1998

Fault-injection through model checking via naive assumptions about state machine synchrony semantics

Sabina Joseph
West Virginia University

Fault-Injection through Model Checking via Naive Assumptions about State Machine Synchrony Semantics

Sabina Joseph

Thesis submitted to the College of Engineering and Mineral Resources
of
West Virginia University
in Partial Fulfillment of the Requirements
for The Degree of

Master of Science
in
Computer Science

Jack Callahan, Chair
Steve Easterbrook
Bojan Cukic

December 8, 1998
Morgantown, West Virginia

Keywords: Fault-injection, State Machine,
Model Checking, Linear Temporal Logic

Acknowledgements

None of this would have been possible, without the tremendous love, unquestionable belief, motivation, and support in infinite ways from my parents, brother and sister.

I would like to convey my sincere thanks to Eric, for his undeniable love, patience and encouragement.

I would sincerely like to thank Dr. Callahan, for his valuable guidance and assistance towards the completion of this thesis. His ideas and time to listen to my thoughts and inferences are greatly appreciated. I would like to thank him for giving me an opportunity to be part of some very intelligent group of people in the SRL (Software Research Laboratory).

I gratefully acknowledge, the help and direction, by Dr. Easterbrook and Dr. Cukic and for agreeing to be part of my thesis committee.

Special thanks are due to John Powell for all our discussions and remarks during my research. I appreciate all the support by Vikram, Reshma and other fellow SRL members.

Thank you, Chamu, for teaching me to follow my dreams and showing me how precious life is.

The author would like to thank Heather, Dawn and all her friends and well-wishers at home, who have helped her in various ways to achieve this.

The financial support provided by NASA-IV&V Cooperative Research Project in the form of a Graduate Research Assistantship is gratefully acknowledged.

Contents

List of Figures

List of Tables

Fault-Injection through Model Checking via Naive Assumptions about State Machine Synchrony Semantics

Sabina Joseph

(Abstract)

Software behavior can be defined as the action or reaction of software to external and/or internal conditions. Software behavior is an important characteristic in determining software quality. Fault-injection is a method to assess software quality through its' behavior. Our research involves a fault-injection process combined with model checking. We introduce a concept of "naive assumptions" which exploits the assumptions of execution order, synchrony and fairness. "Naive assumptions" are applied to inject faults into our models. We use linear temporal logic to examine the model for anomalous behaviors. This method shows us the benefits of using fault-injection and model checking and the advantage of the counter-examples generated by model checkers. We illustrate this technique on a fuel injection Sensor Failure Detection system and discuss the anomalies in detail.

Chapter 1

Introduction

We introduce a process of combining fault-injection with model checking through “naive assumptions” in this thesis. “Naive assumptions” are defined as the various assumptions that can be made about execution order, synchrony and fairness. Our method is carried out on a state machine model of a fuel injection Sensor Failure Detection (SFD) system. We specify the state based representation of the Sensor Failure Detection (SFD) system in a model checking tool while injecting faults through “naive assumptions” and inputs. We make “naive assumptions” about execution synchrony, i.e., we take advantage of loosely specified semantics of the Sensor Failure Detection (SFD) system and the power of asynchronous process interactions in the model checker. Our Sensor Failure Detection (SFD) system models are indeed “naive” with respect to any assumptions about synchrony, execution order and fairness. Once the models are specified, we formulate different properties in Linear Temporal Logic (LTL) to validate different requirements. Based on the properties and the results, we classify the results from the properties into different fault classes and discuss the anomalies in each fault class.

The failure of a computer system or software can lead to a myriad of problems. Since a failure in software can result in bad repercussions, the importance of software quality cannot be overemphasized. In this age, where computers are omnipresent, from watches, everyday appliances, to cars, and planes, it is essential that we have good techniques for assessment of software quality. In spite of this, today's techniques are not sufficient. Software quality is defined as the totality of features and characteristics of software that bear on its ability to satisfy stated or implied needs. Software quality is not an easy characteristic to measure, as it can be determined through an assortment of factors such as the development process, skills of people involved, clarity and interpretation of specifications, management, and environment. A few software quality assessment techniques used today are metrics, formal methods, and testing. These techniques do add to the quality of software but they do not capture the true nature of software which lies in its' behavior.

Software behavior is the action or reaction of software to a change in external and/or internal conditions. Software fault-injection focuses on the behavior of the software. Fault-injection tries to determine how good or bad the software will behave under anomalous circumstances. It injects faults into the software and observes how the software behaves in response to the injected faults. Fault-injection is a technique which when applied to software tests the software under reasonable anomalies but the key is to see what happens under unreasonable anomalies. Unreasonable anomalies are nonsensical circumstances which may seem insipid but it is essential to view the behavior of the software under these conditions. The success of fault-injection lies in the

interpretation of the results. It is not to be used as a standalone software assessment method but rather in a combination with the other methods such as testing, formal methods and metrics. The greatest benefit from it, is when the software doesn't behave as anticipated in response to the injected faults. Since we are looking at behavior, we need the capability to be able to search through all states and transitions that a software can go through. A method, which enables us to do this, is model checking. Model checking is a technique under the formal methods umbrella. It exhaustively searches through all paths that a program can go through to ascertain that a certain property never occurs. Properties are usually specified in some form of temporal logic formula. A model checker runs through the validation process and if it finds an undesirable behavior, that is, if a property is shown to exist in the software, it will generate a counter-example in the form of a trace or sequence of events. These counter-examples are test cases that activate the anomaly. Thus, model checking provides an automatic tool for test case generation. The counter-examples also constitute the results of fault-injection through model checking.

The main objectives of the thesis are:

* To propose methods to inject faults through model checking via "naive assumptions".
* To apply these methods on a case study.
* To formulate Linear Temporal Logic (LTL) formulas to verify and validate the models.
* To classify and interpret the results, related requirements and the properties.

The rest of this thesis is organized into four chapters which review (Chapter 2) the related work and literature related to the present investigation; describe (Chapter 3) the methodology and case study; describe and discuss (Chapter 4) the results of the investigation; summarize and conclude (Chapter 5) by pointing out the need for future work. Appendix A contains the source code for the asynchronous Sensor Failure Detection (SFD) system model in Spin. Appendix B contains the source code for the synchronous Sensor Failure Detection (SFD) system model in Spin. Appendix C contains an example of a counter-example.

Chapter 2

Related Work

This chapter presents a review of the related literature to this thesis. The contents of the chapter are organized as

* Fault-injection
* Model checking
* Linear Temporal Logic (LTL)
* Spin

2.1 Fault-Injection

A failure in a computer can result in a simple discomfort or major economic losses. Sometimes, human lives are lost, as in the Therac-25 accident. The causes of these failures range from physical faults, maintenance errors, design and implementation mistakes to user or operator mistakes. The use of computer systems in our daily lives has increased our dependence on computer systems. All this makes it very important to ascertain the software quality but this is easier said than done. Academia, industry and

government are all working towards improving software quality through research and experiments. But the present assessment techniques for software quality are not sufficient. The main reason that most of these techniques focus on the software process rather than the software product. Process refers to a prescribed development technique being used within the software development lifecycle and product refers to software or hardware [5]. Formal methods that is a process-oriented technique, don't give the required confidence of how the software will behave when released into the real world. Another reason why process-oriented assessment may not be the right approach to predict software quality, is the fact that a process involves a series of steps. Hence it has the potential of being botched as it maybe applied in an erroneous manner due to the lack of proper training and knowledge [5].

Software testing is a product-oriented assessment of software quality. But there are drawbacks to testing. Random testing which is used to make predictions of future behavior of software is only as good as the number of tests performed. The solution to this is exhaustive testing, which is both impractical and cannot demonstrate the capability of the software when it encounters an undesirable circumstance. Software metrics are another product oriented assessment. Metrics focus on program structure and statistics [13]. Due to this, metrics cannot capture the essence of software. This is not to imply that testing, formal methods and metrics do not add to the value of software. They have their advantages and when the right technique is used at the right time and place, it does improve software quality. But the characteristic that defines software quality is software behavior. Software behavior is dynamic in nature as it is the action and reaction of the

software to a change in external and/or internal conditions. It can vary with respect to the connotation and situation the software is used in. According to Jeffrey Voas [13], "software behavior can be viewed without looking at the software's developmental history, organization in which it is developed and/or technique used by developers." Fault-injection techniques are quantifiable and focus on software behavior. Software fault-injection purposely injects faults into the software and checks to see the behavior of software in response to the injected faults. Before explaining the background, history, methods and tools of fault-injection, we define the related terms in the next section.

2.1.1 Nomenclature

Software verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfies the requirements imposed at the start of that phase [33]. Software validation is the process of evaluating a system or a component during or at the end of the development process to determine whether it satisfies specified requirements [33]. An anomaly is some event which if present in a software can change the behavior of the software in the future by corrupting some value in the software. A fault is a flaw in the program. A fault is a preanomaly event that, if executed and given some set of inputs, will corrupt a value in the program. An error is a mental mistake or fallacy made by a programmer that results in a fault [5]. Software failure is a result of a software action or reaction which deviates from the specified requirement. Software testing is the process of executing a program or system with the intent of finding an error. A test case has an identity and is associated with program behavior. A test case has a set of inputs and list of expected outputs. Fault tolerance is

the ability of a system or component to continue normal operation despite the presence of hardware or software failures. Other terms will be explained as and when deemed necessary.

2.1.2 Fault-Injection Background

Fault-injection focuses on determining how well or bad a piece of software behaves under a range of anomalous circumstances. It is incapable of determining correctness. It is only capable of illustrating the kinds of outputs a program can produce under anomalous conditions. Hence it is interesting and important to observe the behavior of the software under reasonable and unreasonable anomalies. Unreasonable anomalies are nonsensical circumstances which may seem insipid but it is essential to view the behavior of the software under these conditions. According to Jeffrey Voas [5], "fault-injection is not concerned with "why" a certain anomalous event may have occurred with respect to some executing code, all fault-injection is concerned with is "what" happens after an event simulated by fault-injection occurs."

There are many kinds of faults, such as hardware faults occurring during system operation, permanent faults caused by irreversible device failures, intermittent faults which tend to fluctuate between periods of erroneous activity and dormancy and software faults caused by incorrect specification, design or coding of a program. The number of anomalies that can occur is large but the following are the candidates for fault-injection: 1. Problems that arise from code defects. 2. Problems related to human factor errors. 3. Problems with corrupt data being read in from stored files and 4. Problems caused by

external failures. In this thesis, we are concerned with software faults and anomalies from code defects and human factor errors.

2.1.3 History

Fault-injection techniques fall under the branch of processes called fault-based. The earliest work in software fault-injection can be traced to Harlan Mill's fault seeding approach which emerged as early as 1972. Fault-seeding is a fault-based technique that estimates both the number of faults remaining as well as their type. Mill's work involved estimating reliability using fault seeding [5]. The next work in fault-injection was software mutation. It is concerned with modifying the syntax of the code. Fault-injection can be applied to hardware and software systems. Fault-injection is not just present in software but its' roots can be seen in other scientific and manufacturing disciplines. In medical world, in order to see if an antidote for a poison works, it is tested on other living beings before it is tested on human beings. The poison and the antidote are injected into another living being and once the results are satisfactory only then is it approved for human beings. In the manufacturing industry, metals are tested for their strength by applying pressure on them to test their malleability before it can be used in the manufacturing industry.

According to Clark [1], there are three types of fault-injection experiments: system abstraction, fault model and injection method, and dependability measure. Fault-injection has been performed on hardware and software of computer systems. Due to the complex nature of hardware systems, it is difficult to inject faults into internal nodes. On

the other hand, simulation fault-injection gives access to internal nodes and makes it easy to inject faults plus simulated fault-injection is low-cost and can be applied much earlier. It can also support all kinds of abstractions such as architectural, electrical, logical, and functional. Any fault-injection experiment requires selection of a fault model. For transient faults, an inversion model is used where a fault produces an error with the opposite logical value [1]. Other models range from detailed device-level to simplified functional-level models, to represent faults. Once a fault model is chosen, the next step is to decide where to insert a fault.

For physical systems, faults can be injected into IC leads, circuit board connectors, and system back plane through corrupt signals. One can use heavy-ion radiation and/or power supply disturbances to inject internal faults into hardware. Trace injection for injecting faults into a computer system uses custom-monitoring hardware or software to periodically sample machine state or record memory references on an operational system. The acquired trace is used to simulate system behavior, as errors that mimic faults in the instrumented components are inserted into the trace. State mutation can be used for injecting faults using scan paths, program debuggers, and system calls.

2.1.4 Fault-Injection Fundamentals

Faults can be injected into inputs, different points in the code and different locations in the state space. The results of the injected faults are seen in the output and different locations in the state space. There are different locations in the state space of the code where faults can be injected. These locations are termed as fault-injection points. Fault-

injection points can be locations where values of variables are changed, variables are initialized, conditional statements are executed which affect the control flow, function calls, return statements and input/output statements. According to Jeffrey Voas [5], one requirement for fault-injection is to use legal inputs; "legal inputs refers to all members from the input value space, that is, those inputs upon which the software is expected to work." The reasons are, legal inputs are useful if collecting metrics and makes it easier to classify the outputs. Outputs are very important to fault-injection. Without the appropriate training, expertise, tools and methods to study the outputs, fault-injection would be futile.

"Instrumentation is a way to collect information about what is occurring internally in the software and processing that information. [5]" It can be done intrusively and non-intrusively. Fault-injection is intrusive if code is modified whereas it is non-intrusive if it is done from the outside through external tools. Fault-injection is costly so it should be applied at the right place. In [5], some broad rules for deciding whether the code is ready for fault-injection are: (1) The code complies on a system (2) The code is deterministic (3) The code does not include infinite loops or if the code runs continuously, there is at least some reasonable way to determine when one run of the code ends and the other begins.

2.1.5 Fault-Injection Applications

Fault-injection was not used in educational institutions and in research projects until the mid-1980's. We have detailed a few applications in fault-injection. Choi and Iyer

applied fault-injection to study error propagation in a jet-engine controller. They used the Focus simulation environment to inject transient faults into one of the two microprocessors in an HS1602 jet-engine controller. They detected the most potent locations for incorporating additional fault-tolerant features. Karlsson et al. [27] combined radiation and power supply disturbances to examine the propagation of internal errors to the bus of an MC6809E. They injected transient faults into this microprocessor by exposing it to heavy ions and pulses into the power supply. Chillarege and Iyer [28] were among the first to measure fault and error latency in memory via trace injection. They also employed trace injection to investigate the relationship between system workload and memory error latency. Czeck and Siewiorek [29] used simulated fault-injection to study the effects of gate-level faults on program behavior in the IBM RT PC. Czeck and Siewiorek later constructed a model for predicting faulty system behavior from workload attributes such as instruction type, control flow structure and instruction mix. Chillarege and Bowen [30] ushered the concept of failure acceleration to increase the speed at which a system transitions between good, erroneous, and failed states during fault-injection experiments. They attained this by decreasing the fault and error latency and increasing the probability of a fault causing a failure without altering the fault model. Goswami and Iyer [31] studied in detail, the impact of latent and correlated transient errors on a commercial fault-tolerant system's availability through the triple-modular redundant (TMR) processing core of the Tandem Integrity S2.

2.1.6 Fault-Injection Tools

Fault-injection experiments are not based around any formal system, hence, there is no generalized principles for fault-injection. Due to this, most fault-injection experiments concern particular systems so it is difficult to apply those same methods to a different system. Also the complexity of today's systems implies large fault spaces. Due to all this, fault-injection tools integrate tools with methods to conduct experiments. We present a few fault-injection tools in existence. *Messaline*, developed by PAAS-CNRS (Laboratory for the Analysis of Systems Architectures at the National Center for Scientific Research), France, is based on formalized methodology. The result is a flexible testbed capable of simultaneously injecting multiple, pin-level faults into different target systems to collect coverage, latency, and error-propagation measurements with a host computer managing fault-injection. *Fiat* (Fault-Injection-Based Automated Testing) environment used software implemented fault-injection to set and clear bytes in the memory images of programs to evaluate the dependability of fault-tolerant distributed systems. *Ferrari* (Fault and Error Automatic Real-Time Injector) traps instructions affected by the fault so that a routine can be executed to mimic system behavior in the presence of the real fault. *Focus* simulation environment conducts fault sensitivity experiments on chip-level designs by injecting transient faults through a runtime modification of the circuit. The *Depend* environment is a joint dependability and performance evaluation tool that analyzes fault-tolerant architectures at the system level. It does this by a library of objects to behaviorally model a system's hardware components while the objects automatically inject the faults. *React* is a software testbed that abstracts

multiprocessor systems at the architectural level through simulated fault-injection to measure dependability.

2.1.7 Fault-Injection Limitations

Fault-injection consists of a family of techniques, experiments and methods. Hence if it is not used properly like any other method the result can be disastrous. We need to understand when, where and why fault-injection needs to be used before it is used. Software fault-injection simulates the events which let's us observe the behavior of the software in the future. But there is no way possible to know all the faults which can be simulated as the list of anomalies for any software is intractable. Fault-injection cannot prove correctness of software. The process of determining where to inject the faults is complex. We need a thorough understanding of the inputs and outputs we need to cover. The most important aspect in fault-injection is the interpretation of the results. Hence we need appropriate methods and expertise in place to interpret the results. Fault-injection experiments, techniques, methods and their results exemplify the anomalies simulated, inputs employed and the scrutiny of the results [5]. Fault-injection is a dynamic technique by nature and is employed to study the behavior of a product, which can be the software or hardware.

2.2 Formal Methods

Formal methods is the use of concepts from formal logic and discrete mathematics in the specification, design and building of computer systems and software. Formal methods logically calculates if the requirements are consistent with the design and if the properties

present are due to the requirements [32]. It translates a non-mathematical specification of the system into a formal specification using some formal language [32]. Formal methods help better understanding of the problem, helps detect defects early, gives an abstract view of the system and detects problems which may not have been detected with traditional testing. They can be applied either in all or selected stages of the software development lifecycle, all or selected sub-components of the system, all or selected system functions or a combination of all of these in varying degrees. Formal methods has its' limitations in the sense, that specifications can be misinterpreted during formal specification and proofs can be miscalculated. It is not to be used as a standalone method to assess software quality but rather in combination with other software assessment methods. Formal methods can include abstraction, formal specifications, model checking and proofs. Abstraction is ignoring needless detail, picking out the applicable detail which helps to focus on the most significant properties and not get tied up in complex detail. Formal specification uses a formal language to translate a non-mathematical model of the system. Proofs involve using a set of rules to credibly debate the validity of the system requirements. This thesis focuses on the model checking technique of formal methods.

2.2.1 Model Checking

Model checking is an automatic method for verifying correctness of systems that has been successfully applied to the verification of industrial systems. The technique is especially aimed at the verification of reactive, embedded systems, that is, systems that are in constant interaction with the environment. Characteristic for model checking is that it

can be relatively easy to apply at any stage of the existing software process without causing major disruptions. To apply model checking to the verification of systems, we need the description of the behavior of the system in some state based formalism. In state based formalisms, the behavior of the system is described in terms of local state changes or events. The global behavior of the system is given as the state-space generated from the system description. State space is the complete range of values held by all the state variables. A model checking algorithm is an automatic procedure that verifies that the property Phi expressed in temporal logic holds in the state space of the system M, that is, it gives an answer to the question M |= Phi.

A model checker takes a description of several concurrent, finite state machines as input and effectively analyzes the expanded computation tree for given properties. A computation tree is an abstract structure that consists of a possibly infinite set of all possible execution paths [9]. An example of a computation tree would be: following is a state machine of two concurrent processes P1 and P2 [9].

Figure 1: State Machine of two Concurrent Processes

The state space for this machine is the union of both these processes that is the Cartesian product which is represented by {(1,1), (1,2), (1,3), (2,2), (2,1), (2,3), (3,3), (3,1), (3,2)}. A computation tree with a start state of (2,2) is

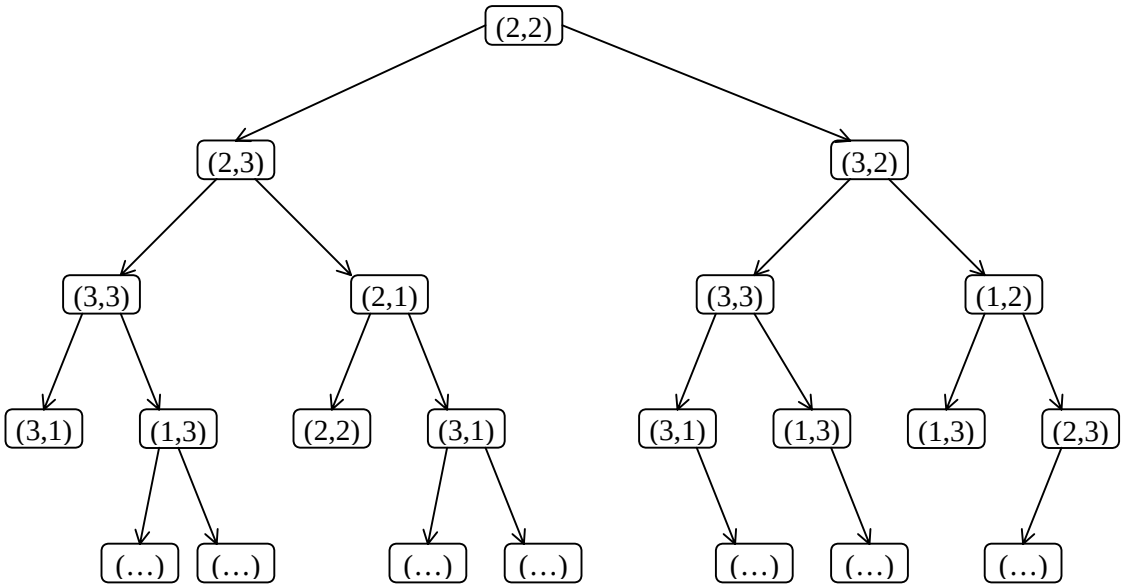

Figure 2: Computation Tree of two concurrent processes

The computation tree is from only one start state. We can have other trees from the remaining eight states from the state space. The tree contains infinitely many nodes, repeated many times. Model checkers will explore all the paths in a computation tree. As seen from the computation tree, the state space size is infinite and even for finite state systems the state space is extremely large and grows at an exponential rate. Searching such trees exhaustively can lead to the state space explosion problem. Model checkers use various methods to cope with this problem. Redundant states can be eliminated from searches due to memory-less properties of finite state machines, binary decision diagrams,

and partial order reduction are used to overcome this problem. On our part we can tighten our abstraction removing irrelevant states, model checking only independent sub-components in the system, and partitioning the system.

Model checking is based on formal logic. The computation tree is checked to see if certain properties are obeyed. A model checker can be used to determine whether or not the model contains paths that satisfy a specific property or requirement behavior. There are three categories of properties that can be checked in the model.

* Liveness property - some paths in the model should exhibit this property
* Invariance property - all paths in the model should exhibit this property
* Safety property - no paths in the model should exhibit this property

Safety and invariant properties are complements of each other and both require that all paths in the model be searched. Temporal logic is used to express the properties or requirement behaviors in model checking.

2.2.2 Linear Temporal Logic

Temporal logic, which was invented as a means of formalizing natural language statements about events in time, has proved to a powerful tool for specifying the behavior of reactive systems. It originated in philosophy as a branch of logic dealing with the topology of time, but for the past 20 years has found its' way into software verification. By using temporal logic, the properties of systems can be specified in a more behavior oriented fashion. Temporal logic describes state changes of a system by a truth value, which is given to each proposition in each state. The possible moves from state to state

are also specified. Hence the formulas of temporal logic include Boolean operators and temporal operators such as the formula should be true in “some future state” (eventually) denoted by ◆ or in “all future states” (always) denoted by [], among the many other temporal formulas. There are many variations on temporal logic such as Linear Temporal Logic (LTL), Branching Temporal Logic (BTL), Real-time Temporal Logic (RTTL), Computation Tree Logic (CTL). Linear Temporal Logic (LTL) views time as a sequence of states. The choice for the next state is either deterministic or non-existent. Linear Temporal Logic (LTL) adds two more additional operators until (U) and since (S). Given a model m and a temporal formula p we represent the notion that p holds at a position i >= 0 in m by

* (m, i) |= p
* (m, i) |= [] p is equivalent to, for all (k >= i), (m, k) |= p

  In other words [] p is true if and only if p is true now and in all the future states
* (m, i) |= ◆ p is equivalent to, for some (k >= i), (m, k) |= p

  In other words ◆ p is true if and only if p is true now or in some future state
* (m, i) |= p U q is equivalent to, for some (k >= i), (m, k) |= q and for every j such that i <= j < k, (m, j) |= p

  In other words q holds at some time in the future with p true until that time
* (m, i) |= p S q is equivalent to, for some k, 0<= k <= i, (m, k) |= p and for every j such that k < j <= i, (m, j) |= p

  In other words q has held true some time in the past since which time p has held true

The three properties of invariance, liveness and safety are represented as:

* Invariance – [] p
* Safety - ~◆ p
* Liveness – [] (p -> ◆ q)

2.2.3 Spin

Spin is a model checking tool among the many other model checking tools such as SMV, Murphi. It supports the design and verification of concurrent systems. Spin accepts design specifications in a verification modeling language called PROMELA (Process Meta Language) and requirements are modeled as correctness claims in Linear Temporal Logic (LTL). PROMELA hides the details that are related to process interactions and allows you to model the specific abstractions. Each statement in Spin is either executable or blocked [34]. A statement is executable if it satisfies its' conditions when the statement is encountered otherwise it is blocked. Skips, assignments and Boolean variables are always executable.

The specification of the concurrent system in PROMELA is done through one or more user-defined proctypes or process templates which interact through shared variables and message channels [7]. A process can be blocked in certain cases if it contains a statement which is blocked but this depends on the context of the statement. All statements and processes in Promela are executed randomly that is they are asynchronous unless otherwise specified. In Promela, a process can be instantiated either separately in an "init" process or by the "active" keyword. If there are a certain set of statements which should be executed without interruption they can be declared as an atomic sequence.

Atomic statements execute until any statement in the atomic block gets blocked. The variables can be local or global and the messages channels can be asynchronous or synchronous. For a more detailed look at Promela syntax and semantics look at [34].

Spin translates each process template into a finite automaton. The global state space automaton of the system is obtained by computation of an asynchronous interleaving product of automata such that there is one automaton per asynchronous process template. In Spin, the Linear Temporal Logic (LTL) correctness claims specify erroneous system behaviors that is behaviors that are undesirable to us. Spin translates the Linear Temporal Logic (LTL) correctness claims into Büchi automaton. A Büchi automaton is an automaton defined over infinite input sequences, rather than finite ones as in standard finite state machine theory [7]. The Büchi automaton is included in the Promela code as a "*never*" claim. This "*never*" claim is invoked after every state change during verification.

Spin performs the validation by computing the synchronous product of the Büchi automaton and the global state space automaton obtaining another Büchi automaton. If this Büchi automaton terminates or ends in a acceptance cycle then the Linear Temporal Logic (LTL) claim was violated that is the undesired behavior was present in the system. An acceptance cycle is a cycle where a state with an accepting label occurs infinitely often [7]. In this event, Spin will produce a "*trail*" file containing the counter-example which exhibits the path or execution where the undesired behavior was found. Spin has the capability of producing multiple "*trail*" files to show other paths where the undesired

behavior was found. If the Büchi automaton is empty it means that the original Linear Temporal Logic (LTL) correctness claim was not satisfied by the system or that the undesired behavior is not found in the system.

In simulation, Spin executes the model in three ways: (1) If random simulation is chosen, Spin will non-deterministically select one execution path if more than one path exists and executes the model. (2) If interactive simulation is chosen, the user can choose an execution path if more than one choice exists and executes the model. (3) If guided simulation is chosen, Spin will use the *"trail"* file to produce the execution of the counter-example. Since the *"trail"* file is extremely cryptic for us to read, guided simulation is chosen to produce output which is more readable and which shows the steps of the counter-example. In Callahan et al [16] recognized that the counter-examples are indeed test templates which can be used to create test oracles that drive and verify an actual test sequence on an implementation.

This thesis as mentioned before is about fault-injection through model checking via "naive assumptions". We would like to mention related work in fault-injection through model checking by Paul Ammann and Paul Black [8]. They have used mutation analysis to generate test cases from formal specifications in model checking. They define syntactic operators, each producing a slight variation on a given model. The operators represent mutation analysis on a model checking specification. They also recognize after [16] that counter-examples are indeed test cases. They define two classes of operators: (1) first, produce test cases from which a correct implementation must differ. (2)

second, produce test cases with which it must agree. In addition they have also proposed a new reduction method for model checkers for generating test sets also using a different soundness rule.

This chapter introduced all the related work and fundamentals in fault-injection, model checking, linear temporal logic and Spin. The next chapter will explain in detail about the methodology and a description of the case study.

Chapter 3

Description and Case Study

This chapter introduces the approach used for injecting faults through model checking via "naive assumptions" and its' application to the Sensor Failure Detection (SFD) system using Spin. First, it briefly depicts the approach and the "naive assumptions". Second, it describes the Sensor Failure Detection (SFD) system. Finally, the process undertaken in building the models in the model checker based on "naive assumptions" is illustrated.

3.1 Approach

Fault-injection means inserting faults into the program and observing the behavior of the program in response to the injected faults. Fault-injection can be used to study the effects of hardware and software faults. Faults can be injected into inputs, different points in the code and different locations in the state space. The results of the injected faults are seen in the output and different locations in the state space. Although fault and anomaly was stated in Chapter 2 on Page 7, it is worth stating it again. An anomaly is some event which if present in a software can change the behavior of the software in the future by corrupting some value in the software. A fault is a flaw in the program. A fault is a preanomaly event that, if executed and given some set of inputs, will corrupt a value in

the program. When one injects faults into a program through one of the many fault-injection methods, it also results in an injection of anomalies. By observing the program to study its' behavior, we are determining the anomalies. Fault-injection takes a global view of the impact of an anomaly.

Model checking, as described in Chapter 2 on Page 15, is used to verify and validate a system through process interactions. The power of model checking lies in its' exhaustive verification and generation of counter-examples if a property is violated. This research focuses on using the dynamic aspect of fault-injection together with the power of model checking. We are provided models of the fuel injection system in Simulink and the Sensor Failure Detection (SFD) system in Stateflow. The approach is applied to the fuel injection Sensor Failure Detection (SFD) system. Stateflow is a tool which is a variant of David Harel's statechart, which is described in detail in this chapter. The first step in our research is to specify the Sensor Failure Detection (SFD) system in a model checking tool Spin. Spin is explained in detail in Chapter 2 on Page 20. Fault-injection is used to determine the fault tolerance of the model checking models of the Sensor Failure Detection (SFD) system. During the specification of the Sensor Failure Detection (SFD) system in Spin, three fault-injection methods are used. The first two are based on "naive assumptions" and the third is done through inputs. We define "naive assumptions" as the various assumptions, which can be made about execution order, synchrony and fairness. We use model checking to examine interleaved execution of processes under various assumptions of execution order, synchrony and fairness. In other words, we make no a priori assumptions about execution order, synchrony and fairness.

The states in the Sensor Failure Detection (SFD) system model in Stateflow as shown in Figure 4 on Page 30 are specified as processes in Spin. We use "naive assumptions" of asynchronous and synchronous executions to inject faults which takes advantage of the loosely specified semantics of the Sensor Failure Detection (SFD) system combined with asynchronous process interactions of the model checker. In other words, we can specify the Sensor Failure Detection (SFD) system in a model checker as two separate models. One model will have the states as asynchronous processes and the second as synchronous processes. Indeed our models are "naive" with respect to any assumptions about synchrony. These "naive assumptions" are combination of code and state space fault-injection as explained in Chapter 2, Section 2.1.4 on Page 10. It is code based fault-injection as we are changing the behavior of the Sensor Failure Detection (SFD) system. We are also injecting faults in the state space as we are changing the flow of control in the model by synchronizing and asynchronizing the executions, which in turn modifies the state of the model between execution steps. The third fault-injection method is to change the inputs of the model as outlined in Chapter 2, Section 3.4 on Page 43. All of these three approaches are described in the rest of this chapter through the Sensor Failure Detection (SFD) system case study. Once the models have been specified, Linear Temporal Logic (LTL) properties are used to validate the behavior of the model and in turn discover the anomalies and errors.

3.2 Sensor Failure Detection System

3.2.1 Overview of Fuel Injection System

The model in Figure 3 on Page 28 represents a simple fault tolerant fuel injection system for throttle body fuel injection. The fault tolerant fuel injection system is a model provided to us in Matlab. Matlab is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. The control algorithms of the fault tolerant fuel injection system are implemented in Simulink block diagrams as seen in Figure 3 on Page 28. Simulink, a companion program to Matlab, is an interactive system for simulating nonlinear dynamic systems. The control logic for detecting and responding to sensor failures is implemented in a Stateflow diagram, which represents the Sensor Failure Detection (SFD) system. It is embedded as a block in the fuel injection system.

The controller is combined with a simple model for airflow and fuel mixing so that the entire feedback control system can be simulated. A sensor failure can be simulated by clicking on the series of switches to short-circuit individual sensors. The system is designed to use redundant sensor information to accommodate any single failure. When more than one sensor fails or when engine speed is excessive, the fuel system is disabled for safety reasons and it is enabled if the speed decreases or sensors recover. Figure 3 on Page 28 illustrates the Simulink diagram of the fuel injection system with the Sensor Failure Detection (SFD) system embedded in the center as the control logic unit.

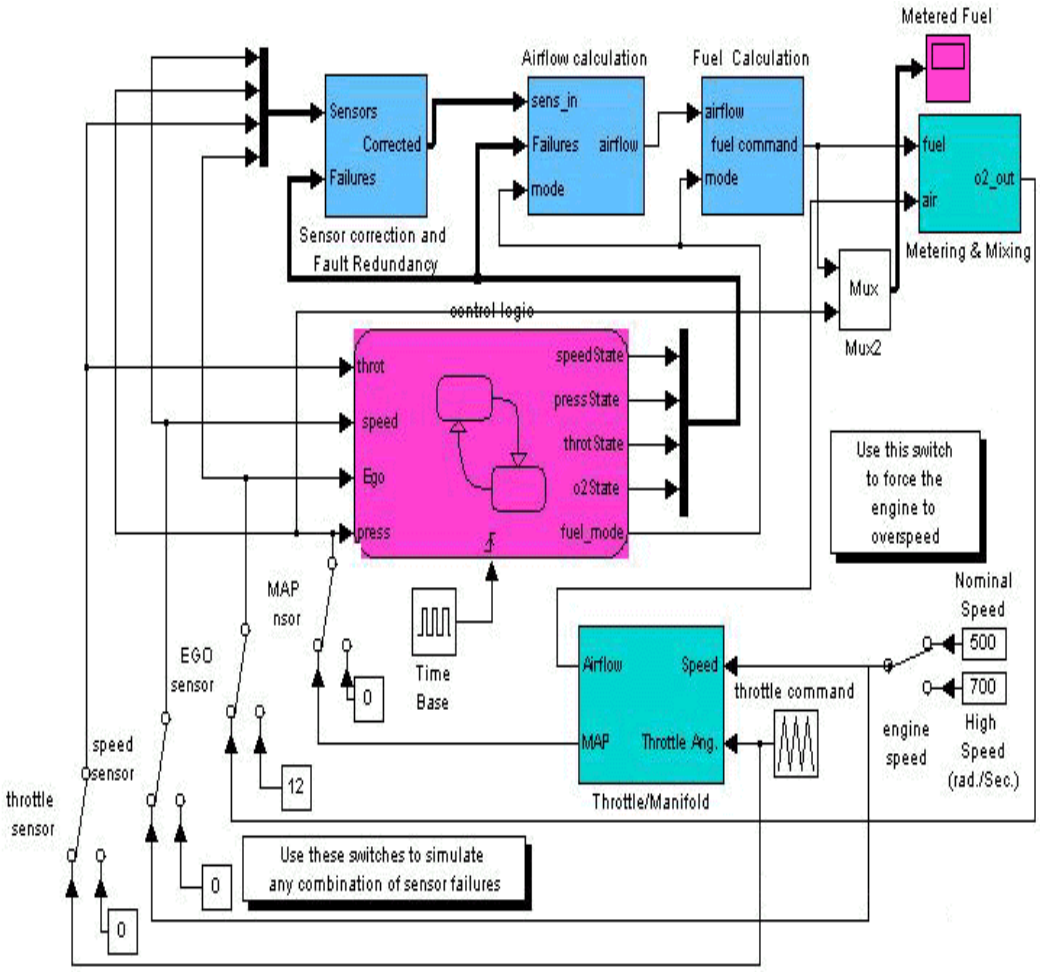

Figure 3: Simulink diagram of the Fuel Injection System

3.2.2 Overview of Stateflow

The Sensor Failure Detection (SFD) system is specified in Stateflow and provided to us. Stateflow uses a variant of the finite state machine notation established by David Harel. A finite state machine is a representation of a reactive system. In a reactive system, the system transitions from one state to another prescribed state, provided that the condition defining the change is true. A Stateflow diagram is a graphical representation of a finite state machine where states and transitions form the basic building blocks of the system.

Additionally, Stateflow enables the representation of hierarchy, parallelism, and history. Hierarchy enables the organization of complex systems by defining a parent/offspring object structure. A system with parallelism can have two or more orthogonal states active at the same time. History provides the means to specify the destination state of a transition based on historical information. Figure 4 on Page 30 illustrates the Stateflow diagram of the Sensor Failure Detection (SFD) system that is the embedded control logic unit for the fuel injection system.

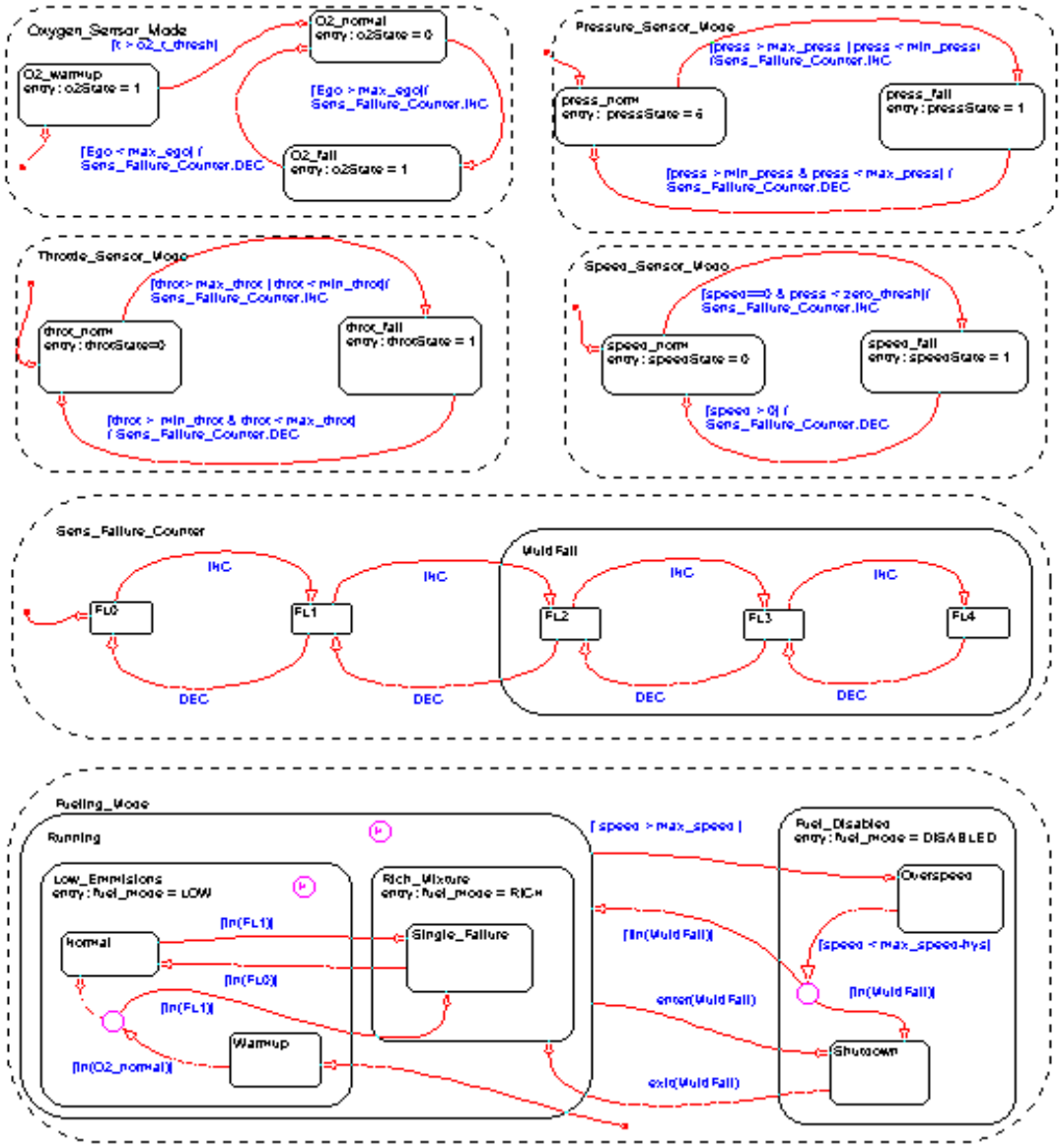

Figure 4: Stateflow diagram of the Sensor Failure Detection system

3.2.3 Description of Sensor Failure Detection System

Our research is concerned with anomalies and the errors present behind the anomalies in the Sensor Failure Detection (SFD) system. The Sensor Failure Detection (SFD) system has four sensors as shown in Figure 4, the Oxygen, Pressure, Throttle and Speed sensors depicted by the first four states. The failures and recoveries for these sensors are input

from the fuel injection model in Simulink by the user clicking on the switches to simulate the sensor failures and recoveries. The engine speed is simulated from the Simulink model. The timed event "t" is also generated by the Simulink model, which is a countdown to the engine exceeding the oxygen threshold. The failure counter state keeps track of the number of failed and recovered sensors. The fueling state has two purposes, if there is more than one failure or engine speed is excessive, the fueling injection system is disabled or if there is no failure or one failure or it is in warmup, the fuel injection system is enabled.

3.2.4 Semantics of the Sensor Failure Detection System

The activity or inactivity of the states changes dynamically based on events and conditions. The dashed boxes represent AND, in other words, parallel states and the solid boxes represent OR, in other words, mutually exclusive states. Each state has a default value which is represented by the dangling arrow. Transitions have a condition, which specifies that a transition occurs given that the specified condition is true. For example: *o2_normal* transitions to *o2_fail* only if *ego* > *max_ego*. Transitions also have a transition action, which is executed if the transition destination is determined to be valid. For example: if *ego* > *max_ego* and if *o2_fail* is determined to be a valid state then *Sens_Failure_Counter.INC* is executed immediately and *o2_normal* transitions to *o2_fail*.

A connective junction is used to represent a decision point, it is like an "if" statement. For example: in the fueling state if *in(o2_normal)* is true in other words

oxygen sensor state is in *o2_normal, warmup* mode transitions to a connective junction where a decision is made whether to go to *normal* mode or *single_failure* mode. A history junction is used to represent a historical decision point. A circle and a "h" inside the circle represents a history junction. A transition into a history junction will take it to the last visited state. States and transitions are organized in a hierarchy. For example: in the fueling state, the *running* mode is the parent of the *low_emmissions* mode and *rich_mixture* mode. A transitions' hierarchy is described in terms of the transitions' parent, source and destination. Hence all transitions out of the *running* sub-state in the fueling state is evaluated before any transitions inside *low_emmissions* mode and *rich_mixture* mode are evaluated.

3.2.5 Typical Run of Sensor Failure Detection System

At initialization, all the states enter their default state for example: *oxygen_sensor_mode* is initialized to *o2_warmup, pressure_sensor_mode* is initialized to *press_norm* and so on. After $t > o2\_t\_thresh$, *o2_warmup* transitions to *o2_normal*. This triggers fueling state to transition from *warmup* mode to either *normal* mode or *single_failure* mode depending on how many sensors have failed. If any sensors fail or recover, the appropriate transitions are made in the sensor states and the fueling state. In the same way, if engine speed decreases or increases, the changes are made in fueling state.

3.3 Application of Approach

3.3.1 Abstraction Process

Any abstraction must not completely deviate from the original hence it must preserve some properties of the original. According to Amman and Black [8], abstractions can be measured by soundness and completeness. Soundness in an abstraction measures if properties in an abstraction are also present in the original. Completeness in an abstraction measures if the properties in the original are also present in the abstraction.

The first step in the construction of the models in Spin based on our "naive assumptions", is abstraction of our model. This means separating from the fuel injection Sensor Failure Detection (SFD) system those aspects of the design that are directly relevant to the properties we are interested in proving correct. In this research, the Stateflow diagram of the Sensor Failure Detection (SFD) system is the only part of the fuel injection system being validated. Hence the Simulink model of the fuel injection system is not modeled in Spin. This takes away all the complex and mathematical functions present in the fuel injection system. But there are three things from the Simulink model, which needs to be abstracted. They are the user simulation of sensor failures and recoveries, the change of engine speed both of which are a series of switches, and the timed event *t*. The timed event *t* is a countdown for oxygen state exceeding oxygen threshold.

Each of the four sensor states was modeled as a separated process or proctype in Spin. The *Sens_Failure_Counter* and fueling state was also modeled as separate

proctypes. The failure and recovery of the four sensors and the change of engine speed was handled randomly such that if any sensor randomly failed or recovered, it could choose to go to recovered or failed respectively. Also if engine speed randomly increased it could choose to decrease and vice versa. In the fueling state proctype, the sub-states of *running* mode and *disabled* mode was modeled as two separate labels such that each one could go to the other label when the label is called in a goto statement. The timed event *t* was also modeled as a separate process such that a variable *t* was randomly increased by a constant value each time.

Spin derives its' global state space by computing the product of all the asynchronous automata. Our first abstraction of the Sensor Failure Detection (SFD) system resulted in a huge state space explosion problem. There are a number of ways to reduce the state space, some of which were used to contain our state space. The first step taken was to see if any variables types could be changed to reduce the memory consumption, such as certain variables which were bytes were changed to bit type and so on. Next step was to define equivalence classes. This was done by identifying subsets in the Sensor Failure Detection (SFD) system which were mutually disjoint such that the union of all these sets was the original set. This led us to identifying five such subsets that is the four sensors and the process depicting the change in engine speed. We also noted that the separate process for the *Sensor_Failure_Counter* was not needed and could be embedded in each of the four sensor proctypes but the fueling state proctype was indeed needed. Hence now instead of eight processes we only have seven processes.

The third step was to see if any of the processes could be modified. What this means is to see if anything else could be abstracted out of the processes to further simplify them? The process, which was illustrating the timed event t, was increasing a variable *t* randomly by a constant. This variable *t* was only needed to evaluate the condition $t > o2\_t\_thresh$ for the oxygen sensor state. So in other words after this condition was evaluated, *t* was no longer needed. Hence the proctype for this timed event *t* was modified to only increase *t* for this condition and after this condition was evaluated this proctype was no longer evaluated. After this reduction, the state space was greatly contained and the explosion problem was overcome.

3.3.2 Asynchronous Model

The crux of this research is fault-injection through model checking via "naive assumptions". The "naive assumptions" are asynchronous and synchronous executions. Hence the first step is to specify the Sensor Failure Detection (SFD) system as an asynchronous model using the Spin tool. Since Spin is used to verify asynchronous processes, the specification of the Sensor Failure Detection (SFD) system is simplified.

Each of the sensors is modeled as asynchronous processes. All the transitions are conditions implemented in an 'if' or a 'do' loop. The simulation of the sensor failures and recoveries is done randomly such that based on the current state of a sensor, the sensor can choose to go to either failed or recovered. Each time a sensor fails or recovers the sensor failure counter is immediately either incremented or decremented respectively and the sensor state is changed respectively. The initial state of all the sensors is set to

normal. At startup, each sensor state is changed based on the value of the sensors. For example: if the *speed_sensor_mode* is *fail* at startup, the state of the *speed_sensor_mode* is set to *speed_fail*. Once the model goes through startup, the sensor failures and recoveries are simulated randomly. The only difference is the oxygen sensor state, which doesn't progress until the condition for *t* is evaluated. The timed event *t* is also specified as a separate asynchronous proctype. The variable *t* is increased by a constant until it goes beyond the oxygen threshold. Figure 5 on Page 36 and Figure 6 on Page 37 illustrates the specification of the oxygen sensor and the timed event *t* respectively.

```
active proctype O2_Sensor_Mode() {


     if
     :: atomic{t > o2_t_thresh -> o2_mode = o2_normal;}
     fi;

     if
     :: atomic{(o2_sensor == fail) && (t > o2_t_thresh) ->
          o2_mode = o2_fail; o2_sensor = 2; fail_cnt = fail_cnt + 1;}
     :: atomic{(o2_sensor == norm) && (t > o2_t_thresh)-> o2_sensor = 2;}
     fi;

     do
     :: (o2_mode == o2_normal) ->
           atomic{o2_mode = o2_fail; fail_cnt = fail_cnt + 1;}
     :: (o2_mode == o2_fail) ->
          atomic{o2_mode = o2_normal; fail_cnt = fail_cnt - 1;}
     od;
}
```

Figure 5: Oxygen Sensor Process

```
active proctype Time_Counter(){

      do
      :: atomic{(t <= o2_t_thresh) -> t = 3;}
      od;

}
```

Figure 6: Timed Event *t* Process

The engine speed is translated as a separate asynchronous proctype wherein *speed* can either decrease or increase based on its' current speed. The fueling state is also specified as a separate asynchronous proctype. The parent and child hierarchy within the *running* mode of fueling state is handled by two inner loops. Two separate labels separate both the sub-states of *running* mode and *disabled* mode. Figure 7 on Page 38 partly illustrates the fueling state proctype. For a complete illustration of the asynchronous model, please see Appendix A.

```
active proctype Fueling_Mode(){

Running:   do
     :: atomic{((fueling_mode == running)  && (speed >
          MAX_SPEED)) -> fueling_mode = disabled; fueling_state
          = overspeed; goto Disabled;}

     :: atomic{((fueling_mode == running)  && (fail_cnt > 1))
          -> fueling_mode = disabled; fueling_state = shutdown;
          goto Disabled;}

     :: ((fueling_mode == running)  && (speed < MAX_SPEED) &&
          (fail_cnt <= 1)) ->

          do
          ----------------------
          od;
     od;

Disabled:  do
           ----------------------
           od;
}
```

Figure 7: Fueling State Process

This completes the specification of the asynchronous model. In order to find the faults that have been injected into the system, the behavior of this model will be validated using Linear Temporal Logic (LTL) properties in Chapter 4.

3.3.3 Synchronous Model

The second model is specified based on the "naive assumption" of synchronous execution. The approach is to synchronize the Sensor Failure Detection (SFD) system model in Spin. Hence the Sensor Failure Detection (SFD) system is synchronized left to right, top to bottom as depicted in Figure 4 on Page 30. The order of the synchronized proctypes are the timed event *t* followed by the oxygen sensor state, pressure sensor state, throttle sensor state, speed sensor state and the fueling state.

The sensor failures and recoveries are simulated randomly that is depending on the current state of a sensor, the sensor can either fail or recover. The significant difference between this and the asynchronous model is once a sensor has been changed, it can't change back again until the respective sensor state and the fueling state indicates the changed state. This is done by a turn taking variable, which indicates which sensor has changed. For example: if the *throttle_sensor* has been set to *fail* from *norm*, the *throt_syn* will be set to *change* such that the *throttle_sensor_mode* has to be set to *throt_fail* and fueling state has to indicate this failure, then *throt_syn* will be set to *nochange*. The *throt_syn* is the turn taking variable which prevents *throttle_sensor* from being reset until *throttle_sensor_mode* has shown the effects of the failed *throttle_sensor* that is the *throttle_ sensor_mode* has to go to failure and the fueling state has to show the effect of the fact that there is at least one sensor which is failed. After this the *throttle_sensor* can again change randomly. Hence sensor failures are synchronized to a certain extent. Figure 8 on Page 40 illustrates the proctype for the sensor failures and recoveries.

```
active proctype Change_sensor_Values(){

      do
      :: atomic{(o2_sensor == norm) && (o2_syn == nochange) ->
           o2_sensor = fail; o2_syn = change;}
      :: atomic{(o2_sensor == fail) && (o2_syn == nochange) ->
           o2_sensor = norm; o2_syn = change;}
      :: atomic{(press_sensor == norm) && (press_syn == nochange) ->
           press_sensor = fail; press_syn = change;}
      :: atomic{(press_sensor == fail) && (press_syn == nochange) ->
           press_sensor = norm; press_syn = change;}
      :: atomic{(throt_sensor == norm) && (throt_syn == nochange) ->
           throt_sensor = fail; throt_syn = change;}
      :: atomic{(throt_sensor == fail) && (throt_syn == nochange) ->
           throt_sensor = norm; throt_syn = change;}
      :: atomic{(speed_sensor == norm) && (speed_syn == nochange) ->
           speed_sensor = fail; speed_syn = change;}
      :: atomic{(speed_sensor == fail) && (speed_syn == nochange) ->
           speed_sensor = norm; speed_syn = change;}
      od;

}
```

Figure 8: Change Sensors Values Process

The timed event *t* proctype is the first among the synchronized processes. The variable *t* is incremented by a constant each time the proctype is executable. The variable *synchronize* handles the turn taking for the synchronized proctypes. Once the variable *t* is beyond the value of *o2_t_thresh*, the timed event *t* proctype no longer increments *t* but just sets the *synchronize* variable to execute the oxygen proctype. Figure 9 illustrates the timed event *t* proctype.

```
active proctype Time_Counter(){

      do
      :: atomic{(t <= o2_t_thresh) && (synchronize == 0) ->
           t = t + 1; synchronize = 1;}
      :: atomic{(t > o2_t_thresh) && (synchronize == 0) ->
           synchronize = 1;}
      od;

}
```

Figure 9: Timed Event *t* Process

Each of the sensors is modeled as synchronous processes. Each time a sensor fails or recovers the sensor *fail_cnt* is immediately either incremented or decremented respectively and the respective sensor state is changed. The initial state of all the sensor is set to normal. But at startup each sensor state checks for the initial value of the sensors. For example: if the *speed_sensor* is *fail* at startup the state of the *speed_sensor_mode* is set to *speed_fail*. The only slight difference is the oxygen sensor that doesn't progress until the condition for *t* is evaluated. Figure 10 illustrates the *oxygen_sensor_mode*.

```
active proctype O2_Sensor_Mode() {

     do
     :: atomic{ (t > o2_t_thresh) && (synchronize == 1) ->
          o2_mode = o2_normal; synchronize = 2; fueling_syn = change; break;}
     :: atomic{ (t <= o2_t_thresh) && (synchronize == 1) -> synchronize = 2;}
     od;

     if
     :: atomic{(synchronize == 1) && (o2_sensor == fail) -> o2_mode = o2_fail;
          synchronize = 2; o2_syn = nochange; fueling_syn = change;}
     :: atomic{(synchronize == 1) && (o2_sensor == norm) -> synchronize = 2;
          o2_syn = nochange;}
     fi;

     do
     :: atomic{(o2_sensor == fail) && (synchronize == 1) && (o2_syn == change)
           -> o2_mode = o2_fail; fail_cnt = fail_cnt + 1; synchronize = 2;
          o2_syn = nochange; fueling_syn = change;}
     :: atomic{(o2_sensor == norm) && (synchronize == 1) && (o2_syn == change)
           -> o2_mode = o2_normal; fail_cnt = fail_cnt - 1; synchronize = 2;
          o2_syn = nochange; fueling_syn = change;}
     :: atomic{(o2_syn == nochange) && (synchronize == 1) -> synchronize = 2;}
     od;
}
```

Figure 10: Oxygen Sensor Process

The fueling state proctype is the last synchronized process. This process is exactly the same at the asynchronous model. The only difference is that it is

synchronized and is only executable when it is its' turn. Also if there has been no change with engine speed or any sensors the turn is simply passed on to the timed event *t* proctype as there is no change in the fueling state. For a complete listing of fueling state proctype and a complete source code of the synchronous model, look at Appendix B. The engine speed is also increased and decreased randomly. The only difference is when engine speed is changed it cannot change back again until fueling state proctype shows the effects of the changed engine speed. For example: when *speed* is increased, it cannot be decreased immediately until fueling state is set to *overspeed* mode which allows the engine *speed* to change randomly again. Figure 11 illustrates the engine speed proctype.

```
active proctype Change_speed_value(){

      do
      :: atomic{(speed == 5) && (slow == change) -> speed = 35;
           fast = nochange; fueling_syn = change;}

      :: atomic{(speed == 35) && (fast == change) -> speed = 5;
           slow = nochange; fueling_syn = change;}

      od;

}
```

Figure 11: Engine Speed Process

3.4 Input Fault-Injection

The third and final fault-injection method is based on the input space of a program. This fault-injection method is applied on the asynchronous and synchronous Sensor Failure Detection (SFD) system models. Input space constitutes all the inputs of a program. It makes sense to only work with legal inputs that is inputs that come from the domain space of a program. The reason is the idea behind fault-injection on inputs is to corrupt the legal inputs and see what happens. Inputs can come from various sources such as external, internal events, sensors, and randomly generated inputs. In the Sensor Failure Detection (SFD) system, the inputs come from sensors and engine speed. There are five inputs in the Sensor Failure Detection (SFD) system which are the oxygen, throttle, pressure, speed sensors and the engine speed. Since all the inputs are randomly simulated in the models we don't need to inject faults into the input during the execution of the program as Spin will look at all possible values of the inputs during program execution. The only case, which needs to be considered, is initial or startup values of the inputs. Hence different values of the inputs are injected for the engine speed and sensors into both our models during startup and the behavior of the Sensor Failure Detection (SFD) system models are studied with all these initial values. For example: in one case at startup, sensors are normal but engine *speed* exceeds *max_speed* and the behavior of the Sensor Failure Detection (SFD) system is observed to see its' response to the increased engine *speed*. The method is illustrated on the Sensor Failure Detection (SFD) system in the next chapter.

In this chapter, we described the fuel injection Sensor Failure Detection (SFD) system. We also outlined our abstraction process and how we overcame the state space explosion problem. We then went on to apply our fault-injection techniques to the Sensor Failure Detection (SFD) system and specified two models in the Spin tool. The next chapter will validate the two Sensor Failure Detection (SFD) system models using Linear Temporal Logic (LTL) properties and explain the results of the validation.

Chapter 4

Analysis, Results, and Discussion

In the previous chapter, we specified two separate state models of the Sensor Failure Detection (SFD) system in Promela based on two "naive assumptions". This chapter describes the Linear Temporal Logic (LTL) properties applied to both our models in order to discover the anomalies. First, the classification of the results from the properties are described. Next, the results for each fault class are interpreted.

4.1 Counter-Examples

In the previous chapter, we explained and demonstrated three ways to inject faults, through "naive assumptions" of asynchronous and synchronous executions and through inputs. The Sensor Failure Detection (SFD) system is an existing model in Stateflow and was provided to us. We applied "naive assumptions" of asynchronous and synchronous executions to specify two models of the Sensor Failure Detection (SFD) system in the model checking tool Spin. Both our models are indeed "naive" with respect to any assumptions of synchrony. These assumptions also led to determinism and nondeterminism in our models. We also injected faults through the inputs in both these models using our third method of injecting faults explained in Section 3.4 on Page 43. In

order to discover the anomalies in the models as the result of our three fault-injection methods, Linear Temporal Logic (LTL) correctness claims are used. These Linear Temporal Logic (LTL) claims represent undesirable behaviors in the Sensor Failure Detection (SFD) system. Together with the state model and a Linear Temporal Logic (LTL) correctness claim or property, the Spin tool will automatically produce a validator or a "*never*" clause. This "*never*" clause is a Büchi automaton. When a validator or Büchi automaton is executed, Spin will either produce a "*trail*" file if a "*never*" claim is violated that is the undesirable behavior is found to exist in our system. A "*trail*" file by itself is cryptic hence it is run by guided simulation to produce a counter-example, which shows that an undesirable behavior is present in the model through a sequence of events. Counter-examples produced by Spin constitute the outputs of the Linear Temporal Logic (LTL) properties. The counter-examples represent test templates that can serve as test cases as they contain the inputs and the resulting outputs. These test cases enable us to see the related requirements that have been violated in our models. By observing the test cases we can detect errors in our specification, which enable us to correct the errors. Thus the test cases help us maintain fidelity between the specification and our models. The counter-examples also serve as the outputs for fault-injection. The generation of the outputs or test cases is automatic and built into the Spin tool. The test cases or outputs can then be studied by going through the sequence of events in the "*trail*" file. The approach used to go through the counter-examples is to step through each state change recorded in them.

The Spin tool has the capability to generate more than one *"trail"* file. We can also set the search depth in Spin that is basically how deep in the computation tree we want to go in order to find the anomaly. In some cases, we might only need to go to a very short depth and the anomaly is visible in the sequence of events but in other cases we might need to go to a greater depth in the computation tree. We may determine that the anomaly is present at a certain depth, but there may be other anomalies present at deeper depths of the computation tree if we go further but they are masked by the previous anomaly. In order to get to the deeper anomalies in the computation tree, we can do one or a combination of these three steps: First, we can correct the previous anomalies; Second we can change the Linear Temporal Logic (LTL) property to make it more specific; Finally, we can generate muliple *"trail"* files and execute the *"trail"* files through guided simulation to produce counter-examples to see if different anomalies are present. The approach we follow to get to the deeper anomalies depends upon the kind of anomalies detected. Depending upon the anomalies detected we might find it may be easier to correct the previous anomalies or change the property or it may be best to generate multiple *"trail"* files.

4.2 Property Classification

We base the requirements of the Sensor Failure Detection (SFD) system on the functionality gathered from the semantics and transition conditions in the Sensor Failure Detection (SFD) system. The Linear Temporal Logic (LTL) claims or properties are formulated to validate the requirements of the Sensor Failure Detection (SFD) system. In all, we validate thirty Linear Temporal Logic (LTL) properties on both our models. Most

of these thirty properties fail in the verification that is they generate counter-examples indicating the presence of undesirable behavior in our models. We decided the best way to interpret and observe the results of these properties are to classify only those properties, which fail in the verification into different categories. Our approach is to classify the outputs or test cases from the Linear Temporal Logic (LTL) properties generated by the Spin tool. By classifying the outputs or test cases we in turn classify the properties and the related requirements. In order to arrive at the classifications, we observe the outputs through a series of questions:

* Do the outputs represent an errant anomaly? (that violates a desired property)
* What are the sequences of events that represent this anomaly?
* Which fault-injection method results in the simulation of this anomaly?
* What error causes this anomaly?
* Is there a general category of anomalies? (i.e., a common pattern among the sequence of events that represents the behavior of the program)
* What are the requirements behind each Linear Temporal Logic (LTL) property?
* Do the Linear Temporal Logic (LTL) properties represent liveness, invariance or safety properties?

Based on the answers to the above questions, we found five fault classes that categorized all the outputs of the Linear Temporal Logic (LTL) properties. These five fault classes are:

* Race Condition

* Ambiguity
* Missing Requirement
* Inconsistent Requirement
* Loss of Failures and Recoveries

We call the above classifications as fault classes as we are looking at the outputs that contain trace sequences of events that lead to the anomalous behavior and also determining the reason behind the anomaly that is the error. In each fault class, there are related requirements of the Sensor Failure Detection (SFD) system based on the functionality gathered from the semantics and transition conditions of the Sensor Failure Detection (SFD) system. By observing the error behind each anomaly we see the requirements that have been violated. We also state Linear Temporal Logic (LTL) properties in each fault class used to validate each of these requirements. Thus by classifying the outputs indirectly classified the properties and their related requirements. Even though there are many requirements and Linear Temporal Logic (LTL) properties in each fault class, all the outputs from the properties exhibit the same anomaly represented by that fault class. For example: all the properties in the race condition fault class seen in Table 1 on Page 53-54 generate outputs, which have the race condition anomaly. The sequence of events in the outputs may be different but eventually all the outputs contain the race condition anomaly.

In certain fault classes, there are properties, which generate outputs that exhibit two different anomalies. For example: a property in the Missing Requirement fault class

and Loss of Failures and Recoveries fault class is the same but upon observing the outputs, we see two different anomalies hence the property is present in both the fault classes. Certain fault classes contain anomalies, which are the results of the faults injected through either one or all, or a combination of the three fault-injection methods explained in the previous chapter. In other words, certain fault classes exhibit anomalies only in the asynchronous model or some cases in only the synchronous model and in some cases they are present in both models. As stated before the third fault-injection method of injecting faults into the inputs as explained in Section 3.4 on Page 43 applies to both our models. Even though both the asynchronous and synchronous models are different they intersect on the anomalies, as some anomalies are present in both our models. In the next few sections, all the properties in each of the fault classes is explained in detail. Each fault class is followed by a detailed explanation of the results and their interpretation. Since the outputs or *"trail"* files can be very large, it was decided to include just one counter-example for all the fault classes in Appendix C to show an example of a Spin guided simulation of a *"trail"* file.

4.3 Fault Classes

4.3.1 Race Condition

In normal terms, race condition takes place at two or more receive events, where the order of the incoming messages is arbitrary. The existence of race condition arises from nondeterminism in the program flow and leads to unpredictable interleaving of threads within a program. Upon observing the outputs generated by the properties in Table 1 on Page 53-54, we conclude that all the outputs exhibit the race condition anomaly. The

properties in Table 1 characterize potential failures resulting from violations of liveness and invariance properties. In order to understand the reason behind the race condition, we need to remember that all of the properties in Table 1 are validated on the asynchronous model. Hence the race condition anomaly is due to the asynchronous "naive assumptions".

All the related requirements, R1-R4 in this fault class signify the functionality of the Sensor Failure Detection (SFD) system that we have gathered from the semantics and transition conditions of the Sensor Failure Detection (SFD) system.

R1: If there are sensor failures and recoveries, there should be a transition eventually to the appropriate sub-states in the fueling state. Hence there are sub-states in fueling state, which should not get starved.

R2: If the fueling state has a certain sub-state as its' current state then the condition for being in that sub-state should always be true. For example: if *fueling_state* is in *single_failure* mode then there should be one sensor failure.

R3: If eventually the timed event *t* does exceed *o2_t_thresh* then eventually the oxygen sensor state should go to *o2_normal*. Hence timed event *t* and oxygen sensor state should not get starved.

R4: If either one or all or a combination of the sensors are failed and/or engine has exceeded the *speed* at the startup or initial state of the system then eventually the system must indicate the sensors failed or the engine overspeed.

In order to determine if the above requirements (R1-R4) are present in our models, we formulate Linear Temporal Logic (LTL) properties as shown in Table 1 on Page 53-54. We explain which properties in Table 1 cover which requirements, R1-R4. We expound one property in Table 1 for each requirement, R1-R4. Properties 1 through 5 cover R1 in Table 1. For example: P1 (property 1) states [](p -> <>q) where p is the occurrence of a multiple sensor failures and q is the indication of it. [] and ◆ are temporal logic operators which mean 'always' and 'eventually' respectively. The term "[]x" means in all future states x is true and the term "◆x" means that at some future state x is true. In the Spin tool, the symbol for 'eventually' it is <> instead of ◆. The formula in P1 expresses the condition that in all future states if *fail_cnt* exceeds one then in some future state *fueling_state* must go to *shutdown* mode. Properties 6 through 8 cover R2. For example: The formula in P6 states that in all future states if *fueling_state* is in *shutdown* mode then *fail_cnt* exceeds one for that state. Property 9 covers R3. For example: The formula in P9 states that taking into account all future states if in some future state *t* exceeds *o2_t_thresh* then in some future state oxygen sensor state or *o2_mode* must progress to *o2_normal*. Properties 10 through 13 cover R4. For example: The formula in P10 states that if at the initial state *o2_mode* has failed then in some future state *fail_cnt* must be at one to indicate the failure.

Property No	*Table 1: Linear Temporal Logic Properties*
P1	Formula: [](p -> <> q) Symbol: #define p (fail_cnt > 1) #define q (fueling_state == shutdown)
P2	Formula: [](p -> <> q) Symbol: #define p (speed > MAX_SPEED) #define q (fueling_state == overspeed)
P3	Formula: [](p -> <> q) Symbol: #define p (fueling_state == shutdown) && (fail_cnt <= 1) #define q (fueling_state == single_failure)
P4	Formula: [](p -> <> q) Symbol: #define p (fueling_state == normal) && (fail_cnt == 1) && (speed < MAX_SPEED) #define q (fueling_state == single_failure)
P5	Formula: [](p -> <> q) Symbol: #define p (fueling_state == single_failure) && (fail_cnt == 0) && (speed < MAX_SPEED) #define q (fueling_state == normal)
P6	Formula: [] (p -> q) Symbol: #define p (fueling_state == shutdown) #define q (fail_cnt > 1)
P7	Formula: [] (p -> q) Symbol: #define p (fueling_state == single_failure) #define q (fail_cnt == 1)
P8	Formula: [] (p -> q) Symbol: #define p (fueling_state == overspeed) #define q (speed > MAX_SPEED)
P9	Formula: [](<> p -> <> q) Symbol: #define p (t > o2_t_thresh) #define q (o2_mode == o2_normal)
P10	Formula: p -> <> q Symbol: #define p (o2_sensor == fail) && (press_sensor == norm) && (throt_sensor == norm) && (speed_sensor == norm) #define q (fail_cnt == 1)
P11	Formula: p -> <>q Symbol: #define p (speed_sensor == fail) && (throt_sensor == fail) && (o2_sensor == norm) && (press_sensor == norm) #define q (fail_cnt == 2) && (fueling_state == shutdown)
P12	Formula: p -> <> q Symbol: #define p (speed_sensor == fail) && (throt_sensor == fail) && (o2_sensor == norm) && (press_sensor == norm) #define q (fail_cnt == 2)

P13	Formula: p -> <> q Symbol: #define p (o2_sensor == norm) && (press_sensor == norm) && (throt_sensor == norm) && (speed_sensor == norm) && (speed == 35) #define q (fueling_state == overspeed)

Table 1: Race Condition

4.3.1.1 Results

We have explained the requirements based on the Sensor Failure Detection (SFD) system and Linear Temporal Logic (LTL) properties formulated to validate the requirements. When the properties in Table 1 failed during verification that is the Spin tool generates outputs or counter-examples, which indicates that the properties and in turn the requirements are violated in our model. By observing the trace sequence of events in the outputs, we conclude that all the outputs have the race condition anomaly. The sequence of events that exhibit this anomaly is the sensors are getting set back and forth without letting other processes execute. For example: In the output of the first property in Table 1 on Page 53, even though *fail_cnt* exceeds one, *fueling_state* is not able to go to *shutdown* mode as the sensors are randomly getting set back and forth due to which *shutdown* mode is getting starved out. This randomness of the sensors is present in all of the properties due to which even though a certain condition is true the transition to the destination state is not occurring. As stated before, the race condition anomaly is only present in our asynchronous model and through the faults injected in the inputs. This race condition was also preventing us from observing other anomalies, which maybe present in the Sensor Failure Detection (SFD) system. In order to observe other anomalies, we can

correct the race condition anomaly by synchronizing our model. Hence by specifying a model of the Sensor Failure Detection (SFD) system through assumption of complete asynchrony led to the revelation of potential race faults.

4.3.2 Ambiguity

This fault class is termed 'ambiguity' as the anomaly exhibited in this fault class based on observing the outputs from the properties in Table 2 on Page 56 is due to an ambiguous requirement in the Sensor Failure Detection (SFD) system. An ambiguous requirement is a requirement which creates doubtfulness or uncertainty in its' interpretation. In the Stateflow diagram of the Sensor Failure Detection (SFD) system Figure 4 on Page 30, there are two conditions under which fueling state exits from the *running* sub-state to *disabled* sub-state. But there is no clear condition stating, which one takes precedence if both conditions are true at the same time. R1 states the requirement, which we formulated based on the absence of a precedence condition.

R1: There is no precedence between the two transition conditions exiting *running* sub-state and entering the *disabled* sub-state.

In order to determine if the above requirement R1 is present in our models, we formulate Linear Temporal Logic (LTL) properties as shown in Table 2 on Page 56. Even though there is only requirement R1 we have three properties in Table 2 as the first property in Table 2 is validated on the asynchronous model but the next two are validated on the synchronous model. We expound one property in Table 2. For example: P2 states

that for all future states if *fail_cnt* exceeds one for any future state then eventually *fueling_state* must be equal to *shutdown* mode for some future state.

Property No	*Table 2: Linear Temporal Logic Properties*
P1	Formula: ([] <> p) -> <> q Symbol: #define p (speed > MAX_SPEED) && (fail_cnt > 1) #define (fueling_state == overspeed)
P2	Formula: [] (p -> <> q) Symbol: #define p (fail_cnt > 1) #define q (fueling_state == shutdown)
P3	Formula: [] (p -> <> q) Symbol: #define p (speed > MAX_SPEED) #define q (fueling_state == overspeed)

Table 2: Ambiguity

4.3.2.1 Results

As mentioned before there are two transitions exiting the *running* sub-state in fueling state, one is for *speed* > *max_speed* and the other for *enter(MultiFail)* which means if *fail_cnt* exceeds one. According to the Stateflow diagram there is no clear statement or condition, stating which one of these transitions should be taken if both conditions are true at the same time, thus there is an inherent ambiguity. We have explained the requirement based on absence of this precedence condition in the Sensor Failure Detection (SFD) system and Linear Temporal Logic (LTL) properties formulated in Table 2 to validate the requirement. When the properties in Table 2 are validated, the Spin tool generates outputs or counter-examples which indicates that the properties and in turn the requirement R1 is violated in our models. By tracing the sequence of events in the

outputs, we observe that the requirement R1 is not violated but rather the ambiguity in the R1 creates an undesirable behavior in our models. Due to this ambiguity, whenever both conditions are true the transition to *overspeed* mode is always taken and the transition to *shutdown* mode is never taken. Thus *shutdown* mode is always getting starved out and there is no fairness.

Our definition of "naive assumptions" is the various assumptions that can be made about execution order, synchrony and fairness. Based on our "naive assumptions" we constructed two models of the Sensor Failure Detection (SFD) system. Both our models are naive with respect to any fairness. Hence this anomaly due to the ambiguous requirement is present in both the asynchronous and synchronous fault-injection models. This assumption of fairness led to the discovery of the anomalous behavior due to the ambiguity in requirement, R1. Even though the anomaly is the same upon observing the outputs from the properties in Table 2, different properties are used to discover the anomaly in our two models. The first property generates a counter-example on the asynchronous model and the last two generate counter-examples on the synchronous model. When the second and third properties are validated on the asynchronous model it resulted in the race condition anomaly, but did not show the anomaly due to the ambiguous requirement. It may have been the case that the anomaly due to the ambiguity in requirement is present at a deeper depth in the computation tree but the race condition anomaly prevents us from observing it. Hence we had to change the second and third properties for the asynchronous model resulting in the first property in Table 2 on Page 56 in order to see the anomaly due to the ambiguous requirement.

4.3.3 Missing Requirement

A missing requirement is a requirement, which is, absent from the specification. This fault class is termed 'missing requirement' as the anomaly exhibited in this fault class based on observing the outputs from the properties in Table 3 on Page 59 is due to a missing requirement in the Sensor Failure Detection (SFD) system. In the Stateflow diagram of the Sensor Failure Detection (SFD) system Figure 4 on Page 30, there is a transition with no transition condition in the fueling state. In the *running* sub-state of the fueling state the transition from the *warmup* junction to *normal* mode has no transition condition. This transition contradicts with the transition coming into *normal* mode from *single_failure* mode as that transition has a transition condition such that *fail_cnt* must be equal to zero. As per the redundant sensor information in the Sensor Failure Detection (SFD) system, if there is more than one sensor failure the system is disabled by transitioning into *shutdown* mode and if there is one failure it should go into *single_failure* mode. The anomaly in this fault class occurs due to the fact that the missing requirement contradicts the redundant sensor information. This missing requirement violates safety properties. R1 states the requirement for the redundant sensor information.

R1: If *fail_cnt* is one then *fueling_state* transitions into *single_failure* mode or if *fail_cnt* is more than one then *fueling_state* transitions into *shutdown* mode.

In order to determine if the above requirement R1 is present in our models, we formulate Linear Temporal Logic (LTL) properties as shown in Table 3 on Page 59. The property in Table 3 verifies if our models exhibit behavior, which is a contradiction to the

requirement, R1. For example: P1 in Table 3 on Page 59 states that all future states considered, in some future state *fueling_state* is in *normal* mode and *fail_cnt* is equal to one. All the properties in this fault class have been validated on both the asynchronous and synchronous models. When a Linear Temporal Logic (LTL) property is entered in Spin, it automatically negates this and creates the "*never*" clause on this negated property. Hence the Spin tool checks to see if a desired behavior never happens in the model. To validate that an undesired behavior never happens, we can either negate the property or we can select the option to indicate that the above property represents an undesired behavior that is no executions of this behavior is allowed. We select the latter option and Spin will create a "*never*" clause for the above formulas 'as is' without negating it. Once the selection is made, Spin checks to see if P1 and P2 in Table 3 are satisfied at any point in our models.

Property No	*Table 3: Linear Temporal Logic Properties*
P1	Formula: [] (<> p) Symbol: #define p (fueling_state == normal) && (fail_cnt == 1)
P2	Formula: [] (<> p) Symbol: #define p (fueling_state == normal) && (fail_cnt > 1)

Table 3: Missing Requirement

4.3.3.1 Results

As per the redundant sensor information in the Sensor Failure Detection (SFD) system, if there is more than one sensor failure the system is disabled by transitioning into *shutdown* mode and if there is one failure it should go into *single_failure* mode. This redundant

information contradicts the absence of no transition condition from the *warmup* junction to *normal* mode. We have explained the requirement for the redundant sensor information and Linear Temporal Logic (LTL) properties formulated in Table 3 on Page 59 to validate the requirement. When the properties in Table 3 are validated, the Spin tool generates outputs or counter-examples which indicates that the properties and in turn the requirement R1 is violated in our models. By tracing the sequence of events in the outputs, we observe that the requirement R1 is violated in both our models. Both the properties in Table 3 represent the same anomaly due to the missing transition condition but exhibit a different sequence of events in the output.

Upon observing the outputs for the first property in Table 3, the sequence of events that represent the anomaly happens when *fueling_state* is in *warmup* mode and *o2_mode* is in o2*_normal* so *warmup* transitions to the junction. Then if *fail_cnt* is equal to one, instead of *warmup* mode transitioning to *single_failure* mode, it transitions to *normal* mode even though both transitions are executable. The reason is due to the missing condition, there is no transition condition to transition to *normal* mode. In the Spin tool, if more than one choice is available, it randomly chooses any path. Hence when *fail_cnt* is equal to one, both the transition to *normal* mode or *single_failure* mode can be taken even though the transition to *single_failure* mode has a condition of *fail_cnt* equal to one, it can choose to go to *normal* mode as both transitions hold true. This anomalous behavior is seen in the outputs by the first property and the second property shows a different sequence of events. In the outputs of the second property in Table 3, if *fail_cnt* exceeds one, and *fueling_state* is in *warmup* mode and *o2_mode* is in o2*_normal*,

again the transition to *normal* mode can be taken even though it should be exiting *running* and going into *shutdown* mode. Both the transition to *normal* mode and the transition for exiting *running* mode and going into *shutdown* mode hold true at the same time. The outputs from both the properties in Table 3 violate the redundant sensor information as seen in our requirement, R1 and it happens due to the missing transition condition. An example of a Spin generated counter-example by running a *"trail"* file through guided simulation for P1 in Table 3 on the asynchronous model is included in Appendix C.

4.3.4 Inconsistent Requirement

Inconsistent requirements are requirements, which can lead to erratic and irregular behavior. This fault class is termed 'inconsistent requirement' as in the Stateflow diagram of the Sensor Failure Detection (SFD) system, the transition in the fueling state exiting from *warmup* mode has a transition condition that *o2_mode* must be in *o2_normal*. The anomaly in this fault occurs due to an inconsistency with the transition condition exiting out of *warmup* mode. In any throttle body fuel injection system a simple requirement is that the system must progress beyond its' warmup state when started up. R1 specifies this requirement:

R1: *fueling_state* must progress from its' *warmup* mode.

In order to determine if the above requirement R1 is present in our models, we must formally prove that it is not violated in our models. In Spin there are three ways to validate a property. They are, by interactive simulation, asserts statements, and Linear Temporal Logic (LTL) formulas. Instead of formulating a Linear Temporal Logic (LTL)

property we show this anomaly due to an inconsistent transition condition by interactive simulation, and observing the output of the simulation. The anomaly is present in both the asynchronous and synchronous model.

4.3.4.1 Results

The transition condition of *in(o2_normal)* which means oxygen sensor state should be in *o2_normal* is inconsistent. This transition condition of *in(o2_normal)* violates the requirement R1 on Page 61 in certain sequence of events. We observe the outputs from an interactive simulation to exhibit this anomalous behavior in both our "naive" models. The following events exhibit the anomaly as seen in the output from interactive simulation:

* The timed event *t* is less than *o2_t_thresh*, *fueling_state* is in *warmup* mode, oxygen sensor state is in *o2_warmup*, *running_history* is in *warmup* mode, and *fail_cnt* is zero
* Either in the next state change or in a future state *speed* exceeds *max_speed* which takes *fueling_state* to *overspeed* mode, and all the remaining states stay the same as mentioned above
* While *fueling_state* is in *overspeed* mode, *t* exceeds *o2_t_thresh* which takes the oxygen sensor state to *o2_normal*, *fail_cnt* is still zero and *speed* is still greater than *max_speed*
* *o2_mode* fails which takes oxygen sensor state to *o2_fail*, *fail_cnt* becomes one

* *speed* decreases to below *max_speed* and *fueling_state* goes to *warmup* mode due to *running_history* being in *warmup* mode
* *fueling_state* cannot transition to *single_failure* mode even though *fail_cnt* is one as oxygen sensor state is not in *o2_normal* but in *o2_fail*

Hence even though oxygen sensor has failed the *fueling_state* is not indicating this failure. In fact, the only way for *fueling_state* to get out of *warmup* mode is to have another sensor failure, which would take the *fueling_state* to *shutdown* mode. In other words *fueling_state* is stuck in *warmup* mode. This behavior clearly violates the requirement, R1 and is due to the transition condition of *in(o2_normal)* which is inconsistent as seen from the above sequence of events. This anomaly was initially found by formal specification and we have shown the existence of it in the Sensor Failure Detection (SFD) system through formal verification.

4.3.5 Loss of Failures and Recoveries

In any sensor failure detection system one of the main characteristic is that all sensor failures and recoveries should be indicated. But in the synchronous Sensor Failure Detection (SFD) system model, there is a loss of sensor failures and recoveries. Upon observing the outputs generated by the properties in Table 4 on Page 64, we conclude that all the outputs exhibit a loss of sensor failures and recoveries. The properties in Table 4 characterize potential failures resulting from violations of safety properties. Requirement R1 specifies that sensor failures and recoveries should always be indicated in the fueling state:

R1: All sensor failures and recoveries must be indicated in the *fueling_state*.

In order to determine if the above requirement R1 is present in our models, we formulate Linear Temporal Logic (LTL) properties as shown in Table 4 on Page 64. All the outputs generated by the properties in Table 4, indicate a loss of sensor failures and recoveries even though the sequence of events leading to the loss of sensor failure and recoveries by each property is different. We expound one property from Table 4. P2 states that considering all future states, in some future state *fueling_state* must be in *single_failure* mode, *fail_cnt* must be equal to zero, *speed* must not exceed *max_speed*, *fueling_state* should indicate that there are no changes to any of the sensors and *fueling_state* must be executable. All the formulas were verified on our synchronous model.

Property No	*Table 4: Linear Temporal Logic Properties*
P1	Formula: [] (<> p) Symbol: #define p (fueling_state == normal) &&(fail_cnt == 1) && (speed < MAX_SPEED) && (fueling_syn == nochange) && (synchronize == 5)
P2	Formula: [] (<> p) Symbol: #define p (fueling_state == single_failure) &&(fail_cnt == 0) && (speed < MAX_SPEED) && (fueling_syn == nochange) && (synchronize == 5)
P3	Formula: [] (<> p) Symbol: #define p (fueling_state == warmup) &&(fail_cnt == 1) && (speed < MAX_SPEED) && (fueling_syn == nochange) && (synchronize == 5) && (t > o2_t_thresh)

Table 4: Loss of Failures and Recoveries

4.3.5.1 Results

Even though the synchronous model has been explained in the previous chapter, it will be worthwhile to reiterate it briefly. The synchronous processes and their order of execution are the timed event *t*, oxygen, pressure, throttle and speed sensor states and then finally the fueling state. There are two more processes for simulating the sensor failures and the engine speed but they are random in execution. The point to note is even though sensor failures and engine speed processes are random they are synchronized in the sense that once a sensor fails or recovers or there is a change in the engine speed, the processes are locked from changing back again. The sensors and engine speed can only change back again after the present change is noted in the respective sensor states and in the fueling state. For example if engine speed exceeds, the engine speed is blocked from decreasing until *fueling_state* transitions to *overspeed* mode.

The best way to explain the loss of sensor failures and recoveries is to go through the sequence of events that exhibit the anomalous behavior for the second property in Table 4.

- *fueling_state* is in *single_failure* mode and *running_history* is in *single_failure* mode and *fail_cnt* equals to one
- then if *speed* exceeds *max_speed* which transitions *fueling_state* to *overspeed* mode but *running_history* is still in *single_failure* mode and *fail_cnt* is equal to one, and execution proceeds to the timed event *t*
- either in the next state or in a future state if *speed* decreases below *max_speed* and *fail_cnt* equals to zero indicating there are no sensor failures, this makes

fueling_state exit out of *overspeed* mode and go to *single_failure* mode due to *running_history* and execution goes to the timed event *t* as per the synchronized processes and their order

- then when *fueling_state* process is reached again instead of *single_failure* mode transitioning to *normal* mode as *fail_cnt* equals to zero *fueling_state* cannot see this failure recovery

This happens due to the fact that *fueling_state* makes only one transition at a time hence when *speed* decreased and *fail_cnt* went to zero it only took into consideration the lowered *speed* and transitioned to *single_failure* mode. But it has no memory of the fact that in the next execution it should go to *normal* mode due to the sensor recovery. The loss of sensor failure and recoveries anomaly is only present in the synchronous model.

4.4 Properties with No Errors

In the previous section, we classified all the outputs or test cases, which were, generated by the properties that in turn classified all the properties and the related requirements. There were a number of properties, which generate no counter-examples, or in other words the properties are not violated in both our models as seen in Table 5 on Page 68-69. All the requirements, R1-R5 in this fault class signify the functionality of the Sensor Failure Detection (SFD) system that we have gathered from the semantics and transition conditions of the Sensor Failure Detection (SFD) system.

R1: If *t* is less than or equal to *o2_t_thresh* then oxygen sensor state should be in *o2_warmup*.

R2: If all sensor states are in normal then *fail_cnt* should be equal to zero until one or

more sensors fail or *fail_cnt* is zero if no sensors fail.

R3: *fail_cnt* should never be less than zero or greater than four since there are only four sensors.

R4: If any one sensor or a combination or sensors have failed then the *fail_cnt* should indicate the right number of failed sensors.

R5: If fueling state is in *overspeed* mode and *running_history* is in *single_failure* mode then if *speed* decrements and there are no multiple sensor failures then *fueling_state* must go to *single_failure* mode due to the *running_history*.

All the properties in Table 5 are formulated to verify if the requirements R1-R5 are violated in both our models. All the properties characterize potential failures resulting from violations of invariance properties on the timed event *t*, sensor failure counter and the fueling state. We explain which properties in Table 5 cover which requirements, R1-R5. We expound one property in Table 5 for each requirement, R1-R5. P1 covers R1 and the formula states that for all future states if *t* is less than or equal to *o2_t_thresh* then *o2_mode* must be in *o2_warmup*. U is another binary temporal logic operator that stands for 'until'. The term 'x U y' means that x is true until y is true after which x may or may not be true. x must be true from the initial state and y must hold before x becomes false and y must be true in some state. There is also a weaker version of the until operator which basically states that y need not be true at some future state and that x may always hold. This operator is not supported in Spin but can be formulated in Spin using other operators. P2 covers R2 and the formula states that in all future states if all sensors are normal then *fail_cnt* is equal to zero until one or more sensors fail or always *fail_cnt* is

equal to zero. Properties 3 and 4 covers R3 and states that *fail_cnt* must not go beyond the minimum and maximum values of zero and four respectively. Properties 5 through 8 covers R4. For example: P6 shows that for all future states if all sensors have failed then *fail_cnt* must equal four. Property P9 covers R5 and states that in all future states if *fueling_state* is in *overspeed* mode and *running_history* is in *single_failure* mode then eventually *fueling_state* will not be in *normal* mode or *warmup* mode until *fueling_state* is in *single_failure* mode and *speed* < *max_speed* or *fueling_state* is in *shutdown* mode and *speed* < *max_speed*. All the properties in Table 5 validated all their respective requirements R1-R5 that is all the requirements are present in both our models.

Property No	*Table 5: Linear Temporal Logic Properties with no errors for both models*
P1	Formula: [] (q -> p U (r || [] (p))) Symbol: #define p (fail_cnt == 0) #define q (o2_mode == o2_normal) && (press_mode == press_normal) && (throt_mode == throt_normal) && (speed_mode == speed_normal) #define r (o2_mode == o2_fail) && (press_mode == press_fail) && (throt_mode == throt_fail) && (speed_mode == speed_fail)
P2	Formula: [] (p -> q) Symbol: #define p (t <= o2_t_thresh) #define q (o2_mode == o2_warmup)
P3	Formula: [] ! (p) Symbol: #define p (fail_cnt > 4)
P4	Formula: [] ! (p) Symbol: #define p (fail_cnt < 0)
P5	Formula: [] (p -> q) Symbol: #define p (o2_mode == o2_fail) && (press_mode == press_normal) && (throt_mode == throt_normal) && (speed_mode == speed_normal) #define q (fail_cnt == 1)
P6	Formula: [] (p -> q) Symbol: #define p (o2_mode == o2_fail) && (press_mode == press_fail) && (throt_mode == throt_fail) && (speed_mode == speed_fail)

	#define q (fail_cnt == 4)
P7	Formula: [] (p -> q) Symbol: #define p (o2_mode == o2_normal) && (press_mode == press_normal) && (throt_mode == throt_normal) && (speed_mode == speed_normal) #define q (fail_cnt == 0)
P8	Formula: [] (p -> q) Symbol: #define p ((o2_mode == o2_fail) && (press_mode == press_fail) && (throt_mode == throt_normal) && (speed_mode == speed_normal)) || ((o2_mode == o2_fail) && (press_mode == press_normal) && (throt_mode == throt_fail) && (speed_mode == speed_normal)) || ((o2_mode == o2_fail) && (press_mode == press_normal) && (throt_mode == throt_normal) && (speed_mode == speed_fail)) #define q (fail_cnt == 2)
P9	Formula: [] (p -> <> r U (q || [] r)) Symbol: #define p (fueling_state == overspeed) &&(running_history == single_failure) #define q ((fueling_state == single_failure) & (speed < MAX_SPEED) || (fueling_state == shutdown) & (speed < MAX_SPEED)) #define r (fueling_state != normal) && (fueling_state != warmup)

Table 5: Linear Temporal Logic properties without errors for both models

4.5 Discussion

This chapter brought together the fault-injection methods applied in Chapter 3 with model checking to determine the behavior of the Sensor Failure Detection (SFD) system. We classified the outputs from the properties so that it would aid in their presentation and understanding. But this classification enabled us to see anomalies, patterns of anomalies, and the related requirements, which were, violated in the two Sensor Failure Detection (SFD) system models. We presented five fault classes that were identified based on the outputs of the properties. We also outlined those properties that yielded no outputs in our models. Based on the results, we can draw the following points:

* Anomalies in the asynchronous model such as race conditions were no longer present in the synchronous model. This was due to the fact that the race condition anomaly was due to complete asynchrony and synchronizing the Sensor Failure Detection (SFD) system model, corrected the race condition. This demonstrates how examining the outputs or test cases for the cause behind the anomalous behavior can enable us to correct the errors in our model and maintain fidelity between our model and the specification.
* In some cases, anomalies, which were present in both models such as missing requirement, occurred at different depths in the computation tree. The Missing requirement anomaly occurs much earlier during validation in the asynchronous model than in the synchronous model. This is because in the synchronous model a greater number of steps had to be executed due to synchronization.

* Certain anomalies were only present in the synchronous model such as loss of sensor failures and recoveries. The reason behind why the anomalies in the synchronous model were not present in the asynchronous model is due to the fact that fairness is established in the synchronous model. Hence the "naive assumption" of fairness led to the discovery of potential anomalies.
* Some Linear Temporal Logic (LTL) properties detected anomalies in the asynchronous model but the same properties showed different anomalies for the synchronous model. This is the case of uncovering anomalies where once the race condition anomaly was no longer present other anomalies were uncovered. We call this "peeling the onion" as there are many anomalies present in the asynchronous model that can be filtered out under some synchrony assumptions. This process of "peeling the onion" leads us to other anomalies present at deeper depths in the computation tree.
* There were some cases where Linear Temporal Logic (LTL) properties applied to synchronous model showed anomalies but could not detect the same anomaly for the asynchronous model. In this case the Linear Temporal Logic (LTL) property was changed by introducing further conditions for the asynchronous model which then showed the same anomaly.
* Fault-injection through inputs had the race condition anomaly in the asynchronous model but did not produce any counter-examples for the other model.

* Test cases or counter-examples enabled us to maintain fidelity between our models, specification and implementation. In addition some anomalies demonstrated how we had misinterpreted certain requirements in the Sensor Failure Detection (SFD) system.
* Both our Sensor Failure Detection (SFD) models are different but they intersect in the cases where anomalies are present in both the models. Figure 12 on Page 71 illustrates as to which anomalies were present in which of our models and which anomalies were present in both our models. Fault-injection through inputs which was applied to both our models only had the race condition anomaly in the asynchronous model.

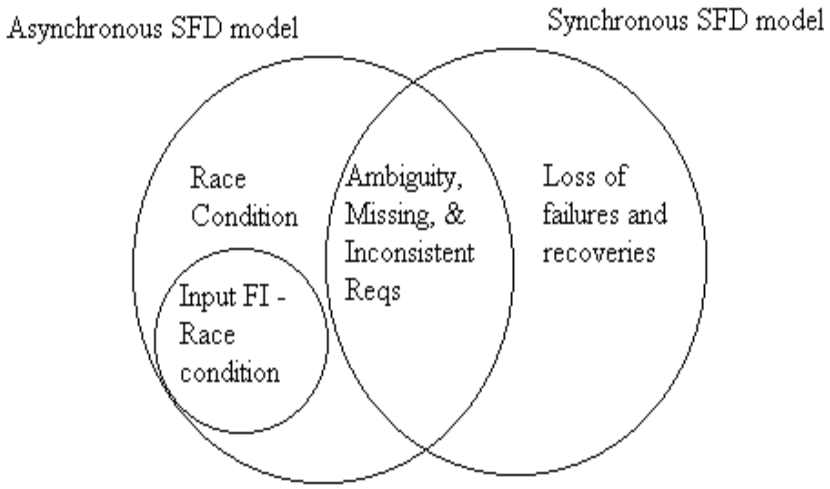

Figure 12: Difference between our SFD Models

In this chapter, we detailed all the Linear Temporal Logic (LTL) properties, which were formulated to discover all the anomalies in the Sensor Failure Detection (SFD) system.

We classified all the properties that generated counter-examples into five fault classes and described each fault class. The next chapter concludes the thesis by discussing some final thoughts and future work.

Chapter 5

Conclusions and Future Work

This chapter summarizes and concludes the thesis, in addition to discussing about future directions. This thesis describes and discusses the research, methods and results of fault-injection through model checking via “naive assumptions”. We injected faults using three methods: “naive assumptions” of asynchronous and synchronous executions and through inputs. The methods and results were conducted on a fuel injection Sensor Failure Detection (SFD) system, a model provided to us in Stateflow. We started with a detailed description of the system and then described the notion of “naive assumptions”. We used the “naive assumptions” to specify two models of the Sensor Failure Detection (SFD) system in the model checking tool Spin. Through these “naive assumptions” we injected faults into our models, hence our models are indeed “naive” with respect to synchrony, fairness and execution order. We also injected faults in both our models through inputs using various combinations of the inputs during system startup.

In order to detect the anomalies present in our models, Linear Temporal Logic (LTL) properties were used to exhaustively validate the models. Some of the properties validated generated counter-examples that is the related requirements represented by the

properties were violated. As stated before, the counter-examples generated by Spin the model checking tool upon running the guided simulation of the *"trail"* files when an undesirable behavior is present are actually test cases. We studied all the test cases and based on our observations we identified five fault classes. These five fault classes include: race conditions, missing requirements, ambiguities, inconsistent requirements, and improper failure and recovery. Classifying the test cases also in turn classified our properties and the related requirements validated by the properties. We took each fault class and showed examples of anomalous behaviors in the Sensor Failure Detection (SFD) system by going through the generated test cases. We also stated all those properties, which did not show any anomalous behavior in our models. In conclusion of the work, we drew comparisons and analogies between the two Sensor Failure Detection (SFD) system models based on the anomalies detected, related requirements violated and Linear Temporal Logic (LTL) properties validated.

The results of this investigation have shown that combining fault-injection with model checking can help detect potential complex anomalies. Fault-injection allows us to examine potential anomalous program behaviors and model checking searches through all paths in a model looking for violations of linear temporal logic formulas. The counter-examples generated by a model checker, which serve as test cases aids developers in keeping the fidelity between the models, specification, and the source code during development. This can be achieved by studying the test cases for the errors behind the anomaly, correcting the error in the source code during development, and identifying inconsistencies in the specification. This characteristic of model checking is very helpful

when we are checking for invariant and safety properties that are very difficult to verify in an implementation. Our "naive assumptions" helped us create different models while injecting faults that led to the discovery of different anomalies as we had different semantic interpretations of the Sensor Failure Detection (SFD) system. We have shown that anomalies present in one model may not be present in another model and when an anomaly is fixed in one model, it can reveal another anomaly. Although a state-based model must exist which in our case it did, it is not a problem if it doesn't as there are plenty of ways to model a design in state machine theory. In spite of all this, we are aware of the limits of fault-injection and model checking. Fault-injection is only as good as the faults injected and anomalies simulated. Fault-injection cannot prove correctness but only shows how bad the software can behave in the future. The main point in fault-injection is the correct interpretation of the results without which all the efforts would be wasted. Model checking is only successful if we are able to abstract the relevant aspects of the design, know which properties we want verified and specify the properties in a concise way. But nevertheless, both are good software quality assessment techniques and when combined can given interesting and meaningful results.

We utilized two "naive assumptions" of asynchronous and synchronous executions in addition to inputs to inject the faults. In the asynchronous model, all the processes were asynchronous and in the synchronous model, synchronization was achieved by assuming left to right, top to bottom execution order in the statechart. But there could be other possibilities such as cyclic or acyclic synchronization of processes or complete synchronization or partial synchronization. In the future, other "naive

assumptions" such as removing certain transitions, different interpretations for the transition conditions, interruption of execution to inject faults, and reading the inputs from the command line during which faults could be injected can also be explored. There could also be other Linear Temporal Logic (LTL) properties, which can be validated for this model. Based on the fact that this thesis examined a finite state machine model, there could be further work into establishing a general class of "naive assumptions" which can be applied to inject faults in model checking. One can also conduct mutation analysis studies specific to the Sensor Failure Detection (SFD) system and based on the results establish a general class of mutation operators.

References

[1] J.A.Clark and D.K.Pradhan, “Fault Injection: A method for Validating Computer-System Dependability”, IEEE Computer, pages 47-56, June 1995.

[2] Glenford J. Myers, “The Art of Software Testing”, John Wiley & Sons, 1979.

[3] Boris Beizer, “Software Testing Techniques”, International Thompson Computer Press, 1990.

[4] Michael R.Lyu, “Software Fault Tolerance”, John Wiley & Sons, 1995.

[5] Jeffrey M. Voas and Gary McGraw, “Software Fault Injection: Inoculating Programs Against Errors”, John Wiley & Sons, 1998.

[6] Paul C. Jorgensen, “Software Testing: A Craftsman’s Approach”, CRC Press, 1995.

[7] Gerard Holzmann, “The model checker SPIN”, IEEE Transactions on Software Engineering, 23(5):279-295, May 1997.

[8] Paul E. Ammann, Paul E. Black, and William Majurski, “Using model checking to generate tests from specifications”, In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM’98) (to be published), IEEE, December 1998.

[9] John R. Callahan, Frank Schneider, and Steve Easterbrook, “Automated software testing using model-checking”, In Proceedings 1996 SPIN Workshop, Rutgers, NJ, August 1996. Also WVU Technical Report #NASA-IVV-96-022.

[10] Jeffrey Voas, “Software Fault-injection: Growing ‘Safer’ Systems”, In Proceedings of IEEE Aerospace Conference, February 1997, Snowmass, CO.

[11] J. Voas, G. McGraw, L.Kassab, and L.Voas, “Fault-injection: A Crystal Ball for

Software Quality", IEEE Computer, Volume 30, Number 6, pp. 29-36, June 1997.

[12] Jeffrey Voas, "Fault Injection for the Masses", IEEE Computer, December 1997.

[13] Jeffrey Voas and Keith Miller, "Using Fault Injection to Assess Software Engineering Standards", In Proceedings of International Symposium on Software Engineering Standards, August 1995.

[14] Matthew B. Dwyer, George S. Avrunin and James C. Corbett, "Property Specification Patterns for Finite-State Verification", In the second Workshop on Formal Methods in Software Practice, March, 1998. Also available at http://www.cis.ksu.edu/~dwyer/spec-patterns.html.

[15] F. Schneider, S.M.Easterbrook, J.R.Callahan and G.J.Holzmann, "Validating Requirements for Fault Tolerant Systems using Model Checking", Third IEEE Conference on Requirements Engineering, Colorado Spring, CO, April 6-10, 1998. Also WVU Technical Report #NASA-IVV-97-014.

[16] J. R. Callahan, S. M. Easterbrook and T. L. Montgomery, "Generating Test Oracles via Model Checking," NASA/WVU Software Research Lab, Fairmont, WV, Technical Report # NASA-IVV-98-015, 1998.

[17] Anup K.Ghosh, Todd A.DeLong, and Barry W.Johnson, "Fault Injection in the Design Process Using VHDL", VHDL International Users' Forum Fall Conference, October 15-19, 1995.

[18] Zohar Manna and Amir Pnueli, "A Hierarchy of Temporal Properties", In 9th Symposium on Principles of Distributed Computing, Quebec, Canada, pp: 377-408, August, 1990.

[19] Amir Pnueli, "The Temporal Logic of Programs", In Proceedings of 18th IEEE

Symp. Found. Of Comp. Sci., 1977, pp: 46-57.

[20] Leslie Lamport, "What good is Temporal Logic", in Proceedings of IFIP 9th World Congress (R.R.A. Mason, ed.), North-Holland, pp: 657-668, 1983.

[21] S.M.Easterbrook and J.R.Callahan, "Formal Methods for V&V of partial specifications: An experience report", In Proceedings of Third IEEE International Symposium on Requirements Engineering (RE'97), Annapolis, MD, 1997.

[22] E. Mikk, Y. Lakhnech, C.Petersohn, and M.Siegel, "On Formal Semantics of Statecharts as supported by Statemate", In 2nd BCS-FACS Northern Formal Methods Workshop, Springer-Verlab, July, 1997.

[23] E.Mikk, Y.Lakhnech, M.Siegel, and G.J.Holzmann, "Implementing Statecharts in Promela/SPIN", to appear In the Proceedings of WIFT'98.

[24] Gerard Holzmann, "On Checking Model Checkers", In Proceedings of Computer Aided Verification Conference LNCS 1427, pp 61-70, Springer-Verlag, Vancouver, Canada, June 1998.

[25] E. Allen Emerson, "Temporal and Modal Logic", in Handbook of Theoretical Computer Science, J. van Leeuwen, Ed. Elsevier/The MIT Press, Amsterdam/Cambridge, MA, 1990, pp: 995-1072.

[26] G.S.Choi and R.K.Iyer, "Focus: An Experimental Environment for Fault Sensitivity Analysis," IEE Trans. Computers, Vol. 41, No. 12, Dec. 1992, pp. 1,515-1,526.

[27] J.Karlsson et al., "Two Fault-Injection Techniques for Test of Fault-Handling Mechanisms," Proc. Int'l Test Conf., IEEE CS Press, Los Alamitos, CA., Order No.2156, 1991, pp. 140-149.

[28] R. Chillarege and R.K.Iyer, "An Experimental Study of Memory Fault Latency."

IEEE Trans. Computers, Vol. 38, No. 6, June 1989, pp. 869-874.

[29] E.W.Czeck and D.P.Siewiorek, “Effects of Transient Gate-Level Faults on Program Behavior,” Proc. 20th Int’l Symp. Fault-Tolerant Computing, IEEE CS Press, Los Alamitos, CA., Order No. 2051, 1990, pp. 236-243.

[30] R.Chillarege and N.S.Bowen, “Understanding Large-System Failures: A Fault-Injection Experiment,” Proc. 19th Int’l Symp. Fault-Tolerant Computing, IEEE CS Press, Los Alamitos, CA., Order No. 1959, 1989, pp.356-363.

[31] K.K.Goswami and R.K.Iyer, “A Simulation-Based Study of a Triple-Modular Redundant System Using Depend,” Proc. Fifth Int’l Conf. Fault-Tolerant Computing Systems, IEEE Press, Piscataway, N.J., 1991, pp.300-311.

[32] "Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion" [NASA-GB-002-95], 1995, 77 pages. Also available at http://eis.jpl.nasa.gov/quality/Formal_Methods/.

[33] “IEEE Standard glossary of Software Engineering Terminology”, IEEE Std 729-1983. IEEE, 1983.

[34] G.J.Holzmann, “Basic Spin Manual.” Also available at http://cm.bell-labs.com/cm/cs/what/spin/index.html.

Appendix A

Spin Source Code for Asynchronous Model

```
#define o2_t_thresh 2
#define MAX_SPEED 30


mtype = {o2_warmup, o2_normal, o2_fail, press_normal, press_fail, throt_normal, throt_fail,
        speed_normal, speed_fail, no_state, single_failure, normal, warmup, running, disabled,
        overspeed, shutdown, no_hist, fail, norm};

byte t;

/*used to set sensor values only at startup*/
byte o2_sensor = norm, press_sensor = norm, throt_sensor = norm, speed_sensor = norm;

/*failure counter*/
byte fail_cnt = 0;

/*used to simulate speed change*/
byte speed ;

/*used to indicate all the sensor states, fueling state and running history, startup values have
already been set*/
mtype o2_mode = o2_warmup;
mtype press_mode = press_normal;
mtype throt_mode = throt_normal;
mtype speed_mode = speed_normal;
mtype fueling_mode = running;
mtype fueling_state = warmup;
mtype running_history = warmup;
byte count = 0;


/*function asynchronously changes press sensor and state and increases failure counter*/
active proctype Pressure_Sensor_Mode() {
```

```
/*At startup it changes press state to fail depending on the startup values for press_sensor, disables press_sensor as only used for startup*/

        if
        :: atomic{press_sensor == fail -> press_mode = press_fail; press_sensor = 2; fail_cnt =
        fail_cnt + 1;}
        :: atomic{press_sensor == norm -> press_sensor = 2;}
        fi;

/*after startup the following changes press state asynchronously*/
        do
        :: (press_mode == press_normal) ->
                atomic{press_mode = press_fail; fail_cnt = fail_cnt + 1;}
        :: (press_mode == press_fail) ->
                atomic{press_mode = press_normal; fail_cnt = fail_cnt - 1;}
        od;
}

/*function asynchronously changes o2 sensor and state and increases failure counter*/
active proctype O2_Sensor_Mode() {

/*if t goes beyond t > o2_t_thresh o2 goes to normal*/

        if
        :: atomic{t > o2_t_thresh -> o2_mode = o2_normal;}
        fi;

/*At startup it changes o2 state to fail depending on the startup values for o2_sensor, disables o2_sensor as only used for startup*/
        if
        :: atomic{(o2_sensor == fail) && (t > o2_t_thresh) -> o2_mode = o2_fail; o2_sensor =
2;      fail_cnt = fail_cnt + 1;}
        :: atomic{(o2_sensor == norm) && (t > o2_t_thresh)-> o2_sensor = 2;}
        fi;

/*after startup the following changes o2 state asynchronously*/
        do
        :: (o2_mode == o2_normal) ->
                 atomic{o2_mode = o2_fail; fail_cnt = fail_cnt + 1;}
        :: (o2_mode == o2_fail) ->
                atomic{o2_mode = o2_normal; fail_cnt = fail_cnt - 1;}
        od;
}

/*function asynchronously changes throt sensor and state and increases failure counter*/
active proctype Throttle_Sensor_Mode() {

/*At startup it changes throt state to fail depending on the startup values for throt_sensor, disables throt_sensor as only used for startup*/
        if
```

```
        :: atomic{throt_sensor == fail -> throt_mode = throt_fail; throt_sensor = 2; fail_cnt =
        fail_cnt + 1;}
        :: atomic{throt_sensor == norm -> throt_sensor = 2;}
        fi;

/*after startup the following changes throt state asynchronously*/
        do
        :: (throt_mode == throt_normal) ->
                atomic{throt_mode = throt_fail; fail_cnt = fail_cnt + 1;}
        :: (throt_mode == throt_fail) ->
                atomic{throt_mode = throt_normal; fail_cnt = fail_cnt - 1;}

        od;
}

/*function asynchronously changes speed sensor and state and increases failure counter*/
active proctype Speed_Sensor_Mode() {

/*At startup it changes speed state to fail depending on the startup values for speed_sensor,
disables speed_sensor as only used for startup*/
        if
        :: atomic{speed_sensor == fail -> speed_mode = speed_fail; speed_sensor = 2; fail_cnt =
        fail_cnt + 1;}
        :: atomic{speed_sensor == norm -> speed_sensor = 2;}
        fi;

/*after startup the following changes speed state asynchronously*/
        do
        :: (speed_mode == speed_normal) ->
                atomic{speed_mode = speed_fail; fail_cnt = fail_cnt + 1;}
        :: (speed_mode == speed_fail) ->
                atomic{speed_mode = speed_normal; fail_cnt = fail_cnt - 1;}

        od;
}

/*function increases t to beyond t_o2_thresh for the o2 sensor state*/
active proctype Time_Counter(){

        do
        :: atomic{(t <= o2_t_thresh) -> t = 3;}
        od;

}

/*function asynchronously changes speed from less than to greater than max_speed*/
active proctype Speed_Change_Mode(){

        do
        :: count == 1 -> atomic{speed = 5; count = 2;}
```

```
        :: count == 0 -> atomic{speed = 35; count = 2;}
        od;

}


/*function changes fueling state depending on if there have been any sensor state changes or
speed changes*/

active proctype Fueling_Mode(){


Running:        do

/*if speed> max_speed the fueling state is set to overspeed*/
        :: atomic{((fueling_mode == running)  && (speed > MAX_SPEED)) -> fueling_mode =
        disabled; fueling_state = overspeed; goto Disabled;}

/*if fail_cnt > 1 the fueling state is set to shutdown*/
        :: atomic{((fueling_mode == running)  && (fail_cnt > 1)) -> fueling_mode = disabled;
        fueling_state = shutdown; goto Disabled;}

/* fail_cnt > 1 or speed > max_speed, control goes to running*/
        :: ((fueling_mode == running)  && (speed < MAX_SPEED) && (fail_cnt <= 1)) ->

                do


                :: (speed == 5) -> atomic{count = 0; speed = 0;}

/* fail_cnt > 1 or speed > max_speed, control goes to running*/
                :: atomic{((fail_cnt > 1) || (speed > MAX_SPEED)) -> goto Running;}

/*if fueling state == warmup, running_history == warmup, speed < max_speed but there is one
failure, and o2 state is in normal fueling state is set to single_failure*/
                :: atomic{((fueling_mode == running) && (running_history == warmup) &&
                (fueling_state == warmup) && (o2_mode == o2_normal) && (fail_cnt == 1)
&&              (speed < MAX_SPEED))  -> fueling_state = single_failure; running_history =
                single_failure;}

/*if fueling state == warmup, running_history == warmup, speed < max_speed  but fail_cnt != 1,
and o2 state is in normal fueling state is set to normal*/
                :: atomic{((fueling_mode == running) && (running_history == warmup) &&
                (fueling_state == warmup) && (o2_mode == o2_normal) && (speed <
                MAX_SPEED)) -> fueling_state = normal; running_history = normal;}

/*if fueling state == normal, running_history == normal, speed < max_speed  but fail_cnt == 1
fueling state is set to single_failure*/
```

```
                :: atomic{((fueling_mode == running) && (running_history == normal) &&
                (fueling_state == normal) && (fail_cnt == 1) && (speed < MAX_SPEED)) ->
                fueling_state = single_failure; running_history = single_failure;}

/*if fueling state == single_failure, running_history == single_failure, speed < max_speed  but
fail_cnt == 0 fueling state is set to normal*/
                :: atomic{((fueling_mode == running) && (running_history == single_failure)
                && (fueling_state == single_failure) && (fail_cnt == 0) && (speed <
                MAX_SPEED)) -> fueling_state = normal; running_history = normal;}

                od;
        od;


Disabled:       do

        :: (speed == 35) -> atomic{count = 1; speed = 30;}

/*if fueling state is in overspeed and fail_cnt > 1 but speed < max_speed fueling state is set to
shutdown*/
        :: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed <
        MAX_SPEED) && (fail_cnt > 1)) -> fueling_state = shutdown;}

/*if fueling state is in overspeed but speed < max_speed fueling state is set to the warmup if
running_history was at warmup*/
        :: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed <
        MAX_SPEED) && (fail_cnt <= 1) && (running_history == warmup)) -> fueling_mode
=       running; fueling_state = warmup; goto Running;}

/*if fueling state is in overspeed but speed < max_speed fueling state is set to the normal if
running_history was at normal*/
        :: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed <
        MAX_SPEED) && (fail_cnt <= 1) && (running_history == normal)) -> fueling_mode =
        running; fueling_state = normal; goto Running;}

/*if fueling state is in overspeed but speed < max_speed fueling state is set to the single_failure if
running_history was at single_failure*/
        :: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed <
        MAX_SPEED) && (fail_cnt <= 1) && (running_history == single_failure)) ->
        fueling_mode = running; fueling_state = single_failure; goto Running;}

/*if fueling state is in shutdown, fail_cnt <= 1, fueling state is set to single_failure*/

        :: atomic{((fueling_mode == disabled) && (fueling_state == shutdown) && (fail_cnt <=
                1)) -> fueling_mode = running; fueling_state = single_failure; running_history =
        single_failure; goto Running;}

        od;
}
```

Appendix B

Spin Source Code for Synchronous Model

```
#define o2_t_thresh 2
#define MAX_SPEED 30


mtype = {o2_warmup, o2_normal, o2_fail, press_normal, press_fail, throt_normal, throt_fail,
        speed_normal, speed_fail, no_state, single_failure, normal, warmup, running, disabled,
        overspeed, shutdown, no_hist, change, nochange, fail, norm };

byte t;

/*used to simulate sensor failures*/
mtype o2_sensor = norm, press_sensor = norm, throt_sensor = norm, speed_sensor = norm;

/*failure counter*/
byte fail_cnt = 0;

/*used to simulate speed change*/
byte speed = 5;

/*used to indicate all the sensor states, fueling state and running history*/
mtype o2_mode = o2_warmup;
mtype press_mode = press_normal;
mtype throt_mode = throt_normal;
mtype speed_mode = speed_normal;
mtype fueling_mode = running;
mtype fueling_state = no_state;
mtype running_history = no_hist;

/* var synchronizes sensor, fueling states*/
byte synchronize=0;

/*synchronizes changes between sensor values, and speed values*/
mtype o2_syn = change, press_syn = change, throt_syn = change, speed_syn = change,
fueling_syn = change, fast = change, slow = change;
```

```
/*function increases t to beyond t_o2_thresh for the o2 sensor state*/
active proctype Time_Counter(){

        do
        :: atomic{(t <= o2_t_thresh) && (synchronize == 0) -> t = t + 1; synchronize = 1;}
        :: atomic{(t > o2_t_thresh) && (synchronize == 0) -> synchronize = 1;}
        od;

}

/*function makes changes to the o2 state to either fail or normal depending on the value of
o2_sensor, and notifies fueling state if there has been a change through a synchronization
variable*/
active proctype O2_Sensor_Mode() {


/*if t goes beyond o2_t_thresh then o2 state goes to o2_normal*/
        do
        :: atomic{ (t > o2_t_thresh) && (synchronize == 1) -> o2_mode = o2_normal;
        synchronize = 2; fueling_syn = change; break;}
        :: atomic{ (t <= o2_t_thresh) && (synchronize == 1) -> synchronize = 2;}
        od;

/*At startup it changes o2 state to fail or normal depending on the startup values for o2_sensor*/

        if
        :: atomic{(synchronize == 1) && (o2_sensor == fail) -> o2_mode = o2_fail; synchronize
        = 2; o2_syn = nochange; fueling_syn = change;}
        :: atomic{(synchronize == 1) && (o2_sensor == norm) -> synchronize = 2; o2_syn =
        nochange;}
        fi;

/*after startup for the rest of the execution, the o2 state is changed to fail or normal if the values
of o2_sensor have changed recently, if no change recently if just goes to the next process*/

        do
        :: atomic{(o2_sensor == fail) && (synchronize == 1) && (o2_syn == change) ->
                o2_mode = o2_fail; fail_cnt = fail_cnt + 1; synchronize = 2; o2_syn =
nochange;                       fueling_        syn = change;}
        :: atomic{(o2_sensor == norm) && (synchronize == 1) && (o2_syn == change) ->
                o2_mode = o2_normal; fail_cnt = fail_cnt - 1; synchronize = 2; o2_syn =
                        nochange; fueling_syn = change;}
        :: atomic{(o2_syn == nochange) && (synchronize == 1) -> synchronize = 2;}
        od;
}
```

```
/*function makes changes to the press state to either fail or normal depending on the value of press_sensor, and notifies fueling state if there has been a change through a synchronization variable*/
active proctype Pressure_Sensor_Mode() {

/*At startup it changes press state to fail or normal depending on the startup values for press_sensor*/
        if
        :: atomic{(press_sensor == fail) && (synchronize == 2) -> press_mode = press_fail;
        fail_cnt = fail_cnt + 1; synchronize = 3; press_syn = nochange; fueling_syn = change;}
        :: atomic{(press_sensor == norm) && (synchronize == 2) -> synchronize = 3; press_syn
=       nochange; fueling_syn = change;}
        fi;

/*after startup for the rest of the execution, the press state is changed to fail or normal if the values of press_sensor have changed recently, if no change recently if just goes to the next process*/

        do
        :: atomic{(press_sensor == fail)  && (synchronize == 2) && (press_syn == change) ->
                press_mode = press_fail; fail_cnt = fail_cnt + 1; synchronize = 3; press_syn =
                nochange; fueling_syn = change;}
        :: atomic{(press_sensor == norm) && (synchronize == 2) && (press_syn == change) ->
                press_mode = press_normal; fail_cnt = fail_cnt - 1; synchronize = 3; press_syn =
                        nochange; fueling_syn = change;}
        :: atomic{(press_syn == nochange) && (synchronize == 2) -> synchronize = 3;}
        od;
}

/*function makes changes to the throt state to either fail or normal depending on the value of throt_sensor, and notifies fueling state if there has been a change through a synchronization variable*/
active proctype Throttle_Sensor_Mode() {

/*At startup it changes throt state to fail or normal depending on the startup values for throt_sensor*/
        if
        :: atomic{(throt_sensor == fail) && (synchronize == 3)  -> throt_mode = throt_fail;
        fail_cnt = fail_cnt + 1; synchronize = 4; throt_syn = nochange; fueling_syn = change;}
        :: atomic{(throt_sensor == norm) && (synchronize == 3)  -> synchronize = 4; throt_syn
=       nochange; fueling_syn = change;}
        fi;

/*after startup for the rest of the execution, the throt state is changed to fail or normal if the values of throt_sensor have changed recently, if no change recently it just goes to next process*/

        do
        :: atomic{(throt_sensor == fail) && (synchronize == 3) && (throt_syn == change) ->
                 throt_mode = throt_fail; fail_cnt = fail_cnt + 1; synchronize = 4; throt_syn =
                nochange; fueling_syn = change;}
```

```
        :: atomic{(throt_sensor == norm) && (synchronize == 3) && (throt_syn == change) ->
                throt_mode = throt_normal; fail_cnt = fail_cnt - 1; synchronize = 4; throt_syn =
                nochange; fueling_syn = change;}
        :: atomic{(throt_syn == nochange) && (synchronize == 3) -> synchronize = 4;}
        od;
}


/*function makes changes to the speed state to either fail or normal depending on the value of
speed_sensor,and notifies fueling state if there has been a change through a synchronization
variable*/
active proctype Speed_Sensor_Mode() {

/*At startup it changes speed state to fail or normal depending on the startup values for
speed_sensor*/
        if
        :: atomic{(speed_sensor == fail) && (synchronize == 4)  -> speed_mode = speed_fail;
        fail_cnt = fail_cnt + 1; synchronize = 5; speed_syn = nochange; fueling_syn = change;}
        :: atomic{(speed_sensor == norm) && (synchronize == 4)  -> synchronize = 5;
speed_syn       = nochange; fueling_syn = change;}
        fi;


/*after startup for the rest of the execution, the speed state is changed to fail or normal if the
values of speed_sensor have changed recently, if no change recently it just goes to next process*/
        do
        :: atomic{(speed_sensor == fail) && (synchronize == 4) && (speed_syn == change) ->
          speed_mode = speed_fail; fail_cnt = fail_cnt + 1; synchronize = 5; speed_syn =
nochange;       fueling_syn = change;}
        :: atomic{(speed_sensor == norm) && (synchronize == 4) && (speed_syn == change) -
>
                speed_mode = speed_normal; fail_cnt = fail_cnt - 1; synchronize = 5; speed_syn
=       nochange; fueling_syn = change;}
        :: atomic{(speed_syn == nochange) && (synchronize == 4) -> synchronize = 5;}
        od;
}


/*function changes fueling state depending on if there have been any recent sensor state changes
or speed changes, if not change then it goes to the o2 process*/
active proctype Fueling_Mode(){


Running:        do
/*if speed> max_speed the fueling state is set to overspeed*/
        :: atomic{((fueling_mode == running)  && (speed > MAX_SPEED) && (synchronize
==      5) && (fueling_syn == change)) -> fueling_mode = disabled; fueling_state = overspeed;
        synchronize = 0; fueling_syn = nochange; fast = change; goto Disabled; }

/*if fail_cnt > 1 the fueling state is set to shutdown*/
```

```
        :: atomic{((fueling_mode == running)  && (fail_cnt > 1) && (synchronize == 5) &&
        (fueling_syn == change)) -> fueling_mode = disabled; fueling_state = shutdown;
        synchronize = 0; fueling_syn = nochange; goto Disabled;}

/*if there is no change in any sensors or speed the execution is passed to timed event t process*/
        :: atomic{(fueling_syn== nochange) && (synchronize == 5) -> synchronize = 0;}

/*if speed < max_speed and fail_cnt <=1*/
        :: ((fueling_mode == running)  && (speed < MAX_SPEED) && (fail_cnt <= 1) &&
        (synchronize == 5) && (fueling_syn == change)) ->

                do
/* fail_cnt > 1 or speed > max_speed, control goes to running*/

                :: atomic{(((fail_cnt > 1) || (speed > MAX_SPEED)) && (synchronize == 5) &&
                        (fueling_syn == change)) -> goto Running;}

/*if fueling state == warmup, running_history == warmup, speed < max_speed but there is one
failure, and o2 state is in normal fueling state is set to single_failure, and execution is passed to
timed event t process */

                :: atomic{((fueling_mode == running) && (running_history == warmup) &&
                (fueling_state == warmup) && (o2_mode == o2_normal) && (fail_cnt == 1)
&&              (speed < MAX_SPEED) && (synchronize == 5) && (fueling_syn == change))
                ->fueling_state = single_failure; running_history = single_failure; synchronize =
                0; fueling_syn = nochange;}

/*if fueling state == warmup, running_history == warmup, speed < max_speed  but fail_cnt != 1,
and o2 state is in normal fueling state is set to normal, and execution is passed to timed event t
process */

                :: atomic{((fueling_mode == running) && (running_history == warmup) &&
                (fueling_state == warmup) && (o2_mode == o2_normal) && (speed <
                MAX_SPEED) && (synchronize == 5) && (fueling_syn == change)) ->
                fueling_state = normal; running_history = normal; synchronize = 0; fueling_syn
=                       nochange;}

/*if fueling state == normal, running_history == normal, speed < max_speed  but fail_cnt == 1
fueling state is set to single_failure, and execution is passed to timed event t process */

                :: atomic{((fueling_mode == running) && (running_history == normal) &&
                (fueling_state == normal) && (fail_cnt == 1) && (speed < MAX_SPEED) &&
                (synchronize == 5) && (fueling_syn == change)) ->
                fueling_state = single_failure; running_history = single_failure; synchronize = 0;
                        fueling_syn = nochange;}

/*if fueling state == single_failure, running_history == single_failure, speed < max_speed  but
fail_cnt == 0 fueling state is set to normal, and execution is passed to timed event t process */
```

```
                :: atomic{((fueling_mode == running) && (running_history == single_failure)
                && (fueling_state == single_failure) && (fail_cnt == 0) && (speed <
                MAX_SPEED) && (synchronize == 5) && (fueling_syn == change)) ->
                fueling_state = normal; running_history = normal; synchronize = 0; fueling_syn
=                       nochange;}

/*if there is no change in any sensors or speed the execution is passed to timed event t process */
                :: atomic{(fueling_syn== nochange) && (synchronize == 5) -> synchronize =
0;}

                :: atomic{(fueling_syn== change) && (synchronize == 5) && (fueling_state ==
                no_state) -> fueling_state = warmup; running_history = warmup; synchronize =
0;              fueling_syn = nochange;}

/*if there is a change in sensors but o2_mode is still in o2_warmup and t < o2_t_thresh then
execution is passed to timed event t process */
                :: atomic{(fueling_syn == change) && (synchronize == 5) && (o2_mode ==
                o2_warmup) && (fueling_state != no_state) -> synchronize = 0;}

                od;
        od;


Disabled:       do

/*if fueling state is in overspeed and fail_cnt > 1 but speed < max_speed fueling state is set to
shutdown, speed synchronization variable is reset so that speed can change asynchronously again
and execution goes to timed event t process */

        :: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed <
        MAX_SPEED) && (fail_cnt > 1) && (synchronize == 5) && (fueling_syn == change))
        -> fueling_state = shutdown; synchronize = 0; fueling_syn = nochange; slow= change;}

/*if fueling state is in overspeed but speed < max_speed fueling state is set to the warmup if
running_history was at no_hist, speed synchronization variable is reset so that speed can change
asynchronously again and execution goes to timed event t process */

        :: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed <
        MAX_SPEED) && (fail_cnt <= 1) && (running_history == no_hist) && (synchronize
        == 5) && (fueling_syn == change)) -> fueling_mode = running; fueling_state = warmup;
        running_history = warmup; synchronize = 0; fueling_syn = nochange; slow = change;
goto    Running; }


/*if fueling state is in overspeed but speed < max_speed fueling state is set to the warmup if
running_history was at warmup, speed synchronization variable is reset so that speed can change
asynchronously again and execution goes to timed event t process */

        :: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed <
        MAX_SPEED) && (fail_cnt <= 1) && (running_history == warmup) && (synchronize
```

== 5) && (fueling_syn == change)) -> fueling_mode = running; fueling_state = warmup; synchronize = 0; fueling_syn = nochange; slow = change; goto Running; }

/*if fueling state is in overspeed but speed < max_speed fueling state is set to the normal if running_history was at normal, speed synchronization variable is reset so that speed can change asynchronously again and execution goes to timed event t process */

:: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed < MAX_SPEED) && (fail_cnt <= 1) && (running_history == normal) && (synchronize == 5) && (fueling_syn == change)) -> fueling_mode = running; fueling_state = normal; synchronize = 0; fueling_syn = nochange; slow = change; goto Running;}

/*if fueling state is in overspeed but speed < max_speed fueling state is set to the single_failure if running_history was at single_failure, speed synchronization variable is reset so that speed can change asynchronously again and execution goes to timed event t process */

:: atomic{((fueling_mode == disabled) && (fueling_state == overspeed) && (speed < MAX_SPEED) && (fail_cnt <= 1) && (running_history == single_failure) && (synchronize == 5) && (fueling_syn == change)) -> fueling_mode = running; fueling_state = single_failure; synchronize = 0; fueling_syn = nochange; slow = change; goto Running;}

/*if fueling state is in shutdown, but speed > max_speed, fueling state goes to single_failure and notifies fueling state that there is a change in speed through a synchronization variable, sends execution to timed event t process */

:: atomic{((fueling_mode == disabled) && (fueling_state == shutdown) && (fail_cnt <= 1) && (synchronize == 5) && (fueling_syn == change) && (speed > MAX_SPEED)) -> fueling_mode = running; fueling_state = single_failure; running_history = single_failure; synchronize = 0; goto Running; }

/*if fueling state is in shutdown, but speed < max_speed, fueling state goes to single_failure, and execution goes to timed event t process */

:: atomic{((fueling_mode == disabled) && (fueling_state == shutdown) && (fail_cnt <= 1) && (synchronize == 5) && (fueling_syn == change) && (speed < MAX_SPEED)) -> fueling_mode = running; fueling_state = single_failure; running_history = single_failure; synchronize = 0; fueling_syn = nochange; goto Running; }

/*if more than 2 sensor have failed or speed continues to be greater than max_speed then simply send the execution to the timed event t process */

:: atomic{ ((fueling_mode == disabled) && (fail_cnt >= 2) && (fueling_state == shutdown) && (fueling_syn == change) && (synchronize == 5)) || ((speed > MAX_SPEED) && (fueling_mode == disabled) && (fueling_state == overspeed) && (fueling_syn == change) && (synchronize == 5)) -> synchronize = 0; fueling_syn = nochange;}

/*if there is no change in any sensors or speed the execution is passed to timed event t process */

```
	:: atomic{(fueling_syn == nochange) && (synchronize == 5) -> synchronize = 0;}

	od;
}


/*This function is not synchronized but asynchronously changes sensor values, but once sensor is changed the change takes effect in the respective sensor state before that sensor gets changed again*/
active proctype Change_sensor_Values(){

	do
	:: atomic{(o2_sensor == norm) && (o2_syn == nochange) -> o2_sensor = fail; o2_syn = change;}
	:: atomic{(o2_sensor == fail) && (o2_syn == nochange) -> o2_sensor = norm; o2_syn = change;}
	:: atomic{(press_sensor == norm) && (press_syn == nochange) -> press_sensor = fail; press_syn = change;}
	:: atomic{(press_sensor == fail) && (press_syn == nochange) -> press_sensor = norm; press_syn = change;}
	:: atomic{(throt_sensor == norm) && (throt_syn == nochange) -> throt_sensor = fail; throt_syn = change;}
	:: atomic{(throt_sensor == fail) && (throt_syn == nochange) -> throt_sensor = norm; throt_syn = change;}
	:: atomic{(speed_sensor == norm) && (speed_syn == nochange) -> speed_sensor = fail; speed_syn = change;}
	:: atomic{(speed_sensor == fail) && (speed_syn == nochange) -> speed_sensor = norm; speed_syn = change;}

	od;

}


/*This function is not synchronized but asynchronously changes speed values, but once speed is changed the change takes effect in the fueling state before speed gets changed again*/

active proctype Change_speed_value(){

	do
	:: atomic{(speed == 5) && (slow == change) -> speed = 35; fast = nochange; fueling_syn = change;}

	:: atomic{(speed == 35) && (fast == change) -> speed = 5; slow = nochange; fueling_syn = change;}

	od;
}
```

Appendix C

Counter-example for Property 1 from Table 3

```
1:      proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 2:     proc  7 (Fueling_Mode) line 145 "sfd19" (state 11)
        [(((((fueling_mode==running)&&(speed<30))&&(fail_cnt<=1)))]
 3:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 4:     proc  6 (Speed_Change_Mode) line 125 "sfd19" (state 5)          [((count==0))]
 5:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 6:     proc  6 (Speed_Change_Mode) line 125 "sfd19" (state 6)          [speed = 35]
 7:     proc  6 (Speed_Change_Mode) line 125 "sfd19" (state 7)          [count = 2]
 8:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 9:     proc  7 (Fueling_Mode) line 153 "sfd19" (state 16)       [(((fail_cnt>1)||(speed>30)))]
10:     proc  7 (Fueling_Mode) line 153 "sfd19" (state 17)       [goto]
11:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
12:     proc  7 (Fueling_Mode) line 139 "sfd19" (state 1)
        [(((fueling_mode==running)&&(speed>30)))]
13:     proc  7 (Fueling_Mode) line 139 "sfd19" (state 2)       [fueling_mode = disabled]
14:     proc  7 (Fueling_Mode) line 139 "sfd19" (state 3)       [fueling_state = overspeed]
15:     proc  7 (Fueling_Mode) line 139 "sfd19" (state 4)       [goto]
16:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
17:     proc  7 (Fueling_Mode) line 177 "sfd19" (state 41)      [((speed==35))]
18:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
19:     proc  7 (Fueling_Mode) line 177 "sfd19" (state 42)      [count = 1]
20:     proc  7 (Fueling_Mode) line 177 "sfd19" (state 43)      [speed = 30]
21:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
22:     proc  6 (Speed_Change_Mode) line 124 "sfd19" (state 1)          [((count==1))]
23:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
24:     proc  6 (Speed_Change_Mode) line 124 "sfd19" (state 2)          [speed = 5]
25:     proc  6 (Speed_Change_Mode) line 124 "sfd19" (state 3)          [count = 2]
26:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
27:     proc  7 (Fueling_Mode) line 183 "sfd19" (state 48)
        [(((((((fueling_mode==disabled)&&(fueling_state==overspeed))&&(speed<30))&&(fail_
cnt<=1))&&(running_history==warmup)))]
28:     proc  7 (Fueling_Mode) line 184 "sfd19" (state 49)       [fueling_mode = running]
29:     proc  7 (Fueling_Mode) line 184 "sfd19" (state 50)       [fueling_state = warmup]
30:     proc  7 (Fueling_Mode) line 184 "sfd19" (state 51)       [goto]
```

```
 31:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 32:     proc  7 (Fueling_Mode) line 145 "sfd19" (state 11)
         [(((((fueling_mode==running)&&(speed<30))&&(fail_cnt<=1)))]
 33:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 34:     proc  7 (Fueling_Mode) line 150 "sfd19" (state 12)        [((speed==5))]
 35:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 36:     proc  5 (Time_Counter) line 115 "sfd19" (state 1)         [((t<=2))]
 37:     proc  5 (Time_Counter) line 115 "sfd19" (state 2)         [t = 3]
 38:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 39:     proc  7 (Fueling_Mode) line 150 "sfd19" (state 13)        [count = 0]
 40:     proc  7 (Fueling_Mode) line 150 "sfd19" (state 14)        [speed = 0]
 41:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 42:     proc  6 (Speed_Change_Mode) line 125 "sfd19" (state 5)          [((count==0))]
 43:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 44:     proc  6 (Speed_Change_Mode) line 125 "sfd19" (state 6)          [speed = 35]
 45:     proc  6 (Speed_Change_Mode) line 125 "sfd19" (state 7)          [count = 2]
 46:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 47:     proc  7 (Fueling_Mode) line 153 "sfd19" (state 16)        [(((fail_cnt>1)||(speed>30)))]
 48:     proc  7 (Fueling_Mode) line 153 "sfd19" (state 17)        [goto]
 49:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 50:     proc  7 (Fueling_Mode) line 139 "sfd19" (state 1)
         [(((fueling_mode==running)&&(speed>30)))]
 51:     proc  7 (Fueling_Mode) line 139 "sfd19" (state 2)         [fueling_mode = disabled]
 52:     proc  7 (Fueling_Mode) line 139 "sfd19" (state 3)         [fueling_state = overspeed]
 53:     proc  7 (Fueling_Mode) line 139 "sfd19" (state 4)         [goto]
 54:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 55:     proc  7 (Fueling_Mode) line 177 "sfd19" (state 41)        [((speed==35))]
 56:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 57:     proc  7 (Fueling_Mode) line 177 "sfd19" (state 42)        [count = 1]
 58:     proc  7 (Fueling_Mode) line 177 "sfd19" (state 43)        [speed = 30]
 59:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 60:     proc  6 (Speed_Change_Mode) line 124 "sfd19" (state 1)          [((count==1))]
 61:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 62:     proc  6 (Speed_Change_Mode) line 124 "sfd19" (state 2)          [speed = 5]
 63:     proc  6 (Speed_Change_Mode) line 124 "sfd19" (state 3)          [count = 2]
 64:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 65:     proc  7 (Fueling_Mode) line 183 "sfd19" (state 48)
         [(((((((fueling_mode==disabled)&&(fueling_state==overspeed))&&(speed<30))&&(fail_
cnt<=1))&&(running_history==warmup)))]
 66:     proc  7 (Fueling_Mode) line 184 "sfd19" (state 49)        [fueling_mode = running]
 67:     proc  7 (Fueling_Mode) line 184 "sfd19" (state 50)        [fueling_state = warmup]
 68:     proc  7 (Fueling_Mode) line 184 "sfd19" (state 51)        [goto]
 69:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 70:     proc  7 (Fueling_Mode) line 145 "sfd19" (state 11)
         [(((((fueling_mode==running)&&(speed<30))&&(fail_cnt<=1)))]
 71:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
 72:     proc  4 (Speed_Sensor_Mode) line  98 "sfd19" (state 6)  [((speed_sensor==norm))]
 73:     proc  4 (Speed_Sensor_Mode) line  98 "sfd19" (state 7)  [speed_sensor = 2]
 74:     proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)   [(1)]
```

```
 75:    proc  4 (Speed_Sensor_Mode) line 103 "sfd19" (state 11)
        [((speed_mode==speed_normal))]
 76:    proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)  [(1)]
 77:    proc  4 (Speed_Sensor_Mode) line 104 "sfd19" (state 12)      [speed_mode =
speed_fail]
 78:    proc  4 (Speed_Sensor_Mode) line 104 "sfd19" (state 13)      [fail_cnt = (fail_cnt+1)]
 79:    proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)  [(1)]
 80:    proc  4 (Speed_Sensor_Mode) line 105 "sfd19" (state 15)
        [((speed_mode==speed_fail))]
 81:    proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)  [(1)]
 82:    proc  2 (O2_Sensor_Mode) line  55 "sfd19" (state 1)    [((t>2))]
 83:    proc  2 (O2_Sensor_Mode) line  55 "sfd19" (state 2)    [o2_mode = o2_normal]
 84:    proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)  [(1)]
 85:    proc  7 (Fueling_Mode) line 160 "sfd19" (state 23)
        [(((((((fueling_mode==running)&&(running_history==warmup))&&(fueling_state==war
mup))&&(o2_mode==o2_normal))&&(speed<30)))]
 86:    proc  7 (Fueling_Mode) line 161 "sfd19" (state 24)      [fueling_state = normal]
 87:    proc  7 (Fueling_Mode) line 161 "sfd19" (state 25)      [running_history = normal]
 88:    proc  - (:never:) line  11 "./sfd131l.ltl" (state 1)
        [(((fueling_state==normal)&&(fail_cnt==1)))]
 89:    proc  7 (Fueling_Mode) line 150 "sfd19" (state 12)      [((speed==5))]
<<<<<START OF CYCLE>>>>>
 90:    proc  - (:never:) line  16 "./sfd131l.ltl" (state 7)  [(1)]
 91:    proc  4 (Speed_Sensor_Mode) line 106 "sfd19" (state 16)      [speed_mode =
speed_normal]
 92:    proc  4 (Speed_Sensor_Mode) line 106 "sfd19" (state 17)      [fail_cnt = (fail_cnt-1)]
 93:    proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)  [(1)]
 94:    proc  4 (Speed_Sensor_Mode) line 103 "sfd19" (state 11)
        [((speed_mode==speed_normal))]
 95:    proc  - (:never:) line  12 "./sfd131l.ltl" (state 3)  [(1)]
 96:    proc  4 (Speed_Sensor_Mode) line 104 "sfd19" (state 12)      [speed_mode =
speed_fail]
 97:    proc  4 (Speed_Sensor_Mode) line 104 "sfd19" (state 13)      [fail_cnt = (fail_cnt+1)]
 98:    proc  - (:never:) line  11 "./sfd131l.ltl" (state 1)
        [(((fueling_state==normal)&&(fail_cnt==1)))]
 99:    proc  4 (Speed_Sensor_Mode) line 105 "sfd19" (state 15)
        [((speed_mode==speed_fail))]
spin: trail ends after 99 steps
#processes: 8
                t = 3
                o2_sensor = 1
                press_sensor = 1
                throt_sensor = 1
                speed_sensor = 2
                fail_cnt = 1
                speed = 5
                o2_mode = o2_normal
                press_mode = press_normal
                throt_mode = throt_normal
                speed_mode = speed_fail
```

```
                    fueling_mode = running
                    fueling_state = normal
                    running_history = normal
                    count = 2
 99:      proc  7 (Fueling_Mode) line 150 "sfd19" (state 15)
 99:      proc  6 (Speed_Change_Mode) line 123 "sfd19" (state 9)
 99:      proc  5 (Time_Counter)
 99:      proc  4 (Speed_Sensor_Mode) line 106 "sfd19" (state 18)
 99:      proc  3 (Throttle_Sensor_Mode) line  77 "sfd19" (state 9)
 99:      proc  2 (O2_Sensor_Mode)
 99:      proc  1 (Pressure_Sensor_Mode) line  35 "sfd19" (state 9)
 99:      proc  - (:never:) line  15 "./sfd131l.ltl" (state 9)
8 processes created
```