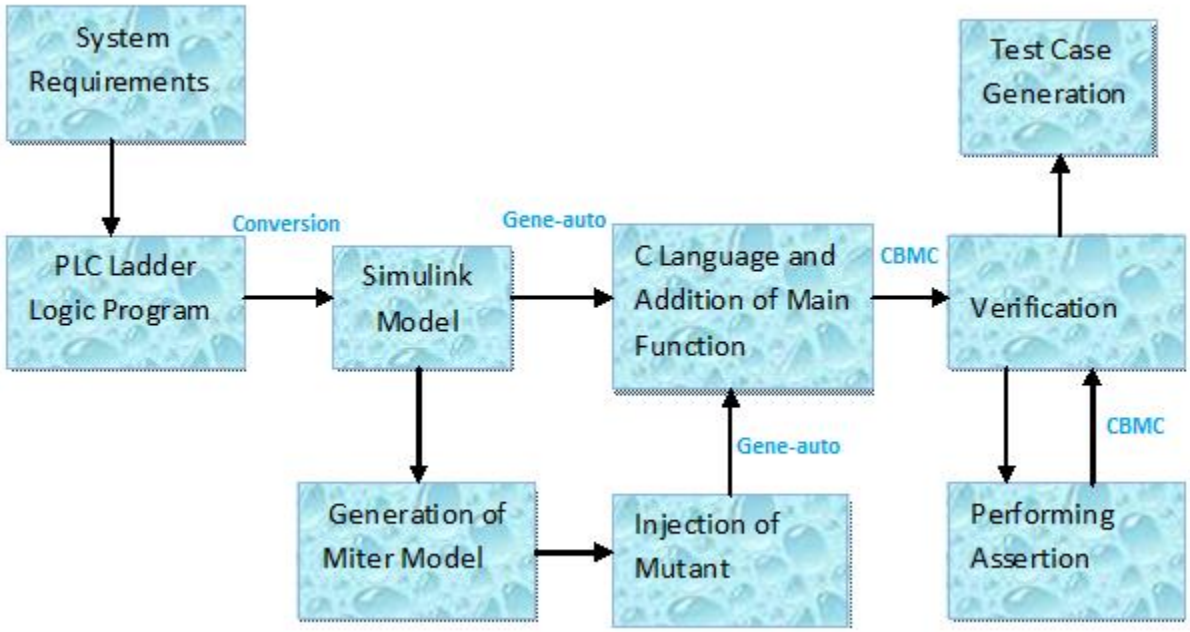


Figure 10: Block Diagram for Modeling Procedure

3.1.2 Case Study

To ensure effective operation of the tank, some safety features were established, this is shown in table 4 below. The control unit consists of three flow lines (Flow line1, Flow line2 and backflow line) with two pumps, Up flow valve, Down flow valve and Backflow valve, there is also a Water tank and the Output valve. The requirement is that the two flow lines must not operate at the same time and this is accomplished by ensuring that the pumps are not working at the same time, should this situation occur, this indicates a Pump failure and thereby error in the control system. Any Pump failure must cause the associated valve to stop working. Another safety is that the valves must be actuated for 10 seconds before the pump should be opened, which means that no pump should work if the valves are not activated for this specified time. Should the Pumps be Idle, both Output valve and the Backflow valve must stop operating. The table 2 below presents the lists of the inputs and the outputs for the control system interactions.

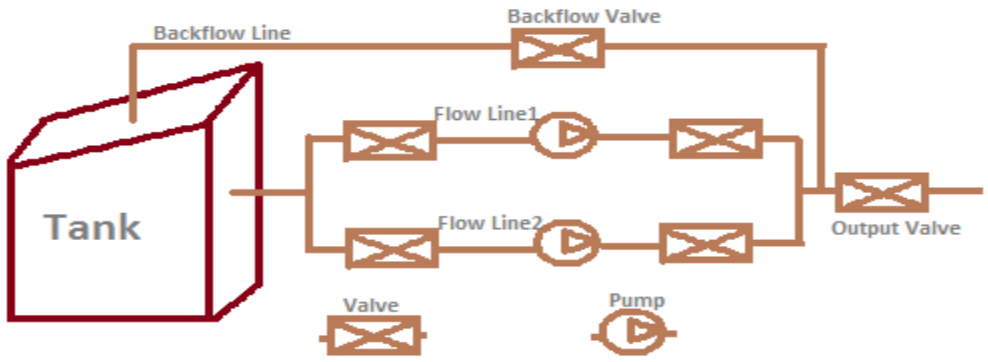


Figure 11: Water Tank Control System

Li_CH_REQ	Request for changing of the line
H_FLOW	High flow rate water distribution request
L_FLOW	Low flow rate water distribution request
Lx_FAIL	Failure from any of the Pumps
SP_FAIL	Indication that there is no distribution
Lx_PUMP	Starting up any of the Pumps
Lx_UP	Activate the Upstream valve of any of the Pumps
Lx_DOWN	Activate the Downstream valve of any of the Pumps
BFW_VALVE	Backflow valve opened
OUT_VALVE	Output valve

Table 5: Input and Output flow

3.1.3 Implementation Details

All the requirements for this system were strictly examined, demonstrated with ladder logic and transformed into a Simulink model. Since our priority for this design is safety, then the control system has to be modeled with the programming environment where adequate testing can be performed, and explicit validation and verification can be carried out. For this kind of design, modeling using Simulink appears to be a good choice because Simulink programs can eventually be converted to a different form of programming languages which can be understood by some testing and verification tools. This flexibility feature has made Simulink probably the most popular tool for model based designs.

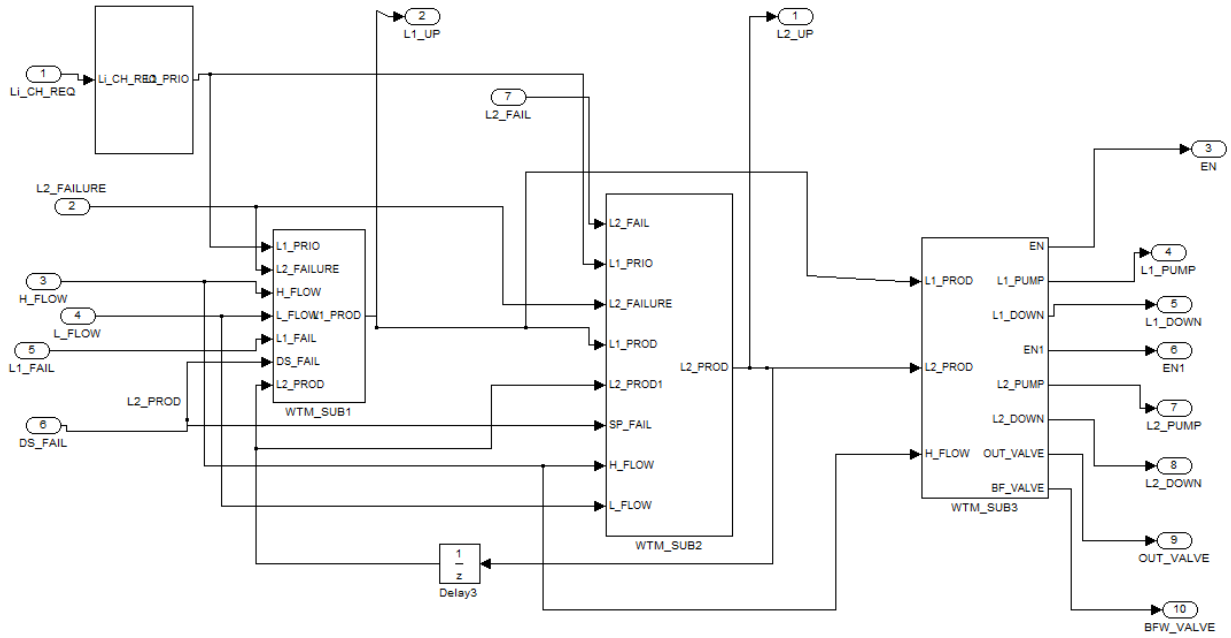


Figure 12: Water Tank Control System Simulink Model

Figure 12 above consists of four subsystems which are made of a combination of logical operators. These are arranged and connected in such a way as to perform the functions specified by the control tank and meet up with all the safety conditions included in the requirement analysis. This starts with a step by step analysis of simple PLC ladder logic programming and interpreting it into combination and connection of simple logical operator blocks in Simulink to check the specific requirement at that step. Figure 13 gives details of this interpretation for the first subsystem.

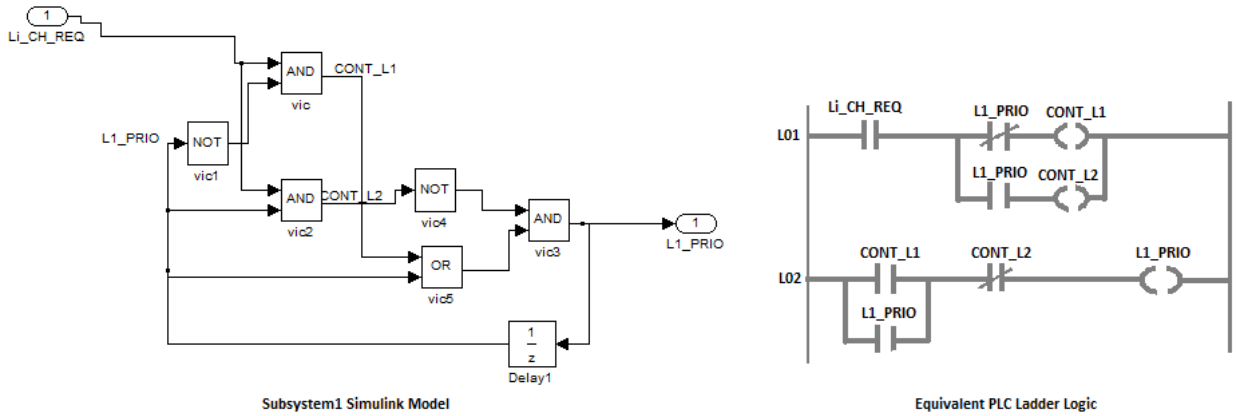


Figure 13: Simulink Model - PLC ladder Logic Equivalency

The description of this model is that **Li_CH_REQ** is "AND" with the "NOT" of **L1_PRIO** to output **CONT_L1**. Again, **Li_CH_REQ** is "AND" with **L1_PRIO** to output **CONT_L2**. Then, **CONT_L1** is "OR" with **L1_PRIO** and the output is "AND" with the "NOT" of **CONT_L2** to output **CONT_L2**. The **Delay1** is a time delay which was used to denote that **L1_PRIO** is a feedback. This same block combinational procedure was used to model the whole system according to the specified requirement and it was reduced to four different subsystems i.e. the first subsystem, WTM_SUB1, WTM_SUB2 and WTM_SUB3.

3.1.4 Gene-auto and CBMC

After modeling each stage of the system, a test is been carried out to ensure that there is no wrong block combinations or connection error in the design. This early stage testing is one of the advantages of Simulink modeling because it saves design time, reduce cost of design and boost confidence of the designer. It is actually understood that it will be very discouraging if a model cannot be tested for error at the early stage and found out to contain some errors when tested at

the end of the whole design. This means that the designer will either have to start all over again or undergo series of troubleshooting steps before any found error might be fixed or rectified.

The Simulink model shown in figure 11 is then run through Gene-auto (testing tool) and the result showing successful simulation is shown in figure 14.

Figure 14: Gene-auto Simulation Output

The purpose for this step is to convert the Simulink model to its equivalent C program and to ensure that the design is modeled correctly because if there is lapse in the design, the simulation will return error result. This step is called the testing level and the fact that there is a successful simulation does not guarantee a perfect design nor does it indicate that all design requirements are met. Since the model is been successfully converted to C language then a verification procedure has been carried out with the use of model checking tools called CBMC. But before CBMC can understand the C program generated by the Gene-auto, a “main” function needs to be added to the program because without this the CBMC simulation will return conversion error. The generated Gene-auto C program is shown at the appendix A.3 and the main function added to the C program is shown below. The window showing the successful verification with the CBMC is presented in figure 15.

```
void main( ) {
T_WATER_TANK_MODELLING3_io *t_io;
T_WATER_TANK_MODELLING3_state *t_state;
WATER_TANK_MODELLING3_compute(t_io, t_state);
}
```

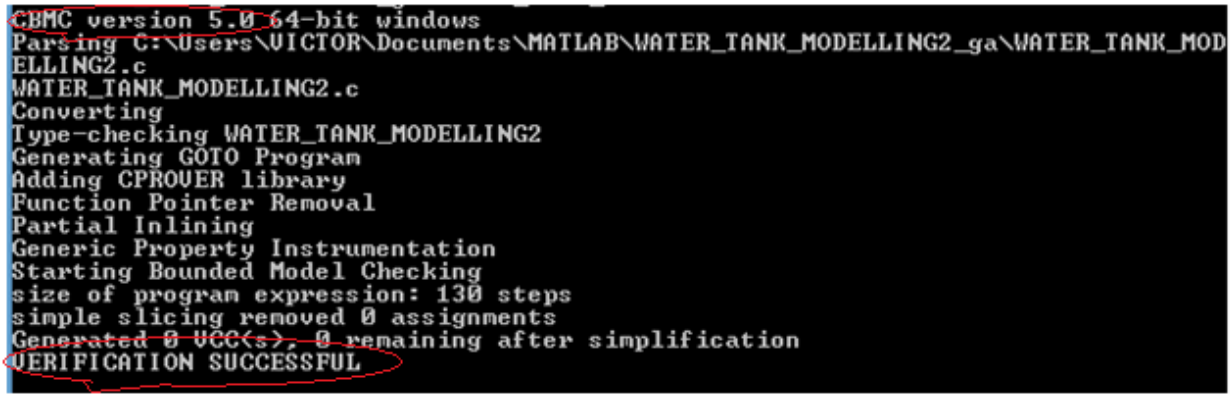


Figure 15: CBMC Simulation Output

Successful verification indicates that the system under test (SUT) is correctly modeled and that is free of error. The next step is checking the properties of the system whether it meets up with the modeling requirements and this can be done by assertion. Assertion is a statement about the intended behavior or a requirement of the system and Assertion-Based Verification ensured that the system/design obey the temporary assertions. This is combining the test cases in such a way that satisfy the intended behavior of the system. The corresponding assertion statement for this system is presented below;

```
void main( ) {
t_WATER_TANK_MODELING3_io*t_io;
```

```
t_WATER_TANK_MODELING3_state*t_state;

t_WATER_TANK_MODELING3_compute(t_io, t_state);

assert(t_io → L2_PUMP ==1 || t_io→ L2_PUMP ==0); //verify Boolean input

assert(t_io → L1_PUMP ==1 || t_io→ L1_PUMP ==0); //verify Boolean input

assert((t_io → L1_PUMP ==1 && t_io → L2_PUMP ==0) || (t_io → L1_PUMP ==1 &&

t_io→ L2_PUMP ==0) || (t_io → L1_PUMP ==0 && t_io → L2_PUMP ==0)); //verify that
the two pumps don't work at the same time.

Assert((t_io →L1_PUMP ==0 && t_io → L2_PUMP ==0) || (t_io → BF_VALVE ==0 &&

t_io→ OUT_VALVE ==0)); //verify that if both pumps are not working then the backflow
valve and the output valve must stop working.

assert((t_io → L2_FAILURE ==1) && (t_io → L2_FAIL ==1) && (t_io → L2_PUMP ==0);
//verify that any failure in line 2 will cause L2_PUMP to stop. }
```

All the above assertion statements return successful verification after simulation. Note that any assertion that does not obey the system requirement will return verification failure.

3.1.5 Miter Model and Mutation

Now, since all of the system requirements are met, we go ahead to search for any redundant component in the design and this is done by automatic generation of miter modules for the system, followed by injection of mutants (mutation) in each miter module [1]. The purpose of this is to kill mutants in the system. A mutant is killed if a block is changed in the design and it caused a significant output change. But if no output change detected (i.e. no mutant killed) after changing the block multiple times, then the particular replaced block is redundant and it can be safely removed from the original system without changing the expected output of the system. This in return actually reduces the size of the system. The more mutants killed in a system, the closer is the system to correctness and the more the confidence of the tester [15]. The system consist of four different logical operator blocks which are AND, OR, NOR, NOT and two time

delay blocks. The logical operator "AND" and "OR" is only considered for miter modules generation. There are seven "AND" blocks and nine "OR" blocks in the system, each of these blocks can be mutated in five different ways i.e. the "AND" block can be changed to OR, NAND, NOR, XOR, NXOR and "OR" block can be changed to AND, NAND, NOR, XOR, X-NOR as described in table 6 below. This means that 40 different miter modules are generated ranging from "miter1_1_1", "miter1_1_2" to "miter2_9_5". But this mutation steps is applied to one block changed in the system at a time. Injection of mutant to a particular block at a time is better compared to mutant injection to many logical operator blocks at a time because the later might result into negative effect cancellation and the system behaves as if nothing has changed [6]. The Matlab script used for automatic generation of the miter modules and the injection of individual mutant is presented at the Appendix A.1. The diagram describing one of the miter models is also shown in Appendix A.2.

ORIGINAL BLOCK		BLOCK TO BE CHANGED TO (MUTANTS)	
1	AND	OR	1
		NAND	2
		NOR	3
		XOR	4
		X-NOR	5
2	OR	XOR	1
		X-NOR	2
		NAND	3
		NOR	4
		AND	5

Table6: Describing Mutation blocks

The expression Miter***x_y_z*** used in this work represents the changes made to the mutated part of the miter model and the meaning of the annotations is described below;

X = represents the number that corresponds to the type of the original block

Y = represents the position of the original block in the system

Z = this is the number that represents the type of the block used as a mutant

The next step is to run the mutated models with Gene-auto. Since there are large numbers of models to run, then this has to be done automatically. The below batch file is used to automatically run the Gene-auto.

```
@echo off
REM list all files with suffix docx in the directory
FOR %%f IN ( miter?_?_?.mdl ) DO (
C:\Users\VICTOR\Documents\Gene-Auto\geneauto2\geneauto2.bat %%f
ECHO %%f
)
```

The simulation is successfully completed. After this level, verification steps with the model checker (CBMC) follows. Any of the models that return a successful verification with the assertion of the original model's outputs and the mutated model's outputs, indicates that the mutated logical operator block in that model is redundant and it can be successfully removed without harming any part of the system, in other word, no mutant is killed or there is an equivalent mutant. Otherwise this will return verification failed. This is shown in figure 16. For this thesis, the verification returned failure for the assertion made for all the miter models (the

original models and the mutated models), which means that the original model's output (behavior) is not equal to the mutated model's output (behavior). In this case it is glaring that the injection of mutants has caused behavioral differences in the mutated models.

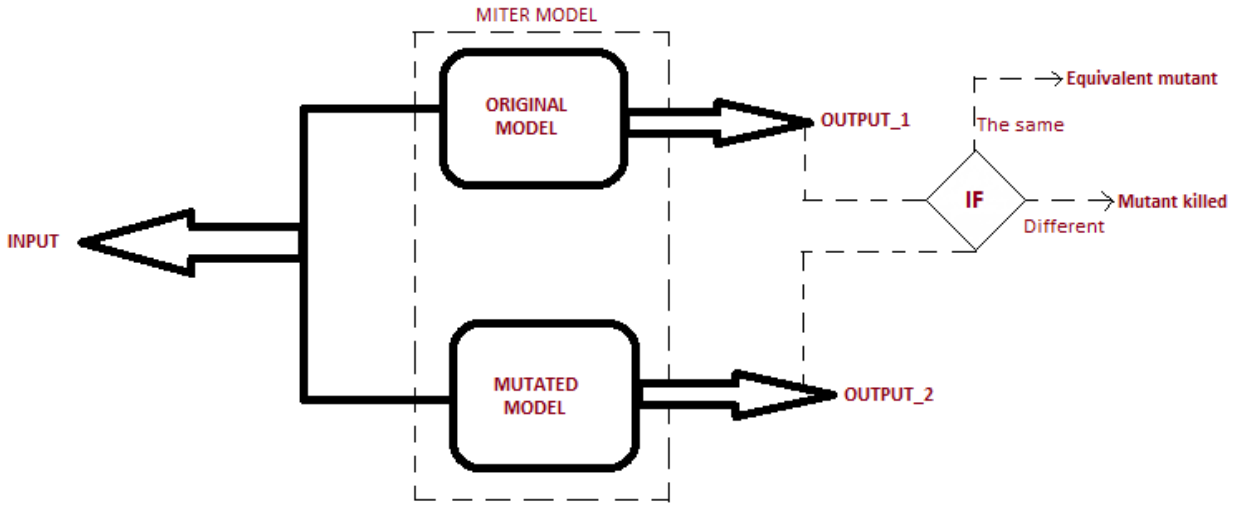


Figure 16: Miter Model Block Diagram

The assertion statement to compare the output of the original model with the output of the mutated model for the miter model "miter2_1_5" is shown below and the verification result is shown in figure 17.

```
void main() {
_Bool t1,t2,t3,t4,t5,t6,t7,t8;
t_miter2_1_5_io *t_io;
t_miter2_1_5_state *t_state;
 miter2_1_5_compute(t_io,t_state);
 t1= t_io->L1_UP == t_io->L1_UP1;
 t2= t_io->L2_UP == t_io->L2_UP1;
 t3= t_io->L2_PUMP == t_io->L2_PUMP1;
```

t4= t_io->L1_PUMP == t_io->L1_PUMP1;

t5= t_io->L1_DOWN == t_io->L1_DOWN1;

t6= t_io->L2_DOWN == t_io->L2_DOWN1;

t7= t_io->OUT_VALVE == t_io->OUT_VALVE1;

t8= t_io->BF_VALVE == t_io->BF_VALVE1;

assert(t1 && t2 && t3 && t4 && t5 && t6 && t7 && t8);

}

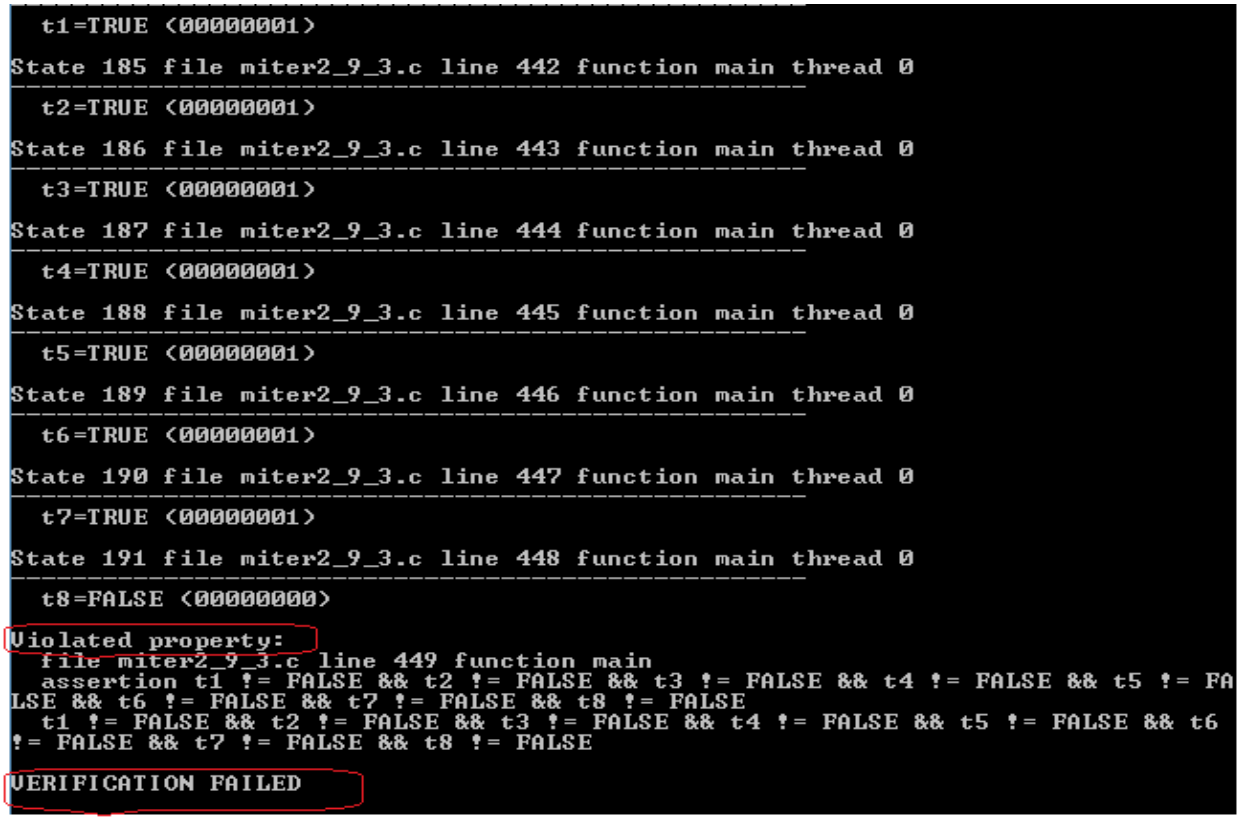


Figure 17: Miter Model Verification Result

Miter Output	Miter1_1_2	Miter1_3_1	Miter1_5_4	Miter1_7_3	Miter2_1_5	Miter2_3_3	Miter2_6_4	Miter2_9_3
t1	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE
t2	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
t3	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
t4	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE
t5	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE
t6	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
t7	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
t8	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE

Table 6: Miter Model Output Result

3.1.6 Observation

The assertion statements were conducted for all the miter models. The above verification table 6 was drawn for the mutant injected at the location close to the system input, around the middle part of the system and close to the system output. It can be seen that the verification failed for all the miter models which indicates that all these mutants were killed. It can also be inferred that the Output t7 is more tolerant to faults because no matter what mutant is injected into the system, the behavior still tends to remain the same except for the mutant injected at location Miter1_7_3. The mutant injected at location Miter1_7_3 actually have great effect on the output t7 because it is located very close to the output as shown in figure 18 below. As it was explained earlier, Miter1_7_3 means that the mutant "OR" is applied to the 7th "AND" block in the system. All other outputs are seen to be more sensitive to faults.

Explanation of M1_7_3

1= represents the original logical operator block which is in this case "AND" block,

7 = means that the "AND" block changed is located at the seventh position in the system, which is very close to the output because there are only seven "AND" block in the system.

3 = indicates that the "AND" block is changed to "NOR" block

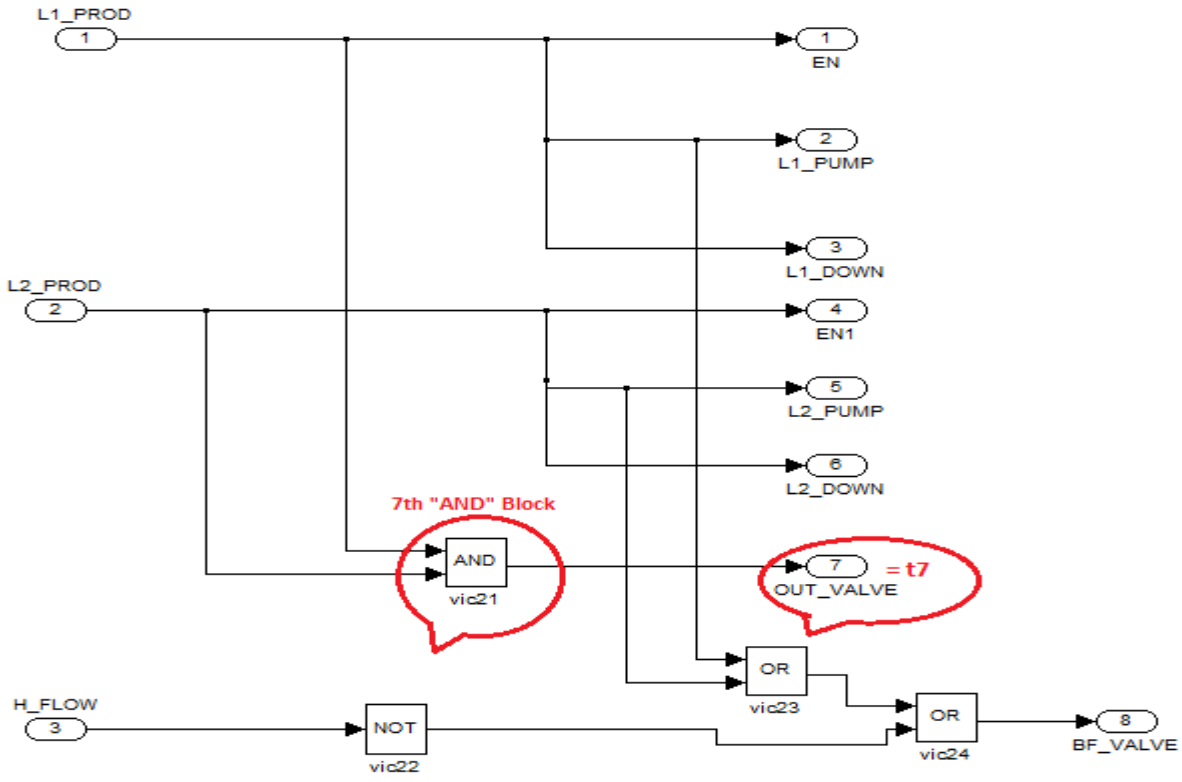


Figure 18: Diagram showing the location of M1_7_3

3.1.7 Future Work

More work can still be done on this system in the future to make the system more robust because high fault tolerant systems can survive more extreme environments and tend to be more reliable. This can be done by finding ways of quenching or absorbing all the effects introduced into the

system by the injected mutants before getting to the system output, irrespective of the inputs to the system. These mutants can be described in real life scenario as unforeseen environmental changes that can affect the system behavior. Implementing this will definitely strengthen the confidence of the system designer as well as the system users.

Chapter 4

Conclusions

We have presented a methodology for model-based verification. We believe that the goals of this thesis work have been met because all the design system properties were explicitly verified and satisfied. We started from an interpretation of the PLC ladder logic represented as a Simulink model, followed by conversion of the Simulink model to C program using Gene-auto. Some main functions and assertion statements were added to conduct the verification using CBMC. We also went further to generate miter models which enable the comparison of the original system behavior with the mutated system behavior. Suggestions on how the system can be made more robust were also discussed.

Appendix A

A.1

The Matlab script that automatically generates the miter models and the injection of mutants is presented below;

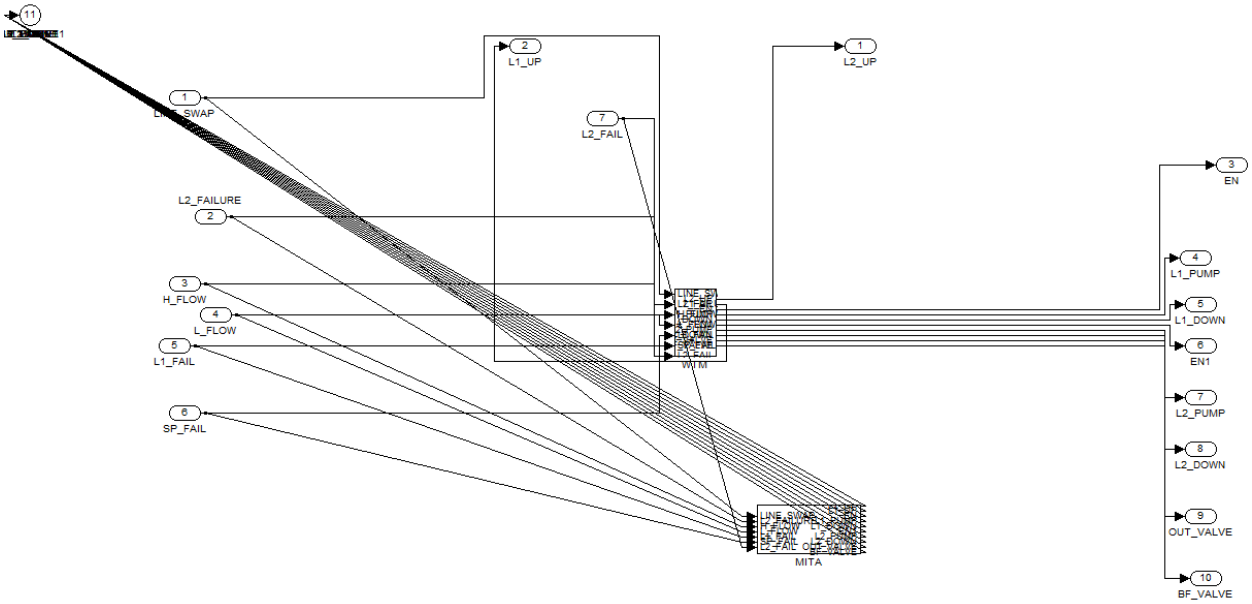
```
function gen_miterv2(model_name)
 % step 1 probe model

 [all_logicBlocks, all_ANDBlocks, all_ORBlocks, all_NOTBlocks] = probe1(model_name);
 N_AND = length(all_ANDBlocks);
 N_OR = length(all_ORBlocks);

 arr = [N_AND N_OR];

 for i = 1:2
    for j = 1 : arr(i)
       for k = 1 : 5
        bdclose('all') ;

        load_system('WATER_TANK_MODELLING3') ;
        new_system('empty', 'Model') ;

        Simulink.SubSystem.copyContentsToBlockDiagram('WATER_TANK_MODELLING3/WTM', 'empty') ;
        Simulink.BlockDiagram.copyContentsToSubSystem('empty', 'WATER_TANK_MODELLING3/MITA') ;

        name = sprintf('miter%d_%d_%d', i, j, k) ;
        save_system('WATER_TANK_MODELLING3', name) ;

        close_system('WATER_TANK_MODELLING3', 0) ;
        close_system('empty', 0) ;
       end
    end
 end
  % generate miter models
  for i = 1:2
    for jj = 1 : arr(i)
       for k = 1 : 5

        name = sprintf('miter%d_%d_%d', i, jj, k) ;
        miter_handler = load_system(name) ;

        mu_sys = find_system(miter_handler, 'SearchDepth', 1, ...
           'BlockType', 'SubSystem', 'Name', 'MITA') ;

        % add 10 outport blocks
        new_b_handler{1} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/L2_UP'], ...
           'MakeNameUnique', 'on') ;
        new_b_handler{2} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/L1_UP'], ...
           'MakeNameUnique', 'on') ;
```

```
      new_b_handler{3} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/EN'], ...
         'MakeNameUnique', 'on') ;
      new_b_handler{4} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/L1_PUMP'], ...
         'MakeNameUnique', 'on') ;
       new_b_handler{5} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/L1_DOWN'], ...
         'MakeNameUnique', 'on') ;
      new_b_handler{6} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/EN1'], ...
         'MakeNameUnique', 'on') ;
      new_b_handler{7} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/L2_PUMP'], ...
         'MakeNameUnique', 'on') ;
      new_b_handler{8} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/L2_DOWN'], ...
         'MakeNameUnique', 'on') ;
      new_b_handler{9} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/OUT_VALVE'], ...
         'MakeNameUnique', 'on') ;
      new_b_handler{10} = add_block('built-in/Outport', [get(mu_sys, 'Parent'), '/BF_VALVE'], ...
         'MakeNameUnique', 'on') ;
      sys_inports = find_system(miter_handler, 'SearchDepth', 1, ...
         'BlockType', 'Inport') ;
      %length(sys_inports)

      port_handlers = get(mu_sys(1), 'PortHandles') ;
       inports = port_handlers.Inport;
      outports = port_handlers.Outport;
      %length(inports)

      for j = 1:length(inports)
        srcPort_handler = get(sys_inports(j), 'PortHandle') ;
        add_line(get(mu_sys, 'Path'), srcPort_handler.Outport(1), inports(j)) ;
      end

      for j = 1:length(outports)
        dstPort_handler = get(new_b_handler{j}, 'PortHandle') ;
        add_line(get(mu_sys, 'Path'), outports(j), dstPort_handler.Inport(1)) ;
      end

      save_system(name) ;
      close_system(name) ;
    end
  end
end

%  generate mutants
 for i = 1:2
   for jj = 1 : arr(i)
     for k = 1 : 5

      name = sprintf('miter%d_%d_%d', i, jj, k) ;
      miter_handler = load_system(name) ;

      mutated_tank = find_system(miter_handler, 'SearchDepth', '1', ...
         'BlockType', 'SubSystem') ;
      %length(mutated_tank)

      if i == 1
```

```
            selectedBlocks = find_system(mutated_tank(1), 'BlockType', 'Logic', 'Operator', 'AND') ;
            length(selectedBlocks)
            for tj = 1:length(selectedBlocks)
                if tj == jj
                    if k == 1
                        set_param(selectedBlocks(tj), 'Operator', 'OR');
                        break;
                    elseif k == 2
                        set_param(selectedBlocks(tj), 'Operator', 'NAND');
                    elseif k == 3
                        set_param(selectedBlocks(tj), 'Operator', 'NOR');
                    elseif k == 4
                        set_param(selectedBlocks(tj), 'Operator', 'XOR');
                    else
                        set_param(selectedBlocks(tj), 'Operator', 'NXOR');
                    end
                end
            end
        else
            selectedBlocks = find_system(mutated_tank(1), 'BlockType', 'Logic', 'Operator', 'OR') ;
        end
        if i == 2
            selectedBlocks = find_system(mutated_tank(1), 'BlockType', 'Logic', 'Operator', 'OR') ;
            length(selectedBlocks)
            for tj = 1:length(selectedBlocks)
                if tj == jj
                    if k == 1
                        set_param(selectedBlocks(tj), 'Operator', 'XOR');
                    elseif k == 2
                        set_param(selectedBlocks(tj), 'Operator', 'NXOR');
                    elseif k == 3
                        set_param(selectedBlocks(tj), 'Operator', 'NAND');
                    elseif k == 4
                        set_param(selectedBlocks(tj), 'Operator', 'NOR');
                    else
                        set_param(selectedBlocks(tj), 'Operator', 'AND');
                    end
                end
            end
        else
            selectedBlocks = find_system(mutated_tank(1), 'BlockType', 'Logic', 'Operator', 'AND') ;
        end


        save_system(name) ;
        close_system(name) ;
    end
  end
end
```

A.2

The diagram of one of the generated miter models is shown below;



A.3

Gene-auto C program for WATER_TANK_MODELING3

```
/*
    WATER_TANK_MODELLING3.c
    Generated by Gene-Auto toolset ver 2.4.10
    (launcher GALauncher)
    Generated on: 17/04/2015 08:42:37.581
    source model: WATER_TANK_MODELLING3
    model version: 7.5
    last saved by:
    last saved on:
*/

/* Includes */

#include "WATER_TANK_MODELLING3.h"

/* Function definitions */

void WATER_TANK_MODELLING3_init(t_WATER_TANK_MODELLING3_state *_state_) {
    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<SequentialBlock: name=Unit Delay>  */

    _state_->Unit_Delay_memory_1 = TO_GABOOL(0);
    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<SequentialBlock: name=Unit Delay>  */
    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<SequentialBlock: name=Unit Delay>  */
```

```
    _state_->Unit_Delay_memory_2 = TO_GABOOL(0);

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<SequentialBlock: name=Unit Delay>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SequentialBlock:
name=Unit Delay>  */

    _state_->Unit_Delay_memory_3 = TO_GABOOL(0);

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SequentialBlock:
name=Unit Delay>  */

}


void WATER_TANK_MODELLING3_compute(t_WATER_TANK_MODELLING3_io *_io_,
t_WATER_TANK_MODELLING3_state *_state_) {

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=LINE_SWAP>/<OutDataPort: name=>  */

    GABOOL LINE_SWAP;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L2_FAILURE>/<OutDataPort: name=>  */

    GABOOL L2_FAILURE;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=H_FLOW>/<OutDataPort: name=>  */

    GABOOL H_FLOW;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L_FLOW>/<OutDataPort: name=>  */

    GABOOL L_FLOW;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L1_FAIL>/<OutDataPort: name=>  */

    GABOOL L1_FAIL;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=SP_FAIL>/<OutDataPort: name=>  */

    GABOOL SP_FAIL;
```

```
  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L2_FAIL>/<OutDataPort: name=> */

  GABOOL L2_FAIL;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SequentialBlock:
name=Unit Delay>/<OutDataPort: name=> */

  GABOOL Unit_Delay_2;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator>/<OutDataPort: name=> */

  GABOOL Logical_Operator;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator1>/<OutDataPort: name=> */

  GABOOL Logical_Operator1_2;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator2>/<OutDataPort: name=> */

  GABOOL Logical_Operator2_2;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator3>/<OutDataPort: name=> */

  GABOOL Logical_Operator3_2;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator4>/<OutDataPort: name=> */

  GABOOL Logical_Operator4_2;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator5>/<OutDataPort: name=> */

  GABOOL Logical_Operator5;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<SequentialBlock: name=Unit Delay>/<OutDataPort: name=> */

  GABOOL Unit_Delay_1;

  /* Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator10>/<OutDataPort: name=> */

  GABOOL Logical_Operator10;
```

```
    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator11>/<OutDataPort: name=>  */

    GABOOL Logical_Operator11;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator12>/<OutDataPort: name=>  */

    GABOOL Logical_Operator12;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator13>/<OutDataPort: name=>  */

    GABOOL Logical_Operator13;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator6>/<OutDataPort: name=>  */

    GABOOL Logical_Operator6;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator7>/<OutDataPort: name=>  */

    GABOOL Logical_Operator7;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator8>/<OutDataPort: name=>  */

    GABOOL Logical_Operator8;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator9>/<OutDataPort: name=>  */

    GABOOL Logical_Operator9;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<SequentialBlock: name=Unit Delay>/<OutDataPort: name=>  */

    GABOOL Unit_Delay_3;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator14>/<OutDataPort: name=>  */

    GABOOL Logical_Operator14;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator15>/<OutDataPort: name=>  */

    GABOOL RS_L2_S;
```

```
    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator16>/<OutDataPort: name=>  */

    GABOOL Logical_Operator16;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator17>/<OutDataPort: name=>  */

    GABOOL Logical_Operator17;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator18>/<OutDataPort: name=>  */

    GABOOL Logical_Operator18;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator19>/<OutDataPort: name=>  */

    GABOOL Logical_Operator19;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator20>/<OutDataPort: name=>  */

    GABOOL Logical_Operator20;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator1>/<OutDataPort: name=>  */

    GABOOL Logical_Operator1_1;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator2>/<OutDataPort: name=>  */

    GABOOL Logical_Operator2_1;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator3>/<OutDataPort: name=>  */

    GABOOL Logical_Operator3_1;

    /*  Output from <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator4>/<OutDataPort: name=>  */

    GABOOL Logical_Operator4_1;

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<SequentialBlock: name=Unit Delay>  */

    Unit_Delay_3 = _state_->Unit_Delay_memory_1;
```

```
    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<SequentialBlock: name=Unit Delay>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=LINE_SWAP>  */

    LINE_SWAP = _io_->LINE_SWAP;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=LINE_SWAP>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L2_FAIL>  */

    L2_FAIL = _io_->L2_FAIL;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L2_FAIL>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L2_FAILURE>  */

    L2_FAILURE = _io_->L2_FAILURE;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L2_FAILURE>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<SequentialBlock: name=Unit Delay>  */

    Unit_Delay_1 = _state_->Unit_Delay_memory_2;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<SequentialBlock: name=Unit Delay>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator1>  */

    Logical_Operator1_2 = !Unit_Delay_1;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator1>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator>  */

    Logical_Operator = LINE_SWAP && Logical_Operator1_2;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator>  */
```

```
    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator2>  */

    Logical_Operator2_2 = LINE_SWAP && Unit_Delay_1;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator2>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator4>  */

    Logical_Operator4_2 = !Logical_Operator2_2;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator4>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator5>  */

    Logical_Operator5 = Logical_Operator || Unit_Delay_1;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator5>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator3>  */

    Logical_Operator3_2 = Logical_Operator4_2 && Logical_Operator5;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem>/<CombinatorialBlock: name=Logical\nOperator3>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator14>  */

    Logical_Operator14 = !Logical_Operator3_2;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator14>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator15>  */

    RS_L2_S = Logical_Operator14 || L2_FAILURE;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator15>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=H_FLOW>  */
```

```
    H_FLOW = _io_->H_FLOW;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=H_FLOW>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L_FLOW>  */

    L_FLOW = _io_->L_FLOW;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L_FLOW>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator16>  */

    Logical_Operator16 = !(H_FLOW || L_FLOW);

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator16>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator9>  */

    Logical_Operator9 = !(H_FLOW || L_FLOW);

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator9>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L1_FAIL>  */

    L1_FAIL = _io_->L1_FAIL;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=L1_FAIL>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=SP_FAIL>  */

    SP_FAIL = _io_->SP_FAIL;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SourceBlock:
name=SP_FAIL>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator10>  */

    Logical_Operator10 = Logical_Operator9 || L1_FAIL || SP_FAIL;
```

```
    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator10>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator12>  */

    Logical_Operator12 = !Logical_Operator10;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator12>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator4>  */

    Logical_Operator4_1 = !H_FLOW;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator4>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SequentialBlock:
name=Unit Delay>  */

    Unit_Delay_2 = _state_->Unit_Delay_memory_3;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SequentialBlock:
name=Unit Delay>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator6>  */

    Logical_Operator6 = !Unit_Delay_2;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator6>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator7>  */

    Logical_Operator7 = Logical_Operator3_2 && Logical_Operator6;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator7>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator8>  */

    Logical_Operator8 = Logical_Operator7 || L2_FAILURE;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator8>  */
```

```
    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator11>  */

    Logical_Operator11 = Unit_Delay_3 || Logical_Operator8;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator11>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator18>  */

    Logical_Operator18 = RS_L2_S || Unit_Delay_2;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator18>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator13>  */

    Logical_Operator13 = Logical_Operator11 && Logical_Operator12;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem1>/<CombinatorialBlock: name=Logical\nOperator13>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L1_UP>  */

    _io_->L1_UP = Logical_Operator13;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L1_UP>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock: name=EN>
*/

    _io_->EN = Logical_Operator13;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock: name=EN>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator17>  */

    Logical_Operator17 = SP_FAIL || Logical_Operator16 || Logical_Operator13 || L2_FAIL;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator17>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator19>  */
```

```
    Logical_Operator19 = !Logical_Operator17;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator19>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator20>  */

    Logical_Operator20 = Logical_Operator18 && Logical_Operator19;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem2>/<CombinatorialBlock: name=Logical\nOperator20>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L2_UP>  */

    _io_->L2_UP = Logical_Operator20;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L2_UP>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L1_PUMP>  */

    _io_->L1_PUMP = Logical_Operator13;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L1_PUMP>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L1_DOWN>  */

    _io_->L1_DOWN = Logical_Operator13;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L1_DOWN>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=EN1>  */

    _io_->EN1 = Logical_Operator20;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock: name=EN1>
*/

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator2>  */

    Logical_Operator2_1 = Logical_Operator13 && Logical_Operator20;
```

```
    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator2>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L2_PUMP>  */

    _io_->L2_PUMP = Logical_Operator20;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L2_PUMP>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator1>  */

    Logical_Operator1_1 = Logical_Operator13 || Logical_Operator20;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator1>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L2_DOWN>  */

    _io_->L2_DOWN = Logical_Operator20;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=L2_DOWN>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator3>  */

    Logical_Operator3_1 = Logical_Operator1_1 || Logical_Operator4_1;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SystemBlock:
name=Subsystem3>/<CombinatorialBlock: name=Logical\nOperator3>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=OUT_VALVE>  */

    _io_->OUT_VALVE = Logical_Operator2_1;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=OUT_VALVE>  */

    /*  START Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=BF_VALVE>  */

    _io_->BF_VALVE = Logical_Operator3_1;

    /*  END Block: <SystemBlock: name=WATER_TANK_MODELLING3>/<SinkBlock:
name=BF_VALVE>  */
```

```
    /*  START Block memory write: <SystemBlock:
name=WATER_TANK_MODELLING3>/<SequentialBlock: name=Unit Delay>  */

    _state_->Unit_Delay_memory_3 = Logical_Operator20;

    /*  END Block memory write: <SystemBlock:
name=WATER_TANK_MODELLING3>/<SequentialBlock: name=Unit Delay>  */

    /*  START Block memory write: <SystemBlock:
name=WATER_TANK_MODELLING3>/<SystemBlock: name=Subsystem>/<SequentialBlock:
name=Unit Delay>  */

    _state_->Unit_Delay_memory_2 = Logical_Operator3_2;

    /*  END Block memory write: <SystemBlock:
name=WATER_TANK_MODELLING3>/<SystemBlock: name=Subsystem>/<SequentialBlock:
name=Unit Delay>  */

    /*  START Block memory write: <SystemBlock:
name=WATER_TANK_MODELLING3>/<SystemBlock: name=Subsystem1>/<SequentialBlock:
name=Unit Delay>  */

    _state_->Unit_Delay_memory_1 = Logical_Operator13;

    /*  END Block memory write: <SystemBlock:
name=WATER_TANK_MODELLING3>/<SystemBlock: name=Subsystem1>/<SequentialBlock:
name=Unit Delay>  */

}
```

REFERENCES

[1]V. Okun, 'Specification Mutation for Test Generation and Analysis', Ph.D Theses, University of Maryland Baltimore County, 2004.

[2]M. Mäkinen, 'Model Based Approach to Software Testing', Maters Theses, Helsinki University of Technology, 2007.

[3]D. Heise, 'Automated Generation of Simulink Models for Enumeration Hybrid Automata', Masters Theses, University of Tennessee, Knoxville, 2013.

[4]A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer and G. Weissenbacher, 'Mutation-based test case generation for simulink models', in *8th international conference on Formal methods for components and objects*, Eindhoven, The Netherlands, 2009.

[5]N. He, P. Rümmer and D. Kroening, 'Test-case generation for embedded simulink via formal concept analysis', in *48th Design Automation Conference*, San Diego, California, 2011.doi: [DOI 10.1145/2024724.2024777]

[6]O. Kupferman, W. Li and S. Seshia, 'A theory of mutations with applications to vacuity, coverage, and fault tolerance', in *Formal Methods in Computer-Aided Design (FMCAD)*, IEEE Computer Society, 2008, pp. 1-9.

[7]D. Große, U. KÜhne and R. Drechsler, 'Estimating functional coverage in bounded model checking', in *Design Automation and Test in Europe*, 2007, pp. 1176–1181.

[8]The MathWorks: Simulink design verifier [Online]. Available: http://www.mathworks.com/help/sldv/%20(2009)%20version%201.5

[9]D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu and S. Sabina, 'Automatic test generation for coverage analysis using CBMC', in *Computer Aided Systems Theory-EUROCAST 2009*, 2009, pp. 287-294.

[10]F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella and M. Vardi, 'Benefits of Bounded Model Checking at an Industrial Setting', in *13th International Conference on Computer Aided Verification*, 2001, pp. 436-453.

[11]A. Biere, A. Cimatti, E. Clarke, O. Strichman and Y. Zhu, 'Bounded Model Checking', *Advances in Computers*, vol. 58, pp. 117-148, 2003.

[12]E. Clarke, D. Kroening and F. Lerda, 'A tool for checking ANSI-C programs', in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 168-176.

[13]M. Ryabtsev and O. Strichman, 'Translation validation: From simulink to c', in *Computer Aided Verification*, Springer Berlin Heidelberg, 2009, pp. 696-701.

[14]H. Chockler, O. Kupferman, R. Kurshan and M. Vardi, 'A practical approach to coverage in model checking', in *Computer Aided Verification*, Springer Berlin Heidelberg, 2001, pp. 66-78.

[15]D. Schuler, V. Dallmeier and A. Zeller, 'Efficient mutation testing by checking invariant violations', in *International Symposium on Software Testing and Analysis (ISSTA)*, ACM, New York, 2009, pp. 69–80.

[16]B. Grun, D. Schuler and A. Zeller, 'The impact of equivalent mutants', in *Software Testing, Verification and Validation Workshops, 2009*, International Conference on. IEEE, 2009, pp. 192-199.

[17]B. Zoubek, J. Roussel and M. Kwiatkowska, 'Towards automatic verification of ladder logic programs', in *IMACS-IEEE" CESA'03":" Computational Engineering in Systems Applications*, 2003, p. CD-ROM.

[19]B. Beizer, *Black-box testing*. New York: Wiley, 1995.

[20]B. Beizer, *Software testing techniques*. New York: International Thomson Computer Press, 1990.

[21]W. Stephan, 'Test models and coverage criteria for automatic model-based test generation with UML state machines', PhD dissertation, Humboldt University of Berlin, 2010.

[22]S. Nidhra, 'Black Box and White Box Testing Techniques - A Literature Review', *IJESA*, vol. 2, no. 2, pp. 29-50, 2012.

[23]L. Apfelbaum and D. John, 'Model based testing', in *Software Quality Week Conference*, 1997, pp. 296-300.

[24]A. Pretschner, 'Model-based testing', in *27th international conference on Software engineering*, St. Louis, MO, USA, 2005, pp. 722-723.doi:[DOI 10.1145/1062455.1062636]

[25]S. Rayadurgam and M. Heimdahl, 'Coverage based test-case generation using model checkers', in *8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, Washington, DC, U.S.A, 2001, pp. 83–91.

[26]'CBMC–C bounded model checker', in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, 2014, pp. 389-391.

[27]P. Mosterman, 'Model-based design of embedded systems', in *IEEE international conference on microelectronic systems education, vol 3*, IEEE, 2007, pp. 3-3.

[28]G. Nicolescu and P. Mosterman, *Model-based design for embedded systems*. Boca Raton, FL: CRC Press, 2010.

[29]S. Muhammad and Y. Labiche, 'A systematic review of model based testing tool support', Tech. Rep, Carleton University, Canada, 2010.

[30]E. Rugina, D. Thomas, X. Olive and G. Veran, 'Gene-auto: Automatic software code generation for real-time embedded systems', in *DASIA 2008 Data Systems In Aerospace*, 2008, pp. vol. 665, p. 28.