University of L'Aquila,
Italy

Applying Model Checking for Verifying the Functional Requirements of a Scania's Vehicle Control System

Master Thesis - GSEEM (Global Software Engineering European Master)

School of Innovation, Design and Engineering Mälardalen University Västerås, Sweden

September, 2012

Shahid Ali
asd11006@student.mdh.se

Muhammad Sulyman
msn11013@student.mdh.se

Supervisors at Scania

Mattias Nyberg
mattias.nyberg@scania.com

Jonas Westman
jowestm@kth.se

Supervisor at University of L'Aquila

Giuseppe Dellapenna
giuseppe.dellapenna@univaq.it

Supervisor at Mälardalen University

Guillermo Rodríguez-Navas González
guillermo.rodriguez-navas@mdh.se

Examiner at Mälardalen University

Paul Pettersson
paul.pettersson@mdh.se

Preface/Acknowledgements

The authors of this master thesis desire to acknowledge the contribution of their supervisors from Scania and Mälardalen University (MDH) for their continuous guideline and support throughout the whole thesis process. Mattias Nyberg and Jonas Westman as supervisors from Scania facilitated the gathering information about Scania's vehicle control system used in this thesis work as case study, defining the scope of this master thesis, continuously checking the status of work to keep it according to schedule, provided guidance to make this work a success and helped making critical decisions during this work. Guillermo Rodríguez-Navas González as supervisor from MDH who helped in resolving both technical and administrative issue during this master thesis project and guided us to make this work according to MDH standards. Paul Pettersson as examiner from MDH finally approved this work, his contributions in this field and his published state of the art material in the literature truly helped us a lot throughout the entire thesis work. Gratitude to Mr. Jon Andersson for his help with managerial issues during our stay at Scania. Thanks to Giuseppe Dellapenna for the guidance to make this work approved in University of L'Aquila, Italy.

Abstract

Model-based development is one of the most significant areas in recent research and development activities in the field of automotive industry. As the field of software engineering is evolving, model based development is gaining more and more importance in academia and industry. Therefore, it is desirable to have techniques that are able to identify anomalies in the system models during analysis and design phase instead of identifying them in development phase where it is difficult to locate them and requires a lot of time, effort and resources to fix them. Model checking is a formal verification technique that facilitates to identify the defects in system model at the early stages of system development. There are a lot of tools in academia and industry that provide the automated support for model checking.

In this master thesis one of Scania's vehicle control system called Fuel Level Display System is modeled in two different model checking tools Simulink Design Verifier and UPPAAL. The requirements that are to be satisfied by the system model are verified in both tools. After verifying the requirements upon system model and checking the model against general design errors, it is established that the model checking can be effectively used for detecting the design errors in early development phases and can help developing better systems. Both the tools are analyzed depending upon the features supported. Moreover, relevance of model checking is studied with respect to ISO 26262 standard.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Background

Due to the increasing complexity in functionality of software applications, the popularity of model-based development has steadily increased over the years and has become a method that is used for both academic and industrial software development purposes. However, verification methods such as manual reviews, walkthroughs or inspections of models are still frequently used in the industry. Manual verification methods are time-consuming and it is very hard to verify that system satisfies all the specifications due to a large network of dependent sub-systems.

Furthermore, with the introduction of new standards, e.g., ISO 26262, requirements engineering has been incorporated into the product development process. Given that a set of requirements has been allocated to a system or sub-systems, the verification of the requirements is necessary. As a natural answer to this, formal model checking has emerged. Formal Model checking aims for automatic verification that a model meets a given set of specifications. There are tools and methods that are currently used for model checking during system development. The aim of this master thesis project is to investigate the possibility of using existing tools and methods for formal model checking in the development of industrial applications.

1.2. Problem Statement

Scania is one of the leading manufacturers of heavy trucks, buses, coaches and engines. A lot of effort, time, money and resources are required to fully test all the vehicle control systems deployed in their products. Scania wants to check if introducing model checking techniques in their systems can improve the quality of current testing process and can it be used as an alternative to testing, which will eventually help them to save some of the resources which are currently being consumed for testing all the vehicle control systems. Furthermore it is also required to investigate the relevance of model checking with respect to the latest ISO 26262 standards, and to analyze different model checking tools based upon their features for model checking.

1.3. Goals

This master thesis project can be divided into following different sub-goals
1. Analyze the academic and industrial world to find initial set of relevant tools/methods for formal model checking.
2. Narrow down the initial set of methods/tools down and finalize two suitable model checking tools for modeling based upon their coverage, manageability, degree of maturity, and applicability.
3. Choose a well-documented, non-trivial best-practice example at Scania to be modeled by using the selected model checking tools. Selection of the vehicle control system is based upon the documentation, test data, requirements and state machines available for the system at Scania.
4. Model the best practice selected vehicle control system in finalized model checking tools and verify the system model against the specified requirements. Also verify the system model against general design errors.
5. Evaluate the results with respect to industrial relevance and usefulness in terms of compliance with ISO 26262. Analyze the model checking tools used during this master thesis project based upon the features offered by the tools.

1.4. About Scania Group

This master thesis is performed at Scania Group situated in Södertälje, Sweden in REPA department. Scania was founded in 1891 and presently is one of the leading manufacturers of heavy trucks, buses, industrial and marine engines. Scania has branches in around 100 countries all over the world. Overall Scania has more or less 37,500 people employed in different departments. Out of 37,500 employees 3,300 are working in research and development department in Södertälje, Sweden [3].

Disclaimer:
All the numerical values mentioned in the requirements of fuel level display system are arbitrary values and do not represent the actual values in Scania fuel level display system.

2. Model Checking

With the evolution of modern technologies dependability of organizations and individuals over software applications is remarkably increased. In many situations these software applications are critical for the safety of individuals and the environment in which they are operating. So it has become a strict requirement that these applications must be fail safe and error free. But due to the complex nature of software, this requirement cannot be achieved 100%. There exist different testing strategies but it is very expensive, time consuming and difficult to make a complete and comprehensive list of testing scripts. To cope with these limitations formal verification technique called model checking can be used. By using formal verification techniques verification activity can be initiated very early in the design process which makes verification activity more effective and less time consuming [1].

Models describing the behavior of the system in a precise and clear manner are used by the model based verification techniques. Correct modeling of the system behavior can discover the inconsistencies in system behavior with respect to its specifications [1].

In model checking all possible systems scenarios and states are explored in a brute force manner just like the chess program that examines all possible moves. By using this way of verification it can be proved that the system truly satisfies certain properties. Certain errors which remain undiscovered during simulation, testing and emulation can be potentially discovered by using model checking. As the name implies, model checking checks the model of the system that is an abstract and high level description of the system, it does not checks the actual program of the system. It is a great challenge for the developers to make a model of the system that truly represents the actual system. The main obstacle in model checking is the abstraction of the software application. It states that during abstracting the real world problem, the important and critical aspects of the system model should not be left out. The abstraction should be comprehensive enough to contain all the critical aspects of the system. Otherwise, the model checker will be useless if it cannot explore the states which could possibly give rise to errors and compromise the safety in some critical software applications [1].

Model checking is a software verification technique that uses model of the software application and tries to verify certain properties on the software model during execution. Precisely it can be said that model checking process is basically composed of designing system model, properties definition, running the model checker and analyzing the results [1].

2.1. The Model-Checking Process

Following are the different phases in model checking process [1]

1) Designing system model:
Model description language provided by the model checking tool under hand is used to model the system under consideration. Designing a system model usually involves making finite state machines which specify desired behavior of the system that can be defined in terms of variables, initial values for the variables, environmental assumptions, and description of conditions under which values of variables will change [1].

2) Properties definition:
Properties of the system to be verified against the system model are defined by using some temporal logic constraints for example linear temporal logic (LTL) or computation tree logic (CTL). Each model checking tool supports different kind of property specification language [1].

3) Running the model checker:
Model checker is executed to check the validity of the defined properties against the system model. Results of the model checker are stored to be analyzed at the later stages [1].

4) Analyzing the results:
Results of the model checker are analyzed. If the current property for which model checker was executed is satisfied by the system model then check the next property if there is any. And if the property is violated by the system model then analyze the counter example or error path generated by the model checker and based upon that refine the system model, design of system or may be the property. Repeat the entire above procedure until all the properties are satisfied by the system model. During model checking lot of states are generated by the model checker so there is a possibility of getting out of memory error. In such case either reduces the system model or use more sophisticated techniques for model checking [1]. Figure 1 presents the overall architecture of model checking.

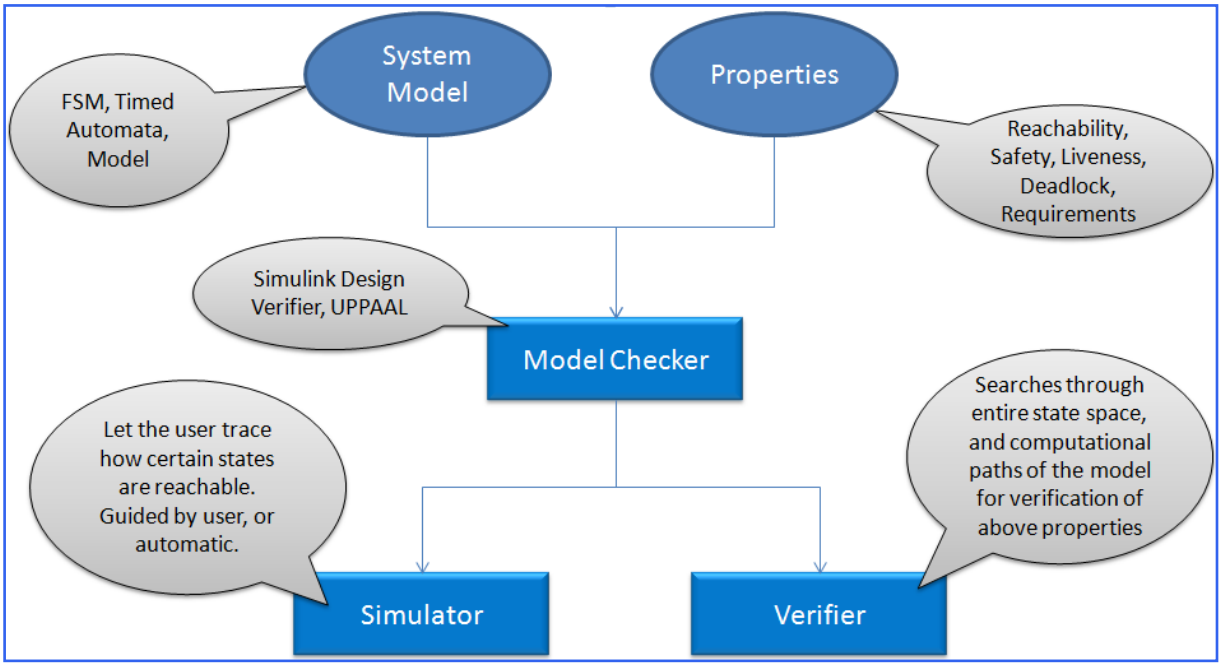

Figure 1: Model checking architecture.

One problem that is normally faced during model checking is the state space explosion problem. It is a very critical issue to examine the largest possible state spaces that can be handled with available resources (processor and memory). State-of-the-art model checkers can handle 10^8 to 10^9 states with explicit state-space enumeration. By employing efficient algorithms and customized data structures 10^{20} to 10^{476} states can be handled [1].

2.2. Types of model checkers

Based on the model specification and state space, there exist different types of model checkers. Mainly the model checkers can be divided into two categories
1) Explicit model checkers.
2) Implicit model checkers.

Explicit model checkers construct a searchable representation of the design model and store a representation of each state visited [1]. Implicit model checkers also called symbolic model checkers use logical representations of sets of states (such as binary decision diagrams) to describe the regions of model state space that satisfy the properties being evaluated. Such compact representations generally allow symbolic model checkers to handle a much larger state space than explicit model checkers [1].

Some recently developed model-checkers use satisfiability modulo theories (SMT) solvers for reasoning about infinite state models containing real numbers and unbounded arrays. These model checkers use a form of induction over the state transition relation to automatically prove that a property holds over all executable paths in a model. While these tools can handle a larger class of models, the properties to be checked must be written to support inductive proof [1].

2.3. Model checking in System Development Lifecycle

Following figure 2 represents the timing for conducting model checking activity in classic waterfall model, during system development life cycle. Model checking can be categorized into two categories classic model checking and modern model checking. Classic model checking can be performed during analysis, design or before the coding activity in system development lifecycle. And improvements can be made in the system model based upon the outcomes of model checking. Modern model checking is carried after the coding activity. This can complement the testing activity and improvements can be made in the model depending upon the outcomes of model checking activity [13].

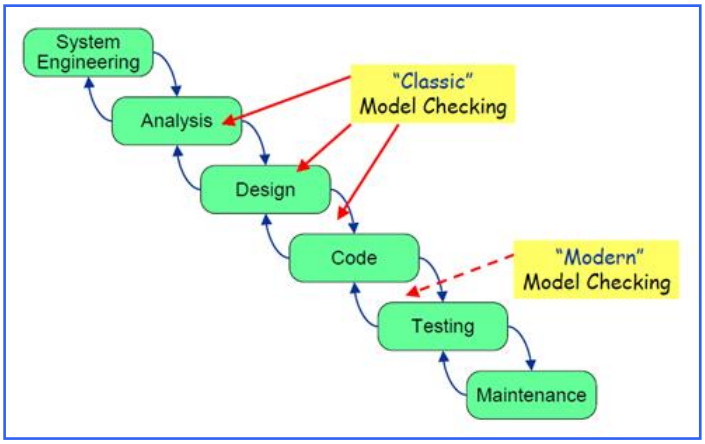

Figure 2: Model checking in classical waterfall model

2.4. Model checking in the fuel level display system

In case of Scania's vehicle control system fuel level display system, this system is already developed, tested and working in their trucks. The main purpose of applying model checking to this vehicle control system is to investigate the validity of the design by using model checking techniques and also to see whether it's feasible to replace current verification techniques used in Scania with model checking. Or if not completely replacing the current verification style then how model checking can help improving the current verification activity. Because model checking is supposed to complement the testing, it's not an alternative of testing.

Current verification methods employed in Scania are time-consuming and due to the large network of dependent sub-systems it can be hard to verify that a system meets all the specifications. By using model checking it can be ensured that almost 100% test coverage has been achieved and it can be justified with proof from model checking tools which is hard to perform in case of manual testing techniques.

Since the fuel level display system application is already developed, tested and is working in actual products which mean that the system is in maintenance phase so the model checking activity in this case is after the testing activity during maintenance activity as shown in the following figure 3. But since the company is open to the changes in the design of the system depending upon the out-come of the model checking activity so this model checking activity can also be related to the analysis and design phases and also before the coding phase as described by the classical waterfall software development model of Pressman 1996.

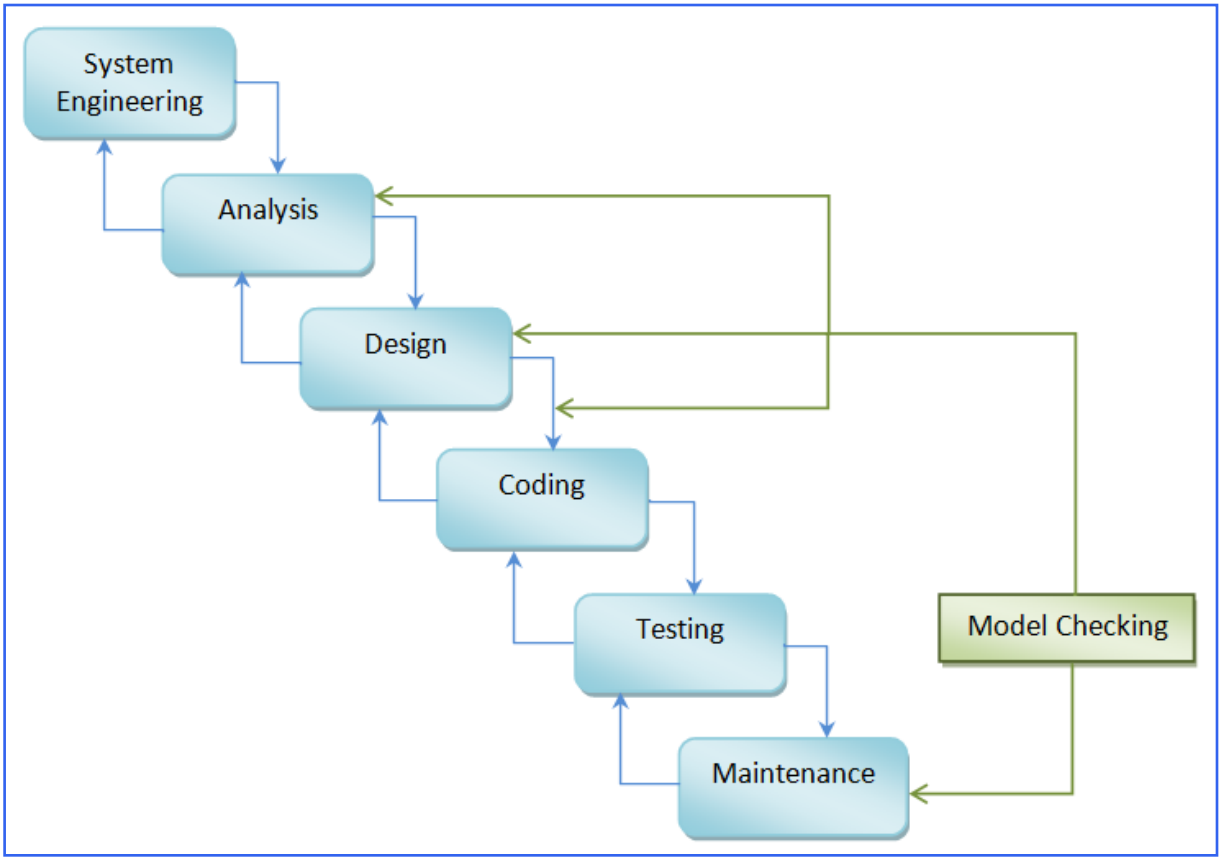

Figure 3: Model checking in case of fuel level display system.

3. Systematic Literature Review:

Systematic literature review for searching the existing model checking tools was initiated on 17 March, 2012. Springerlink, ACM Digital Library and Google Scholar search engines are used to search for the existing model checking tools. Systematic literature review was started with following two search strings.

1. "Model checker".
2. "Model checking tool".

Following table 1 shows the results that are obtained on the basis of two search strings. This table differentiates the number of results based upon the title and abstract search, and whole body search.

Search Engine	Search String	Title and Abstract	Whole Paper/body
Springerlink	Model Checking Tool	966	46955
Springerlink	Model Checker	1159	11297
ACM Digital Library	Model Checking Tool	Title: 9 Abstract: 396	17,167
ACM Digital Library	Model Checker	Title: 36 Abstract: 463	5806
Google Scholar	Model Checking Tool	Title: 133	1220000
Google Scholar	Model Checker	Title: 631	79200

Table 1: Initial search results based on two search strings.

Based upon the initial research results it was discovered that the first search string "Model Checker" is returning most relevant results as compare to the string "Model checking tool". Therefore the string "Model Checker" was finalized for this research.

Search Engine	Search String	Title and Abstract	Whole Paper/body
Springerlink	Model Checker	1159	11297
ACM Digital Library	Model Checker	Title: 36 Abstract: 463	5806
Google Scholar	Model Checker	Title: 631	79200

Table 2: Search results against search string "model checker".

After reading the title, abstracts and introduction 96 papers are considered relevant. Then after reading the whole papers only 37 papers are found relevant, each of them presented at least one model checking tool. Eventually it is discovered according to the systematic literature review that there are 54 model checking tools present with different features and functionality that they support. Therefore these 54 tools are considered the currently existing tools for this systematic literature review about model checkers. Since the main purpose of this master thesis is not to study about the existing model checking tools but to perform model checking upon fuel level display system of Scania, that's this systematic literature review is kept short and brief.

3.1. Inclusion Exclusion Criteria

Inclusion Criteria:

1. All papers that present at least one model checking tool are included.
2. All the papers about model checking tools are included.
3. All papers generally talking about model checking techniques and methodologies are included.
4. All papers about how model checking tools and techniques work are included.
5. Books, thesis, articles, journals, conference proceedings and technical reports related to model checking are included
6. No specific time limit is enforces on the search. All relevant articles until the search date are included.

Exclusion Criteria:

1. All papers which are not in English language are excluded.
2. All papers about how model checking tools are built are excluded.

3.2. Model Checking Tools

Different types of temporal logics are used by the model checking tools to express the properties of system as logical formulas to be verified on the system model. Most of model checking tools use either linear time or branching time logics. Properties that can be expressed in either of these logics they can also be expressed in the other one as well. But some of the tools use different types of logic to formulate the properties.

Linear Time:
[16] Linear time also called linear temporal logic or linear-time temporal logic (LTL) is a modal temporal logic with modalities that refer to time. In LTL formulas future can be encoded. For example a condition will finally be true; a condition will be true until another fact becomes true, etc.

Branch Time:
[16] Branch time also called computation tree logic (CTL) is branching-time logic. Its model of time is a structure like tree where the future is not determined. There are different paths in the future that can be followed; any one of them can be selected as future path. In CTL it can be specified that when an initial condition is true then all possible executions of a program avoid some unwanted state or condition.

Other:
There are model checking tools which use different type of logics to formulate the properties of the system to be verified as mentioned in the table below in the column named as other, next to the column named linear time.

Real Time:
Real time systems are the systems that operate under time sensitive environment and provide guaranteed response within strict time constraints. Real time column shows that the tool supports the modeling, verification and analyses of real time systems.

Probabilistic:
Probability is an important component in the design and analysis of complex systems. Probabilistic model checking is a formal verification technique for modeling and analyzing the systems.

Probabilistic column in the table below shows the tools that support the modeling and analysis of systems that exhibit probabilistic behavior.

Hybrid:
Hybrid column shows that the tools marked as hybrid supports both real time and probabilistic model checking approaches.

GUI:
Graphical user interface (GUI) column shows that which of the tools provide graphical user interface for modeling the system and for formulating the properties of the system to be verified.

Availability:
Availability column shows that under what conditions the tools is available for use. Availability of the model checking tools can be divided into three types

1. Free - Free means that the tools are freely available to be downloaded and to be used for all kinds of users.
2. Free Under condition - Free under condition means that tool is available to be downloaded and used for the academic purposes but if someone wants to use it for the commercial purposes then licenses are required.
3. Commercial - Commercial means that license must be acquired for downloading and using the model checking tool.

Following table presents the model checking tools, and features supported by them, which exist until now in academia and industry [2].

Short Name	Full Name	Linear Time	Branch Time	Other	Real Time	Probabilistic	Hybrid	GUI	Availability
Alpina	Alpina			Reachability				✓	Free
APMC	Approximate Probabilistic Model Checker	✓	✓			✓		✓	Free Under Condition
ARC	AltaRica Checker			Modal mu-calculus					Free Under Condition
Bandera	Bandera	✓						✓	Free
Blast	Berkeley Lazy Abstraction Software verification Tool	✓							Free
Cadence SMV	Cadence SMV	✓	✓					✓	Free Under Condition
Cascade	Cascade			Abstract syntax tree		✓			Free
CADP	Construction and Analysis of Distributed Processes		✓			✓		✓	Free Under Condition
CWB - NC	The Concurrency Workbench of New Century	✓	✓		✓			✓	Free Under Condition
DBRover	DBRover	✓			✓			✓	Commercial
DiVinE Tool	Distributed Verification Environment -- Tool Set	✓							Free
DREAM	Distributed Real-time Embedded Analysis Method		✓	Discrete event simulator	✓				Free
Edinburgh CWB	Edinburgh Concurrency Workbench		✓						Free Under Condition
Expander2	Expander2		✓				✓	✓	Free
Fc2Tools	Fc2Tools and Autograph		✓						Free
GEAR			✓	Modal mu-calculus				✓	Free Under Condition
HSolver	HSolver			Safety			✓		Free
HyTech		✓	✓		✓		✓		Free

IF	IF Toolbox		✓		✓			✓	Free
INA	Integrated Net Analyzer		✓	Deadlock-freedom, Reachability, Coverability, Invariants, Structural Analysis					Free
JPF	Java PathFinder	✓							Free
KRONOS	KRONOS		✓		✓				Free
LTSA	Labelled Transition System Analyzer	✓						✓	Free
MCRL	MCRL toolset		✓						Free
mCRL2	mCRL2 toolset	✓	✓		✓			✓	Free
Mocha	Mocha	✓	✓	ATL				✓	Free
Moped	Moped	✓							Free
MRMC	Markov Reward Model Checker		✓		✓	✓			Free
mucke	mucke		✓	Mu-calculus					Free Under Condition
NuSMV	NuSMV: a new symbolic model checker	✓	✓					✓	Free
PAT	Process Analysis Toolkit	✓			✓	✓		✓	Free
PEP	PEP - Programming Environment based on Petri nets	✓	✓		✓			✓	Free
PRISM	Probabilistic Symbolic Model Checker	✓	✓			✓		✓	Free
PROD	PROD	✓	✓	Deadlock-freedom, Reachability					Free
PVS	Prototype Verification System		✓		✓		✓	✓	Free Under Condition
Reactis Tester	Reactis Tester			Automatic Test Case Generation			✓	✓	Commercial
SGM	State-Graph Manipulators		✓		✓			✓	Free Under

									Condition
SMCWWI	Simple Model Checker With Web Interface		✓					✓	Free
SPIN	Simple Promela Interpreter	✓						✓	Free Under Condition
Statestep	Statestep			Safety (non-reachability), explicit completeness				✓	Free Under Condition
STeP	Stanford Temporal Prover	✓			✓		✓	✓	Free
TAPAAL	TAPAAL		✓		✓			✓	Free
TAPAs	Tool for the Analysis of Process Algebras		✓					✓	Free Under Condition
Temporal Rover	Temporal Rover	✓			✓				Commercial
The Kit	The Model-Checking Kit	✓	✓	Deadlock-freedom, Reachability					Free
TIMES	A Tool for Modeling and Implementation of Embedded Systems		✓		✓			✓	Free Under Condition
TRON	Uppaal TRON		✓	Timed testing using model-checking techniques	✓			✓	Free Under Condition
Truth	Truth		✓						Free
TwoTowers		✓				✓		✓	Free Under Condition
UPPAAL	UPPAAL Toolkit		✓		✓			✓	Free Under Condition
VeriSoft	VeriSoft	✓						✓	Free
VIS	Verification Interacting with Synthesis	✓	✓						Free
Ymer	Ymer		✓		✓	✓			Free Under Condition
[mc]square	[mc]square		✓					✓	Free Under Condition

Simulink Design Verifier (Prover)	Simulink Design Verifier (Prover)			SMT – Satisfiability modulo theories				✓	Commercial
CMurphi	CMurphi	✓							Free
RT-SPIN	Real Time - Simple Promela Interpreter	✓			✓			✓	Free Under Condition

Table 3: Model checking tools and their features.

3.3. Why Simulink Design Verifier and UPPAAL?

During this systematic literature review fifty five tools are discovered that can be used for model checking. For this master thesis project Simulink Design Verifier and UPPAAL are selected to perform the activity of model checking on Scania's vehicle control system.

The main reason for the selection of Simulink Design Verifier is that Scania uses Simulink to build the models of their vehicle control systems. It is desirable, from Scania's point of view, to determine whether the existing Simulink models of Scania's vehicle control system can be used in the future for model checking by Simulink Design Verifier or not. Building the system model from scratch in any other model checking tool requires a lot of time and resources and if the existing Simulink models can be used for model checking then a lot of such effort can be saved.

The vehicle control system that is selected for model checking for this master thesis is a real-time system. UPPAAL is a model checking tool particularly suitable for modeling, simulation, validation and verification of real-time systems. It is also one of the most used, updated and continuously evolving tool. Hence, after discussion with supervisors from Scania, UPPAAL is selected as the second tool for model checking during this master thesis project.

4. Simulink Design Verifier

Simulink Design Verifier identifies design errors, generate test vectors, and verify designs against requirements. It uses formal methods to identify hard to find design errors in models without requiring extensive tests or simulation runs. Design errors that can be detected by Simulink Design Verifier include dead logic detection, integer overflow identification, division by zero, and violations of design properties and assertions [8].

Model blocks that contain errors are highlighted in Simulink Design Verifier and it is also possible to highlight the blocks that do not contain any error. Signal range boundaries are calculated and test vectors are generated by Simulink Design Verifier for reproducing the errors later during simulation. Simulink Design Verifier also facilitates to perform analysis of model in simulink environment. Design and requirements can be validated in early phases without generating the code. Verification and validation activities can be conducted in the whole design phase. Model analysis supports simulation and allows the users to use simulation results as inputs to analyze the models by using formal methods. Discrete-time subset of Simulink and State flow are also supported by Simulink Design Verifier. State flow is typically used in embedded control designs [8].

Following are the key features provided by Simulink Design Verifier [8]

- Simulink Design Verifier uses Polyspace and Prover Plug-In as formal analysis engines.
- Detection of design errors like integer and fixed-point overflows, division by zero, dead logic detection, violations of design properties and assertions violations.
- Simulink Design Verifier provides blocks and functions for modeling functional requirements and safety requirements.
- Test vector can be generated from functional requirements. Model coverage is ensured by model coverage objectives which include condition, decision, and modified condition/decision coverage (MCDC).
- Property satisfaction by identification of failure locations for analysis and debugging.
- Simulink Design Verifier also provides model support for fixed-point and floating-point models. Support for floating point really makes a difference as compared to UPPAAL that does not support the floating point models.

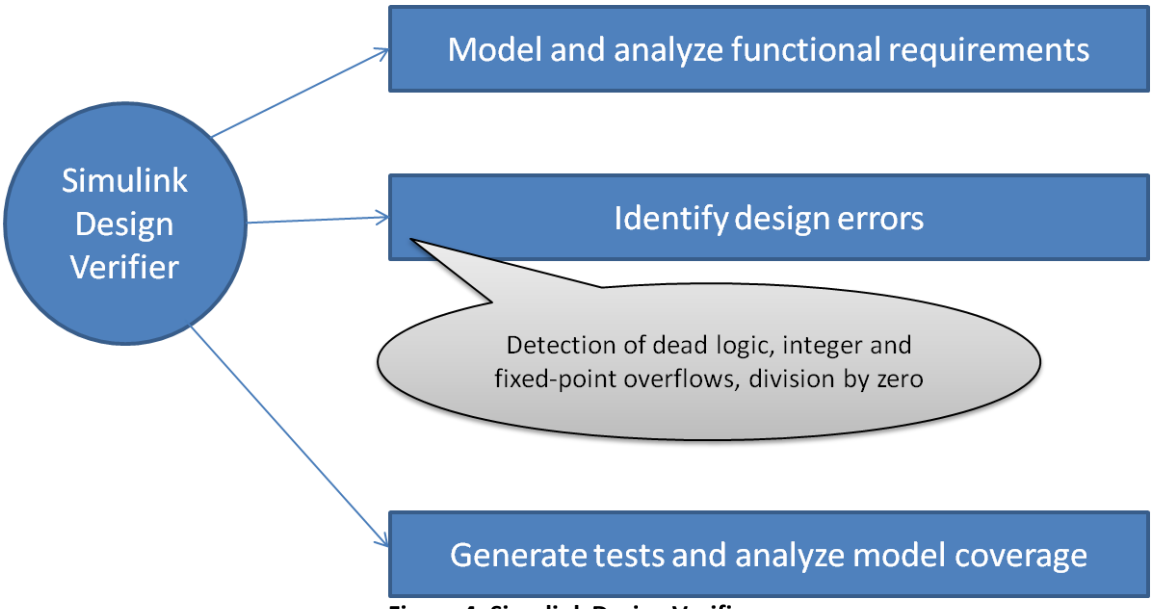

Figure 4: Simulink Design Verifier.

Formal analysis techniques manufactured by Prover Technology called Prover Plug-In and Polyspace formal analysis engine created by MathWorks are used by Simulink Design Verifier for formal analysis. Both of these formal analysis techniques depend upon rigorous mathematical procedures for searching the possible execution paths in the model for test cases generation and counter examples generation. In traditional testing methods test scenarios and expected results are presented with concrete data values but it's not the same in formal analysis techniques. In formal analysis techniques user can work with models of system behavior rather than concrete data values. Models of test scenarios and verification objectives that describe expected and unexpected system behaviors can be included in system behavior models. Formal analysis that is conducted by using such models complements simulation and provides a deeper understanding of system design [8].

4.1. Error Detection Using Formal Methods

Simulink Design Verifier can discover whether certain scenarios can occur under specific conditions or not. This information can be used to improve the design, refine the requirements, guide the simulation for debugging, verification and validation. Simulink Design Verifier supports the detection of design errors like integer overflow, division by zero, dead logic, and assertion violations [10].

4.1.1. Detecting Integer Overflow and Division by Zero

Simulink Design Verifier provides design error-detection mode to discover integer overflow and division by zero errors. This analysis is automatically performed which does not require the user to provide any additional inputs. Permitted ranges for all type of signals on every block are provided to direct the user in identifying the root cause of error. Results can be viewed in the form of model or in an HTML report format after the analysis is finished. Blocks are marked as green, yellow, or red in the model. Blocks marked as green shows the blocks that are proven to be unable to cause any integer overflow or division by zero errors. Blocks are marked with yellow when analysis cannot produce a conclusive result or when the time limit for the analysis is exceeded than expected. When an error is detected in the model execution sequence, all sub-blocks in the path that can cause integer overflow and division by zero are marked with yellow. Blocks marked with red shows the blocks that have integer overflow or divide by zero errors. For the blocks marked with red Simulink Design Verifier generates test cases that can reproduce the problem during simulation or testing. Test case can be invoked and simulation can be executed directly within simulink [10].

4.1.2. Detecting Dead Logic

Test-generation mode in Simulink Design Verifier is used to detect dead logic in model, which identifies the model objects that are either outdated or the model objects that are proven to remain inactive during the execution process. Dead logic can be caused by a design error or a requirement error. When code is generated then, if dead logic is present in the system model it leads to dead code. It is very difficult to detect dead logic by testing only in simulation because even after running many simulations, it can be difficult to state that a specific logic or portion of model remains inactive. At the end of the analysis of test-generation the model is colored according to the test-generation criteria. Portions of model that have dead logic are marked as red and the parts of model that have logic which can be completely activated in simulation are marked as green. Test cases are generated by Simulink Design Verifier to reproduce the dead logic in simulation [10].

4.1.3. Detecting Assertion Violations

To detect assertion violations Simulink Design Verifier provides assertion violation detection setting in property-proving mode. Simulink Design Verifier checks if there is any valid scenario that can

trigger some assertions during simulation by remaining within the number of time steps mentioned in the analysis settings. All the assertions that can be violated by any valid scenario are marked with red. Test vector is generated for the triggered assertion. Some assertions are available in simulink by default and Simulink Design Verifier also provides extra blocks for defining additional constraints for analysis that empower the user to thoroughly analyze design behavior and detect design flaws prior to running the simulation [10].

4.2. Verification of designs against requirements

Functional requirements of systems are traditionally clear statements about expected behaviors of the system. It can also be mentioned in the requirements about the behaviors that a system must never demonstrate. Scenarios that must never be shown by the system are referred to as functional safety requirements [11].

In order to formally check that the system design behaves according to the functional requirements, the requirement statements which are normally written in plain human language are first needed to be translated into the language that is understandable by the formal analysis engine under consideration. Simulink Design Verifier allows the users to state formal functional requirements using simulink blocks, MATLAB functions and Stateflow. Every requirement which is created in simulink for verification has one or more objectives associated with it. These verification objectives are eventually used to investigate if the system design fulfills the functional and safety requirements or not. Simulink Design Verifier provides a block library that includes blocks and functions for defining test objectives, proof objectives, assertions, constraints, and dedicated set of temporal operators for modeling of verification objectives with temporal aspects [11].

Simulink Design Verifier provides two blocks to specify property proofs: 1) Proof Objective 2) Proof Assumption. Proof objective define the values of a signal to be proved and proof assumption ensure constrains on the values of signals during a proof.

Figure 5 describes the property verification architecture that is followed in Simulink Design Verifier for verifying the properties. This property proving architecture is followed for verifying the properties of fuel level display system of Scania during this master thesis project. First of all, inputs to the system are defined and then assumptions sub-model defines the assumptions on the input values. Assumptions defined on inputs in assumptions sub-model are used by both the system model and the verification sub-model. Verification sub-model also uses the output from system model for property proving. Verification sub-model can have multiple objectives associated to it that the system is expected to satisfy.

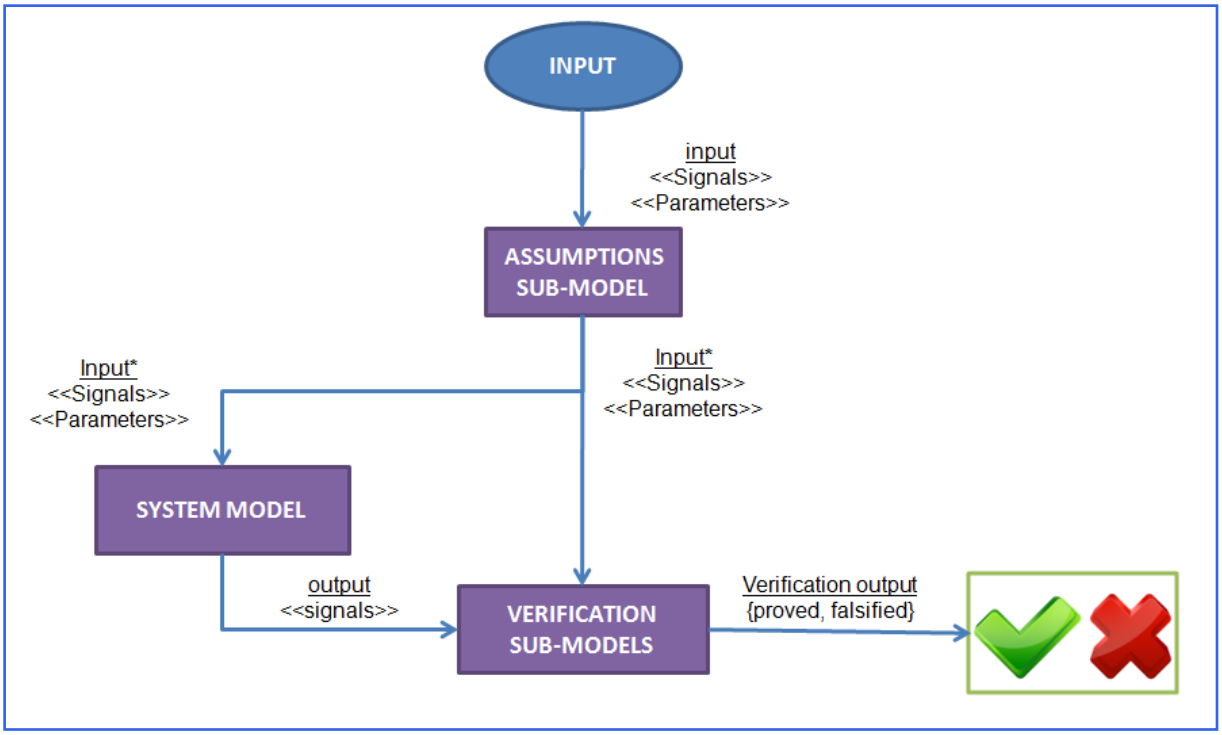

Figure 5: Architecture for property verification in Simulink Design Verifier.

Once requirements and verification objectives are identified and captured in the verification model they can be used to illustrate the correctness of system design using formal methods. Test Objective blocks and MATLAB functions can be used for defining test objectives. During test generation Simulink Design Verifier will try to find a valid test case that meets the specified test objectives. If there is a situation that certain objective can never be met then the system design cannot perform the desired feature against the specified set of analysis constraints [11].

In order to test the correctness of system design against safety requirements, proof objective blocks and MATLAB functions are used for specifying proof objectives. Simulink Design Verifier examines all possible input conditions that can cause undesired behavior during analysis and then report the results. System design can be proven valid or it can violate the functional safety requirements for a given proof objective. Whenever a violation of proof objective against requirements is detected, test vector is generated by Simulink Design Verifier that can be used later to demonstrate the violation during simulation [11].

Algorithms and logic developed in Simulink and stateflow models to generate test cases and parameters is analyzed by Simulink Design Verifier. This kind of analysis is required by industry for developing systems with high integrity level and to meet the objectives set by the standards. To make sure the structural test coverage criteria for test generation condition, decision and modified condition/decision coverage (MC/DC) techniques are used by Simulink Design Verifier [12].

TÜV SÜD has certified Simulink Design Verifier. Simulink Design Verifier is certified to be used in development process of systems. Processes of system development must comply with ISO 26262, IEC 61508, or EN 50128 standards [12].

5. UPPAAL

UPPAAL is a model checking tool for modeling, simulation, validation and verification of real-time systems that are modeled as networks of timed automata. Finite state machines with time clocks are known as timed automata. UPPAAL does the verification and validation of the modeled systems by using two techniques. 1) constraint-solving 2) on-the-fly techniques. In UPPAAL validation is done by graphical simulation and verification is done by automatic model-checking. In simulation modeled system is executed interactively and is observed that whether the system satisfy the expected behavior or not. UPPAAL uses finite state automata extended with clock and data variables. UPPAAL uses a subset of TCTL (timed computation tree logic) to model the requirements to be verified on the system model. UPPAAL facilitates with the features of counter example generation and counter example visualization [7], [17].

Typical application areas for which UPPAAL is considered best include real-time controllers and communication protocols in particular, those where timing aspects are critical. It is designed mainly to check invariant and reachability properties which are done by exploring the state-space of a system. Two main parts of UPPAAL are graphical user interface and a model checker engine. Graphical user interface is implemented in Java language and is executed on the user end. Model checker engine is developed is C++ and is also executed on the user work station but UPPAAL offers the flexibility of running the engine on a separate machine that is more powerful and can be referred as server. Two main advantages of UPPAAL are efficiency and easy to use. Model checker engine applies on-the-fly searching technique in combination with the symbolic technique which makes the verification problem reduced to the problem of solving simple constraints system [7].

UPPAAL can also generate a problem trace automatically, that can be used to diagnose the problem and can also be used to explain why a property is or is not satisfied by the described system. The traces can be visualized graphically by using the simulator provided by UPPAAL. Three main components of UPPAAL are 1) editor, 2) simulator and 3) verifier. It is freely available for academic purposes and it also has a commercial version available for industrial use which requires purchasing license for commercial use. It is available for Windows, UNIX and related platforms. UPPAAL is a joint venture of Department of Information Technology at Uppsala University, Sweden and Department of Computer Science at Aalborg University, Denmark. Initial version of UPPAAL was released in 1995 since then it has been continuously evolving and updating [17].

6. Fuel Level Display System

This section contains the detailed description for fuel level display system of Scania. Functional requirements that are to be verified on the system model of fuel level display system during model checking activity are described. The overall architecture of the vehicle control system is also explained in this section in detail.

6.1. Background

In the beginning the plan was that different existing vehicle control systems in Scania will be studied and one of them will be selected for model checking based upon its documentation, test data, requirements and state machines available. Later-on the supervisors at Scania proposed to use Fuel Level Display System which is a well-documented and best-practice example at Scania. This system has already been used in different research projects and master thesis as case study.

Basic goal of the fuel level display is to keep the vehicle from running out of fuel. Current fuel level is continuously calculated and displayed to the driver on gauge. Driver may not monitor the fuel level gauge frequently so a warning is useful. A low fuel level warning is introduced to make the driver aware of the fuel level if fuel level in the fuel tank drops below a predefined limit and a refill is needed.

6.2. Purpose

This system intends to show the fuel level for all vehicles independent of vehicle and fuel type.
A low fuel level warning is also present on vehicles, indicating if the fuel level is low as compared to a predefined limit.

6.3. User interaction

The fuel level is measured and presented to the driver in the instrument cluster. If the fuel level is below a predetermined level a warning is presented in the instrument cluster. The low fuel level warning could be used on both trucks and buses. Fuel level display system can be divided into following two sub categories

1. Fuel level estimation: It estimates the current fuel level in the fuel tank.
2. Low fuel level warning: it creates a warning if the fuel level in the tank is considered to be low as compared to a predefined value.

There are two documents in Scania listing all the functional requirements related to fuel level estimation and low fuel level warning. AE201 contains the functional requirements for fuel level estimation and AE202 contains the functional requirements for low fuel level warning.

The fuel level sensor comes in different types depending on the fuel tank, fuel injection system and electrical characteristics of the fuel sensor.
Following are the different steps which are performed in fuel level display system

1. Read fuel height
2. Calculate fuel level
3. Send fuel level
4. Detect and display low fuel level
5. Display fuel level
6. Calculate fuel level multiple sources

6.4. Configuration Parameters

There are following 4 different types of variants for fuel level display system

1. Truck with fuel engine
2. Bus with fuel engine
3. Bus with gas engine
4. Truck with gas engine

There are multiple types of sensors and multiple sizes of fuel tanks that are used in fuel level display system. Total fuel level can be either calculated by COO (Coordinator) or received from CAN (Controller area network). For the sake of simplicity and clarity in this master thesis only one type of system variant is considered that is truck with fuel engine. One type of sensor and one size of fuel tank is considered and the variant where total fuel level is calculated by COO is used.

Variant 1	**Truck with fuel engine**
fuelLevelSensorParam	15 (Wema-general : short)
fuelTankSizeLeft	13 (450 liter volume: General)
fuelTankSizeRight	Same as for fuelTankSizeLeft
fuelLevelTotalParam	10 (Total fuel level calculated by COO)

Table 4: Parameters for fuel level display system.

6.5. System Architecture

Following figure 6 shows the components at abstract level that are involved in the fuel level display system. Fuel level sensor is mounted inside the fuel tank. ECUs (electronic control units) that are involved in fuel level display system are COO (coordinator), EMS (engine management system) and ICL (instrument cluster). Gauge is not a separate component it is a part of ICL.

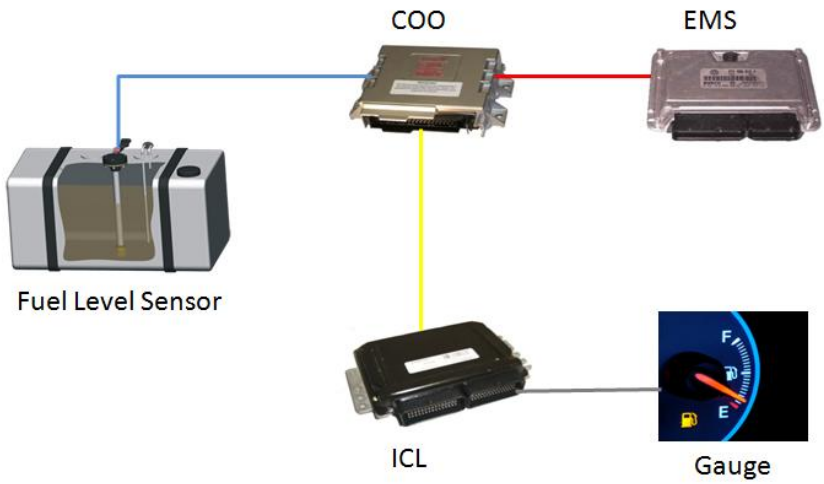

Figure 6: Fuel level display system - components involved.

Figure 7 shows the information flow between different components involved in fuel level display system. Fuel Rate (RTDB_FUEL_CONSUMPTION_RATE_E) comes from EMS through red CAN bus to RTDB (real time data base). Value from fuel level sensor (RTDB_FUEL_LEVEL_E) inside the tank comes through blue CAN bus to RTDB. COO uses the values from RTDB through different buses for performing different operations.

COO reads fuel level sensor value (RTDB_FUEL_LEVEL_E) on blue CAN bus, parking brake value (RTDB_PBRAKE_APPLIED_HIGH_E) on red CAN bus, fuel Rate (RTDB_FUEL_CONSUMPTION_RATE_E)

on red CAN bus, old fuel volume (RTDB_FUEL_LEVEL_E) on yellow bus from RTDB. COO places fuel level (RTDB_UP_FUEL_LEVEL_TOT_E) and low fuel level indication (RTDB_UP_LOW_FUEL_LEVEL_IND_E) in RTDB through yellow CAN bus.

ICL uses the fuel level (RTDB_UP_FUEL_LEVEL_TOT_E) and low fuel level indication (RTDB_UP_LOW_FUEL_LEVEL_IND_E) from RTDB through yellow CAN bus to indicate the fuel level and the low fuel level warning respectively.

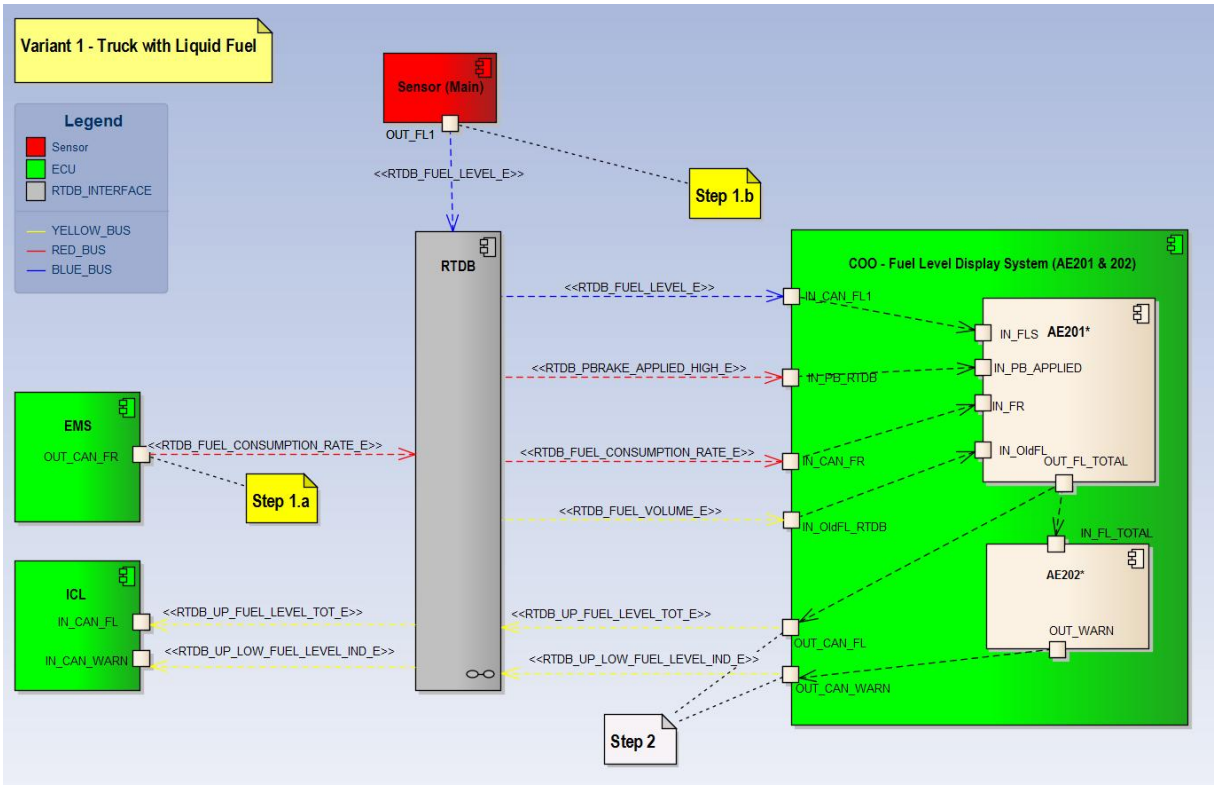

Figure 7: Fuel level display system - information flow.

Requirements mentioned in document AE201 and AE202 are implemented inside the COO. * With AE201 and AE202 represent that here only subsets of these requirements that are mentioned in the documents are implemented. All the requirements mentioned in table 5 are implemented inside COO. Rests of the requirements are either background requirements or assumptions about the system. Only the requirements that are mention in the table 5 are considered relevant for model checking.

6.6. Allocation Element Requirements

Table 5 contains the functional requirements for fuel level display system of scania that are to be verified on the fuel level display system model in both Simulink Design Verifier and UPPAAL during model checking activity.

Disclaimer:
All the numerical values mentioned in the requirements of fuel level display system are arbitrary values and do not represent the actual values in Scania fuel level display system.

ID	Description	Reference
	AE201	
AER-01	"TotalFuelLevel should be the output of a filter that includes information from both fuelLevel and fuelRate to achieve a stable signal. The filter should be implemented with a Kalman algorithm given by the following equations and with the feedback gain K=1.0786*10^{-5}." [14]. $x_{start} = \begin{cases} y_s(t), & \lvert y_s - \hat{x}_{old} \rvert > 0.1X_{tot} \text{ or } y_s > 0.9X_{tot} \\ \hat{x}_{old}, & otherwise \end{cases}$ $\hat{x}(t+T_s) = \begin{cases} x_{start}, & during\ start-up \\ \hat{x}(t) - T_s u(t) + K(y_s(t) - \hat{x}(t)), & other \end{cases}$ $y(t) = F(\hat{x}(t))$ Where: $y_s = measured\ \ fuel\ level\ [m^3]$ $\hat{x}_{old} = fuel\ level\ at\ last\ shutdown\ [m^3]$ $X_{tot} = total\ \ fuel\ volume\ [m^3]$ $T_s = sampletime\ [s]$ $u = fuel\ consumption\ [m^3/s]$ $\hat{x} = estimated\ \ fuel\ level\ [m^3]$ $K = feedback\ gain\ [-]$ $F(x) = function\ converting\ m^3\ to\ corresponding\ \%$ $y = total\ \ fuel\ level\ [\%]$ $x_{start} = start\ state\ [m^3]$ "K(t) is calculated using the theory of Kalman filters based on a estimated variance for the inputs u(t) and y_s(t). Variance(u(t))= 9.9334e-013 and Variance(y_s(t))= 0.0085 gives K= 1.0786e-005." [14].	AER_201_11
AER-02	"The start-up state for the totalFuelLevel estimated should be the state saved from last shutdown if the stored value and fuelLevel doesn't differ with more than 15% of the total volume or if fuelLevel is above 95% of the useable tank capacity." [14].	AER_201_12
AER-03	"If a refill of the tank is done while the ECU is on it should be detected by the algorithm if the sensor(s) indicates a 35% increase compared to the estimated volume. The increase should be held at least 10 seconds so that sloshing is ignored." [14].	AER_201_13

AER-04	"The refill detection should be possible only when the parking brake is applied. The parking brake should be steadily applied for at least 10 seconds before the vehicle is considered to be parked." [14].	AER_201_14
AER-05	"If a refill is detected the filter algorithm should not be used, the estimate should instead the value indicated by the fuel level sensor(s) until the refill is done (parking brake released). When the refill is ended the algorithm continues to calculate using the current value from fuel level sensor(s) signal as initial value." [14].	AER_201_15
AE202		
AER-06	"The lowFuelLevelWarning should be set to 1 (true) when input totalFuelLevel is below a pre-defined level. The level should be 15% for tank sizes equal or below 905liters and 12% for tanks sizes larger than 905liters." [15].	AER_202_2
AER-07	"The lowFuelLevelWarning should be kept true, once it is activated, until the algorithm is restarted by an ECU shutdown or if the totalFuelLevel reaches above 25%." [15].	AER_202_3
AER-08	"Output signal lowFuelLevelWarning should have initial value 0 (false)." [15].	AER_202_5

Table 5: Requirements for fuel level display system.

7. Simulink Design Verifier - Verification Model and Results

Figure 8 represent the verification model built in simulink that is used for verification and property proving of fuel level display system in Simulink Design Verifier. It contains the block for input assumptions, a block that represent the simulink model of fuel level display system, blocks for all the requirements of fuel level display system that it is supposed to satisfy and a verification button for each requirement.

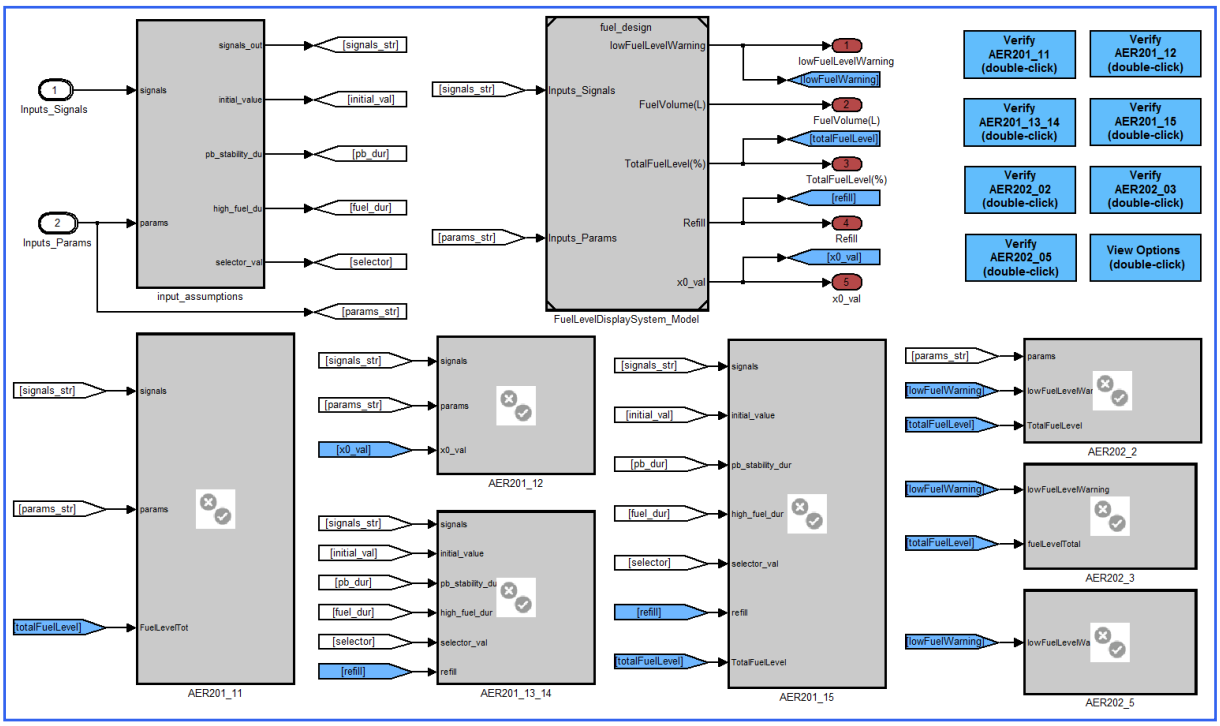

Figure 8: Simulink Design Verifier - verification model.

Figure 9 represents the model of fuel level display system in Simulink that is to be verified against the requirement and to see whether the requirements are satisfied by the system model or not. In the next section Simulink model of fuel level display system is presented briefly. Complete model of the fuel level display system cannot be presented in this thesis because of the privacy policies of Scania.

Scania already has the model of fuel level display system available in simulink but that model covers all the four variants of fuel level display system plus it also covers all the sensor type and all sizes of fuel tank, it also contains some extra features which are not part of the actual system. In order to make the model according to requirements the model is simplified and all the extra details are removed from it. Now the following simulink model is for one system variant, one type of fuel level sensor, one type of fuel tank and where total fuel level is calculated by COO.

7.1. Model of Fuel Level Display system in Simulink

Constants
All the values that are written in capital letters in the fuel level display models in Simulink are constants and the following table 6 presents the values of all the constants that are used in fuel level display system model in Simulink and their description.

Constant	Value	Description	Used in Module
PCT_TO_DEC_F32	1/100	Conversion from percentage to decimal.	Figure 16
VOLUME_DIFF_ACCEPTED _F32	0.1	10% volume diff between estimated and sensor value is accepted to use the stored estimate from shutdown as startup value.	Figure 18
NEAR_TOP_PERCENTAGE_ F32	90	Using a limit to always start with the raw sensor value if fuel level is high. This will override the condition that there have to be 10% difference in volume at startup for using the raw value. It is useful when the tank is top-filled with small amounts.	Figure 18
REFUEL_DETECTION_LIMIT _F32	30	Refuel detection.	Figure 21
L_PER_H_TO_M3_PER_S_ F32	1/(1000*3600)	Liter per hour to m3 per second.	Figure 23
LITER_TO_M3_F32	1/1000	Liter to m3.	Figure 24, Figure 26
KALMAN_FEEDBACK_GAIN _F32	1e-009 * 10780	Kalman gain calculated as specified in AER201.	Figure 25
TS_F32	0.01	COO7 constants.	Figure 25
EPSILON_F32	1.1755 e-038	Value used to avoid divide by zero situation.	Figure 27
LOW_IND_RELEASE_LEVEL _F32	20	Used if a refuel is not detected by the algorithm the low level warning should be turned off.	Figure 30
LOW_FUEL_LEVEL_LARGE _TANKS_F32	7	Percent warning level (large tanks above 900 liter capacity)	Figure 31
LARGE_TANK_LIMIT_F32	900	Above 900bliters theoretical volume is considered as large tanks.	Figure 31
LOW_FUEL_LEVEL_NORM AL_TANKS_F32	10	Percent warning level.	Figure 31
LOW_FUEL_LEVEL_IND_A CTIVE_U08	uint8(10)	Parameter.	Figure 33

Table 6: Constants used in Simulink model of fuel level display system.

Due to the confidentiality concerns of Scania this report only contains a small portion of fuel level display system that is modeled in simulink, complete Simulink model of fuel level display system cannot be presented in this report due to the privacy policies of Scania.

Figure 9 presents the block that represents the fuel level display system in the verification model with two main inputs and outputs as shown in the figure 9 below.

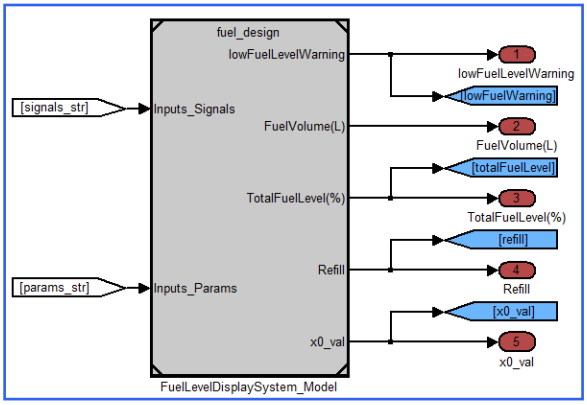

Figure 9: Fuel level display system.

When the model present in figure 9 is expanded by double clicking on it then the model that is present in figure 10 below is opened.

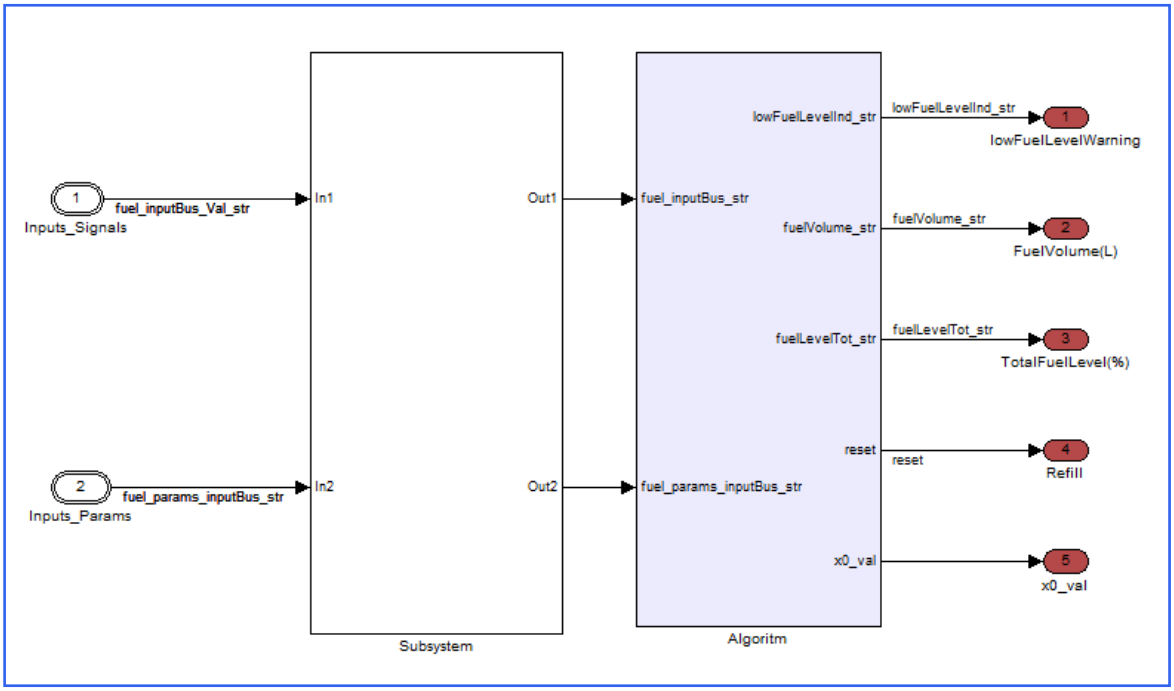

Figure 10: Fuel level display system algorithm.

Model present in figure 10 contains a sub block named algorithm which contains within it the model that is present in the figure 11 below which contains 3 sub blocks representing evaluate parking brake, fuel level estimation algorithm and low fuel level warning.

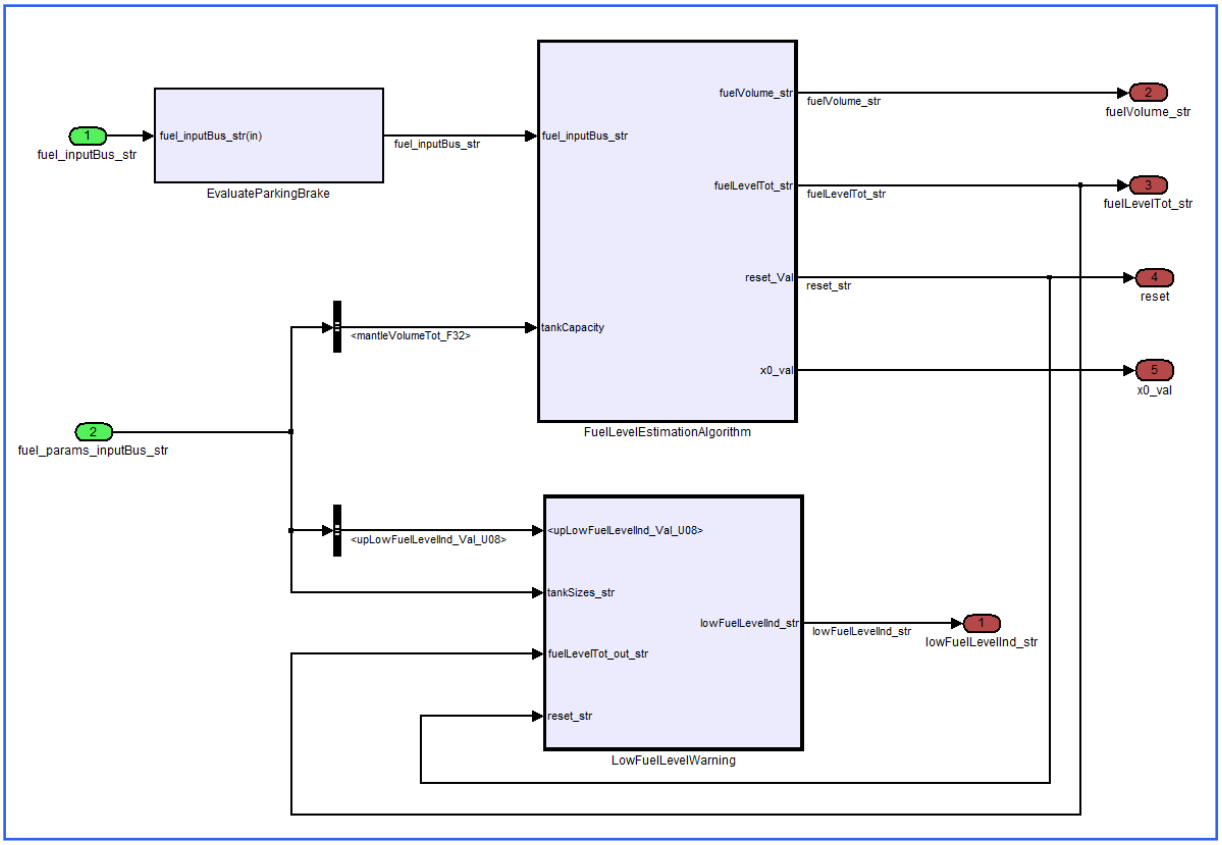

Figure 11: Fuel level display system algorithm.

Expanding evaluate parking brake sub block presents the following model showed in figure 12, which has a state flow for evaluate parking brake applied.

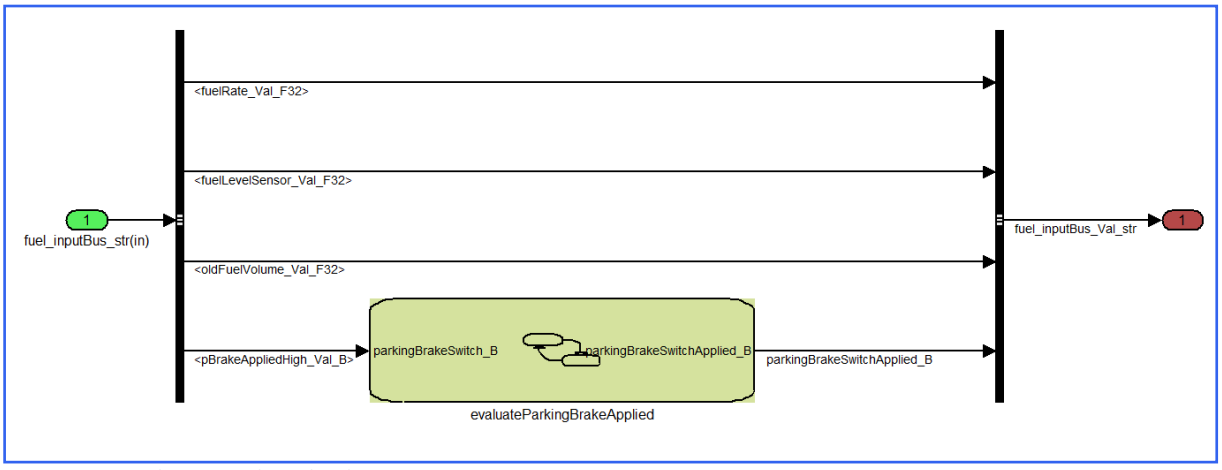

Figure 12: Evaluate parking brake.

Evaluate parking brake applied state flow contains within it the state flow present in figure 12, which evaluates if parking brake has been applied consecutively during 0.5 seconds.

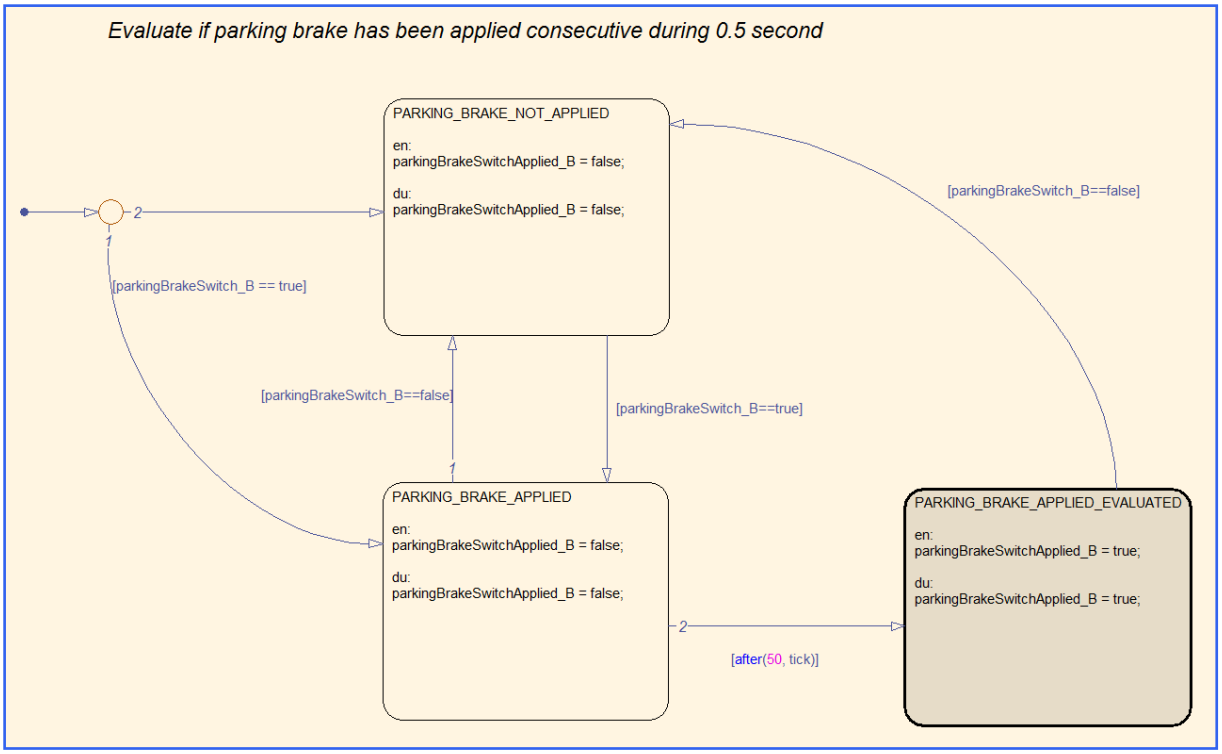

Figure 13: Evaluate parking brake applied.

Sub block fuel level estimation algorithm in figure 11 contains three sub blocks named calculate current volume levels, algorithm reset calculation and Kalman observer estimation present in figure 14 which can be seen by expanding the fuel level estimation algorithm sub block present in figure 11.

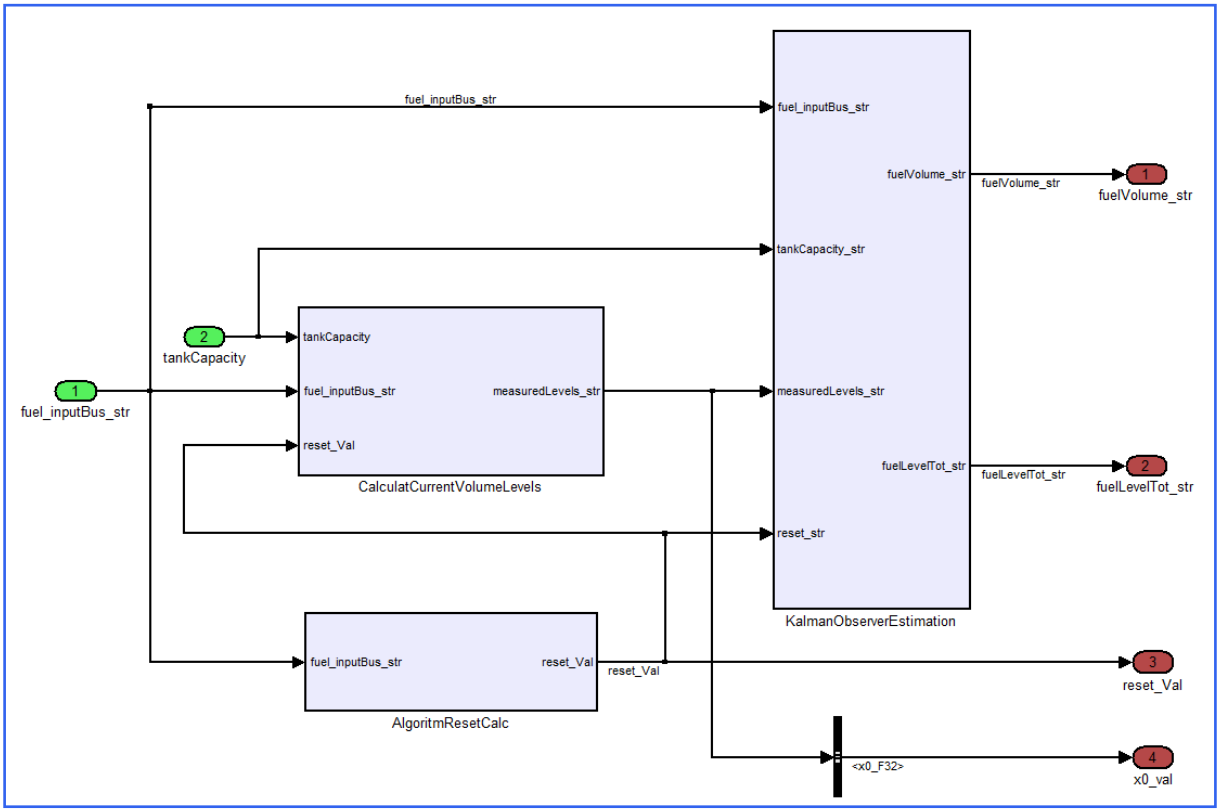

Figure 14: Fuel level estimation algorithm.

Algorithm reset calculation sub block in figure 14 contains the following model present in figure 15 which contains a state flow called refuel detection.

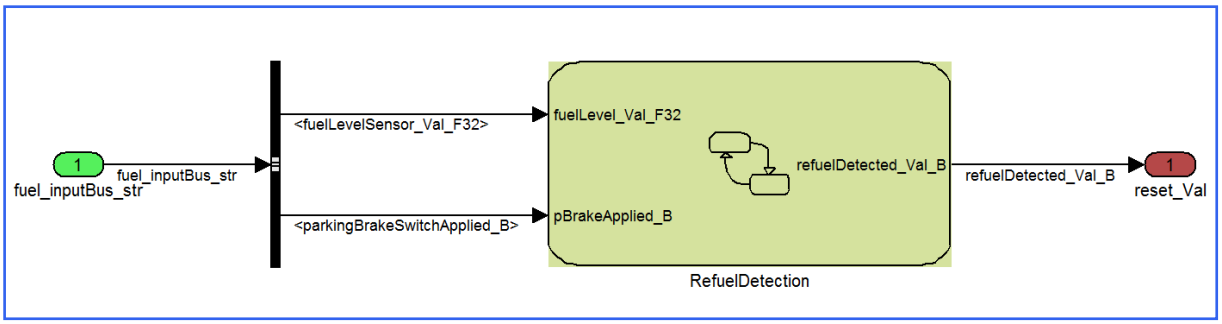

Figure 15: Algorithm reset calculation.

State flow for refuel detection in figure 15 contains the following state flow present in figure 16 in it that is used to check if refuel is performed or not.

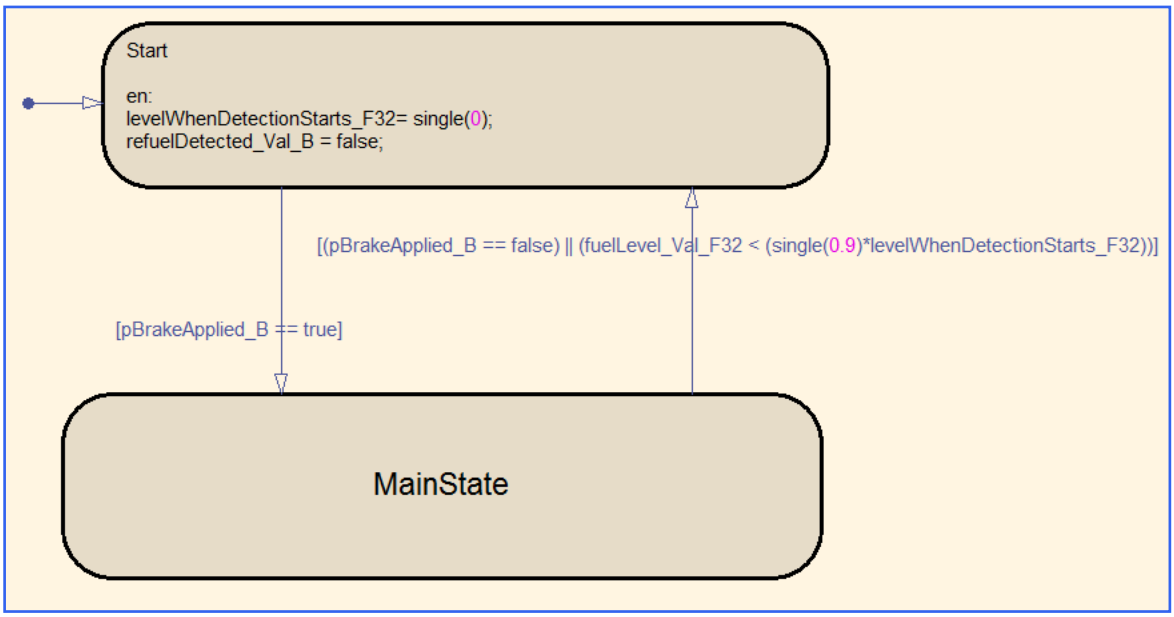

Figure 16: Refuel detection.

The main state in model present in above figure 16 contains the following state flow present in figure 17 that is the actually process for the detection of refuel.

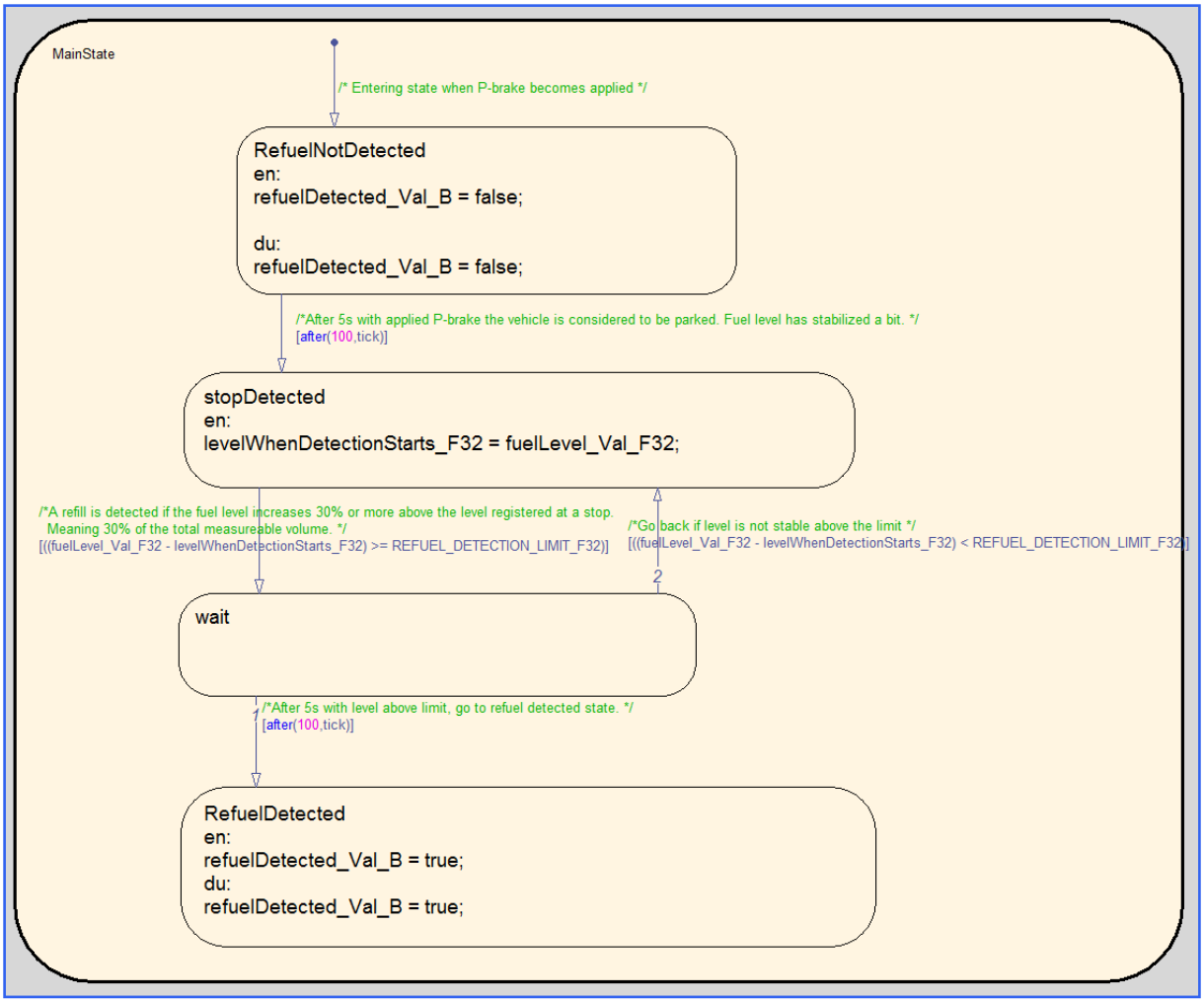

Figure 17: Refuel detection.

7.2. Input Assumptions

Figure 18 represents the input assumptions block in verification model. Following section represent how the assumptions on input params and signals are defined. Later-on the buttons are used to set the input assumptions for each requirement to be verified by the Simulink Design Verifier.

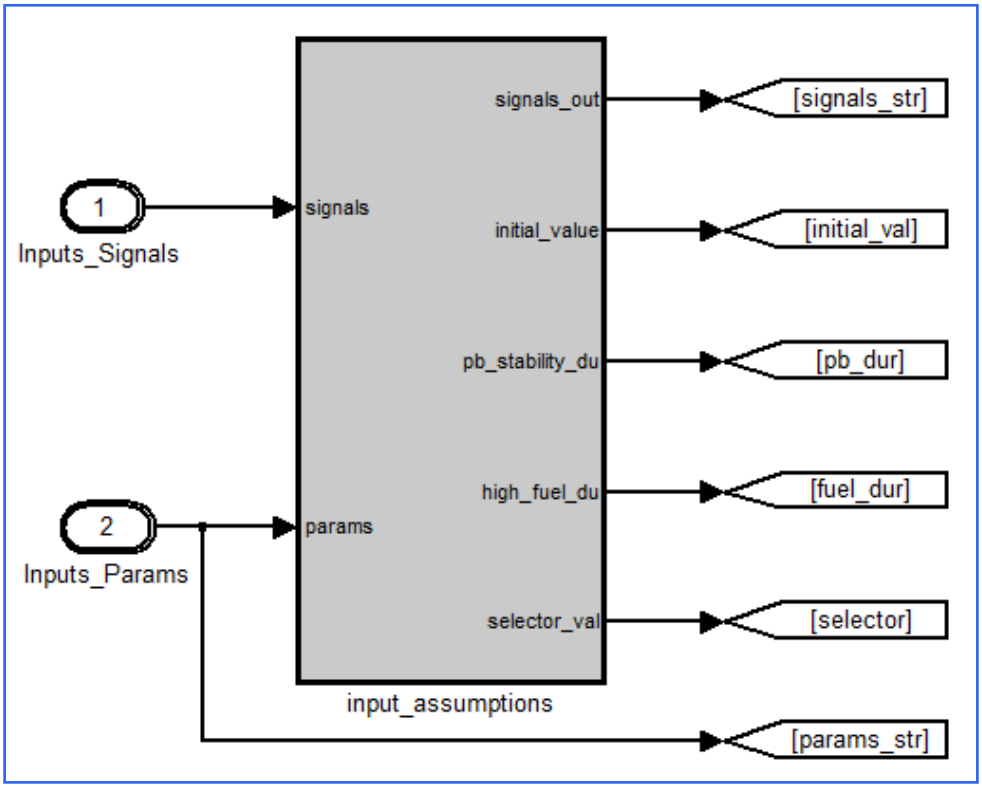

Figure 18: Input assumptions.

Input assumptions block in figure 18 contains the model present in figure 19 which contains 3 sub blocks params assumptions, signal assumptions and verification selectors.

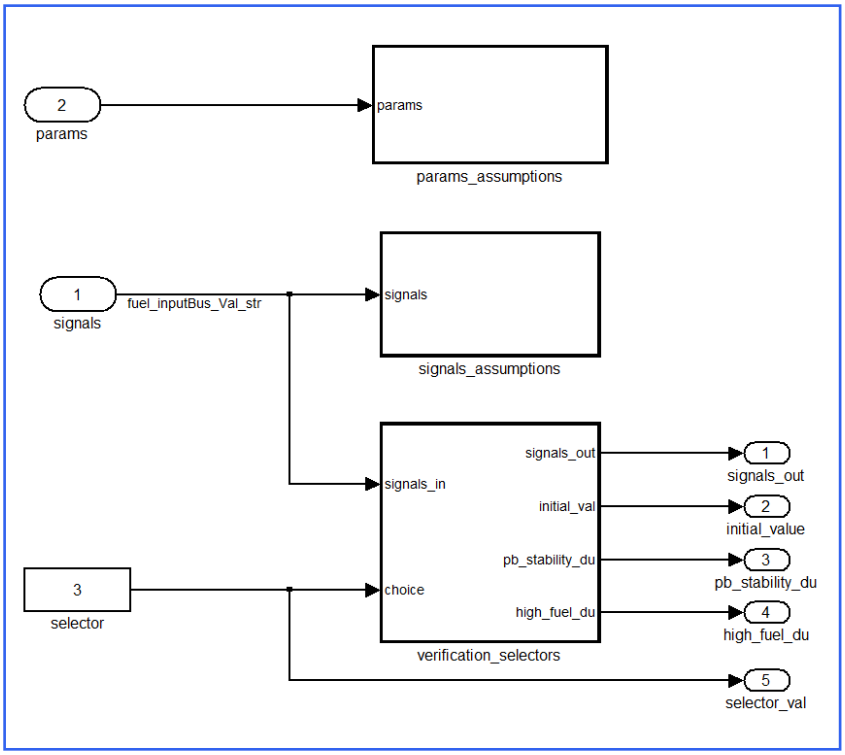

Figure 19: Input assumptions.

Params assumptions sub block present in figure 19 contains the model present in figure 20. This actually defines the limits on all the input params used in fuel level display system. These are activated and deactivated by using the instructions present behind verification buttons because each requirement require different params not all of them.

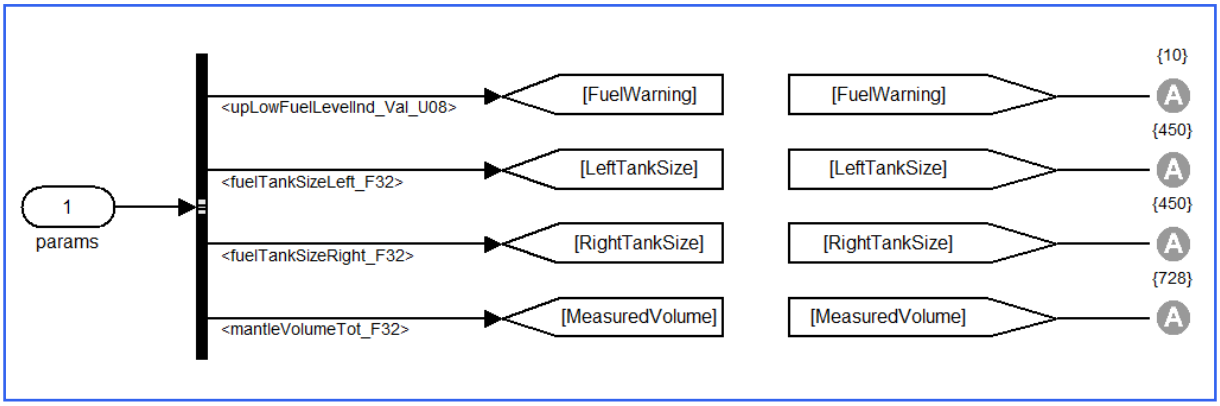

Figure 20: Params assumptions.

Signals assumptions sub block present in figure 19 contains the model present in figure 21. This actually defines the limits on all the input signals used in fuel level display system. These are activated and deactivated by using the instructions present behind verification buttons because each requirement require different signals not all of them.

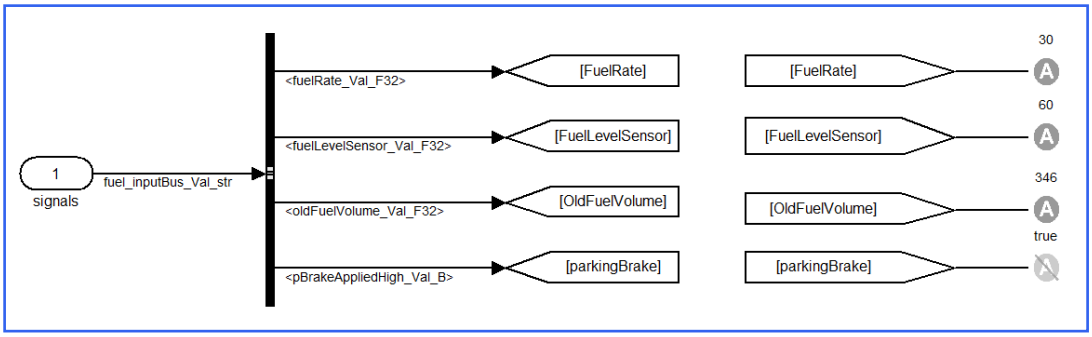

Figure 21: Signal assumptions.

Figure 22 to Figure 26 show how the scenarios are generated for refill and requirement AER-07 (AER202_03). Verification selectors sub block present in figure 19 contains the model present in figure 22 which contains two sub blocks refill scenario and AER202_03 scenario.

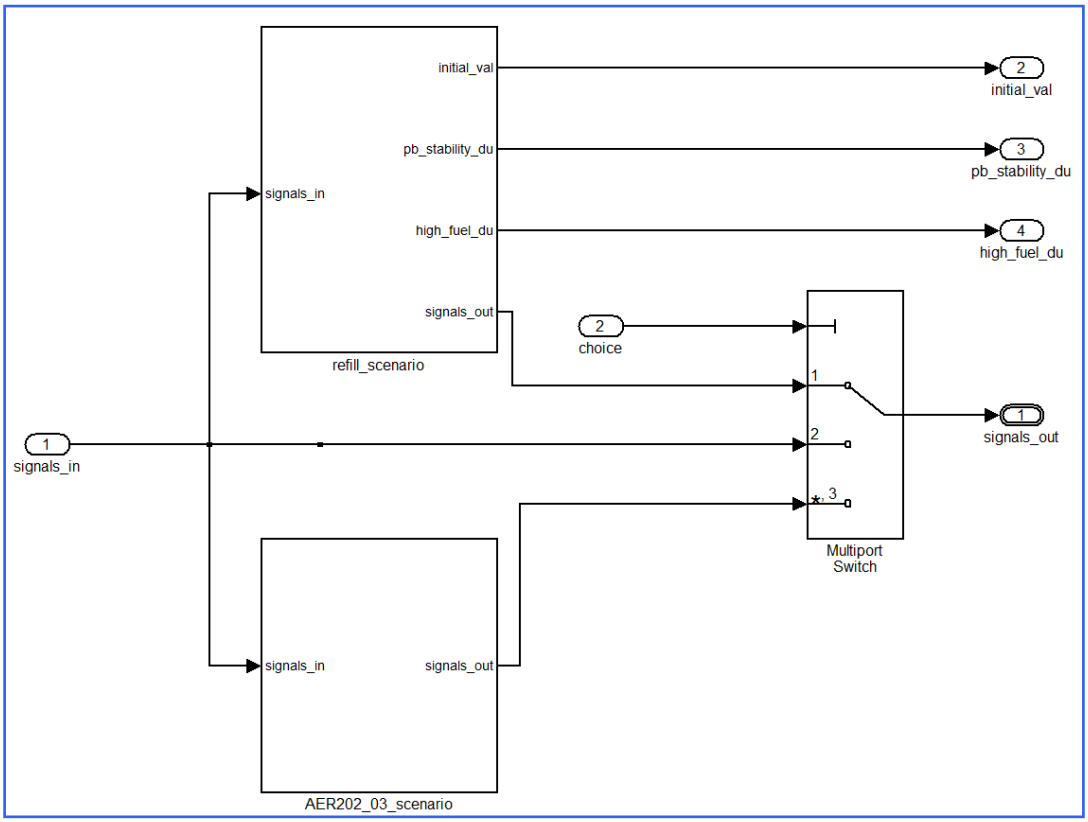

Figure 22: Verification selectors.

Refill scenario sub block present in figure 22 contains the model present in figure 23 which contains a state flow refill scenario generator.

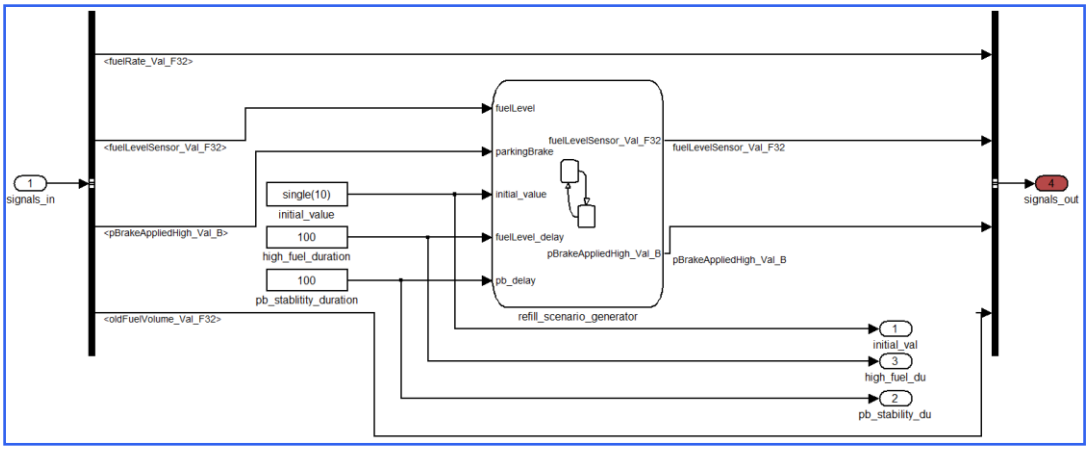

Figure 23: Refill scenario.

The refill scenario generator state flow present in figure 23 contains the state flow present in figure 24 which actually describes that how the scenario is generated for the refill.

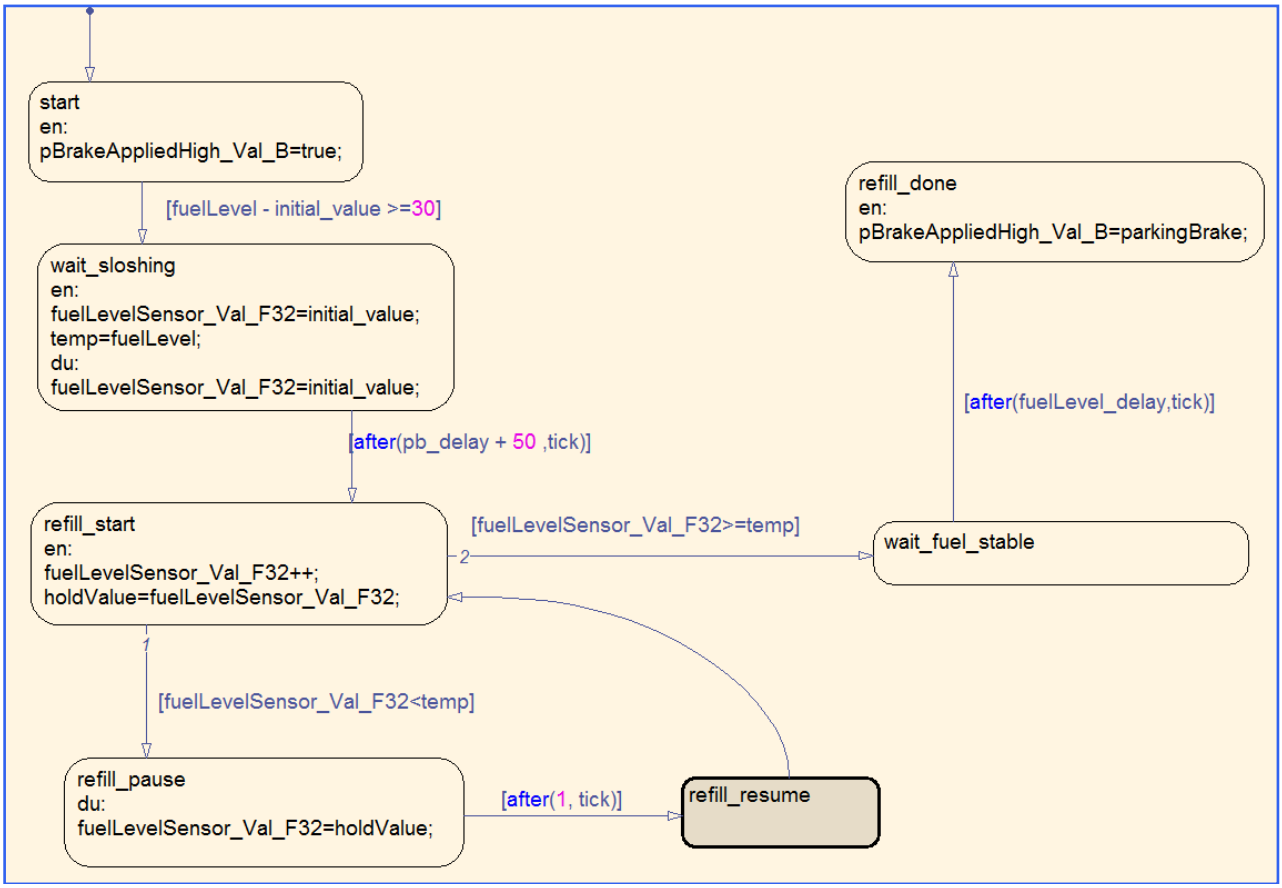

Figure 24: Refill scenario generator.

AER202_03 scenario sub block present in figure 22 contains the model present in figure 25 which contains a state flow AER202_03 scenario generator.

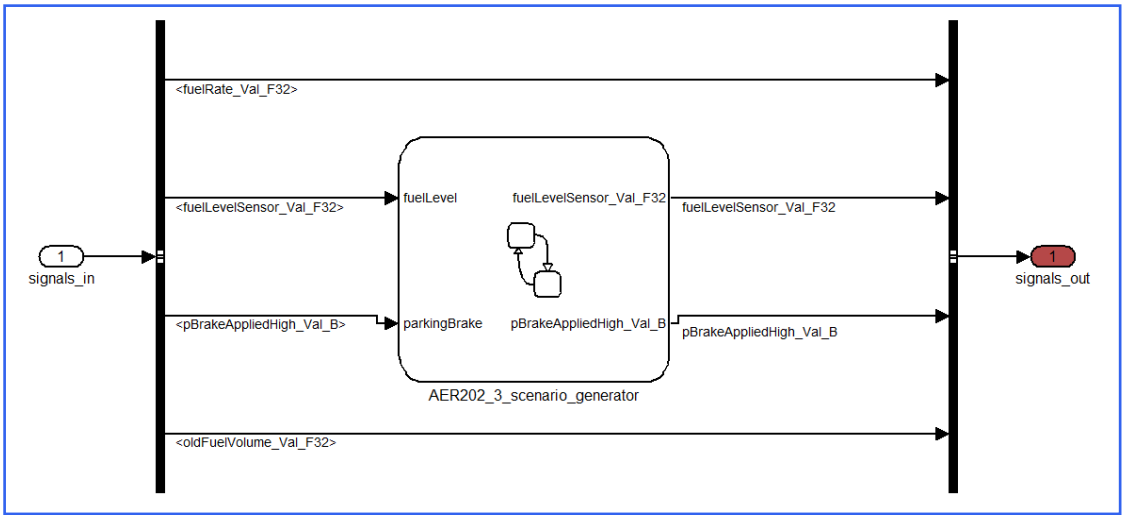

Figure 25: AER202_3 scenario.

AER202_03 scenario generator state flow present in figure 25 contains the state flow present in figure 26 which actually describes how the scenario is generated for the requirement AER-07 (AER202_03) that is "To verify that he lowFuelLevelWarning should be kept true, once it is activated, until the algorithm is restarted by an ECU shutdown or if the totalFuelLevel reaches above 20%" [15].

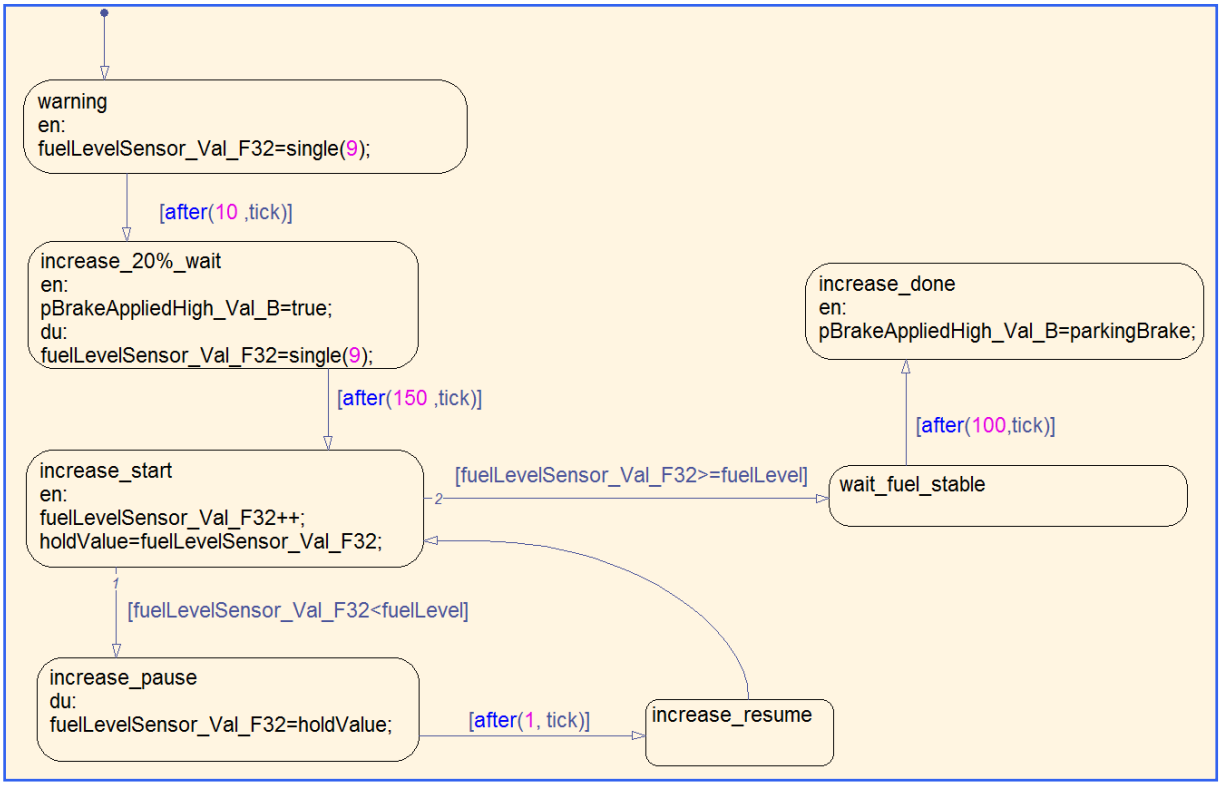

Figure 26: AER202_3 scenario generator.

7.3. Activation buttons

Figure 27 presents the buttons that are used to run the verification for each requirement according to their names. Every button has code written behind it that is used to set the input assumption for each requirement in the input assumptions block.

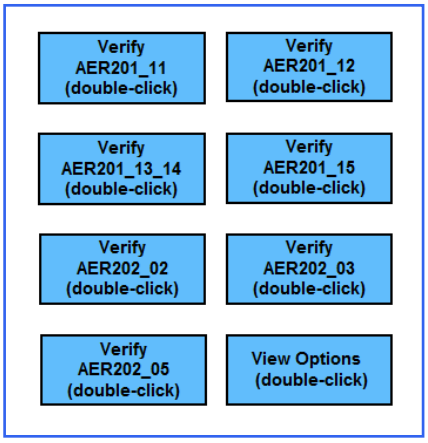

Figure 27: Verification buttons.

Common code behind for buttons (1-7)

```
mdlname = bdroot(gcb);
prop_path1 = [mdlname '/AER201_11/P_201_11']
prop_path2 = [mdlname '/AER201_12/P_201_12'];
prop_path3 = [mdlname '/AER201_13_14/P_201_13_14'];
prop_path4 = [mdlname '/AER201_15/P_201_15'];
prop_path5 = [mdlname '/AER202_2/P_202_2'];
prop_path6 = [mdlname '/AER202_3/P_202_3'];
prop_path7 = [mdlname '/AER202_5/P_202_5'];

assump_path1 = [mdlname '/input_assumptions/selector'];
assump_path2 = [mdlname '/input_assumptions/signals_assumptions/fuelLevel'];
assump_path3 = [mdlname '/input_assumptions/signals_assumptions/pbrake'];
assump_path4 = [mdlname '/input_assumptions/signals_assumptions/oldFuelVolume'];

if strcmp(get_param(gcs, 'Dirty'), 'on')
save_system;
end
sldvdemo_helper('run',gcbh);
```

Table 7: Common code behind for buttons (1-7).

Code behind button AER_201_11

```
Simulink.BlockDiagram.loadActiveConfigSet('fuel_verification_0','config1.m');

set_param(prop_path1,'enabled','on')
set_param(prop_path2,'enabled','off')
set_param(prop_path3,'enabled','off')
set_param(prop_path4,'enabled','off')
set_param(prop_path5,'enabled','off')
set_param(prop_path6,'enabled','off')
set_param(prop_path7,'enabled','off')

set_param(assump_path1,'Value','2')
set_param (assump_path2,'intervals', '[1 60]')
```

Table 8: Code behind button AER_201_11.

Code behind button AER_201_12

```
Simulink.BlockDiagram.loadActiveConfigSet('fuel_verification_0','config2.m');

set_param(prop_path1,'enabled','off')
set_param(prop_path2,'enabled','on')
set_param(prop_path3,'enabled','off')
set_param(prop_path4,'enabled','off')
set_param(prop_path5,'enabled','off')
set_param(prop_path6,'enabled','off')
set_param(prop_path7,'enabled','off')

set_param(assump_path1,'Value','2')
set_param (assump_path2,'intervals', '[1 100]')
```

Table 9: Code behind button AER_201_12.

Code behind button AER_201_13_14

```
Simulink.BlockDiagram.loadActiveConfigSet('fuel_verification_0','config2.m');

set_param(prop_path1,'enabled','off')
set_param(prop_path2,'enabled','off')
set_param(prop_path3,'enabled','on')
set_param(prop_path4,'enabled','off')
set_param(prop_path5,'enabled','off')
set_param(prop_path6,'enabled','off')
set_param(prop_path7,'enabled','off')

set_param(assump_path1,'Value','1')
set_param(assump_path3,'intervals','true')
set_param (assump_path2,'intervals', '60')
```

Table 10: Code behind button AER_201_13_14.

Code behind button AER_201_15

```
Simulink.BlockDiagram.loadActiveConfigSet('fuel_verification_0','config2.m');

set_param(prop_path1,'enabled','off')
set_param(prop_path2,'enabled','off')
set_param(prop_path3,'enabled','off')
set_param(prop_path4,'enabled','on')
set_param(prop_path5,'enabled','off')
set_param(prop_path6,'enabled','off')
set_param(prop_path7,'enabled','off')

set_param(assump_path1,'Value','1')
set_param(assump_path3,'intervals','true')
set_param (assump_path2,'intervals', '60')
```

Table 11: Code behind button AER_201_15.

Code behind button AER_202_02

```
Simulink.BlockDiagram.loadActiveConfigSet('fuel_verification_0','config2.m');

set_param(prop_path1,'enabled','off')
set_param(prop_path2,'enabled','off')
set_param(prop_path3,'enabled','off')
set_param(prop_path4,'enabled','off')
set_param(prop_path5,'enabled','on')
set_param(prop_path6,'enabled','off')
set_param(prop_path7,'enabled','off')

set_param(assump_path1,'Value','2')
set_param (assump_path2,'intervals', '[0 9]')
set_param(assump_path3,'enabled','off')
set_param (assump_path4,'intervals', '72')
```

Table 12: Code behind button AER_202_02.

Code behind button AER_202_03

```
Simulink.BlockDiagram.loadActiveConfigSet('fuel_verification_0','config1.m');

set_param(prop_path1,'enabled','off')
set_param(prop_path2,'enabled','off')
set_param(prop_path3,'enabled','off')
set_param(prop_path4,'enabled','off')
set_param(prop_path5,'enabled','off')
set_param(prop_path6,'enabled','on')
set_param(prop_path7,'enabled','off')

set_param(assump_path1,'Value','3')
set_param (assump_path2,'intervals', '60')
set_param(assump_path3,'enabled','off')
set_param (assump_path4,'intervals', '346')
```

Table 13: Code behind button AER_202_03.

Code behind button AER_202_05

```
Simulink.BlockDiagram.loadActiveConfigSet('fuel_verification_0','config1.m');

set_param(prop_path1,'enabled','off')
set_param(prop_path2,'enabled','off')
set_param(prop_path3,'enabled','off')
set_param(prop_path4,'enabled','off')
set_param(prop_path5,'enabled','off')
set_param(prop_path6,'enabled','off')
set_param(prop_path7,'enabled','on')

set_param(assump_path1,'Value','2')
set_param (assump_path2,'intervals', '11')
set_param(assump_path3,'enabled','off')
set_param (assump_path4,'intervals', '465')
```

Table 14: Code behind button AER_202_05.

Code behind button View Options

The last button named View Options has sldvdemo_helper('showopts',gcbh); in code behind and can be used to open the dialog box of configuration parameters.

Table 15: Code behind button View Options.

7.4. Property Verification

This section represents all the requirements as they are formulated in Simulink Design Verifier supported format and how the blocks for property verification are defined in simulink to be verified by Simulink Design Verifier.

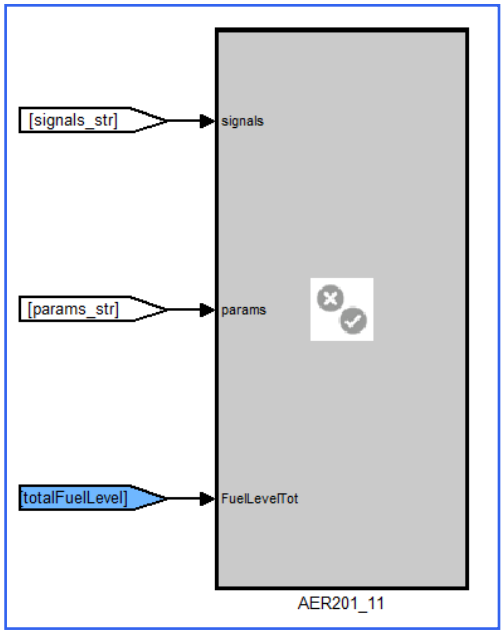

Figure 28: AER-01 (AER201_11).

Figure 29 and 30 present how the requirement AER-01 (AER_201_11) is implemented in simulink. "The requirement is to verify that totalFuelLevel should be the output of a filter that includes information from both fuelLevel and fuelRate to achieve a stable signal. The filter should be implemented with a Kalman algorithm." [14]. Details of the requirement are mentioned in Table 5.

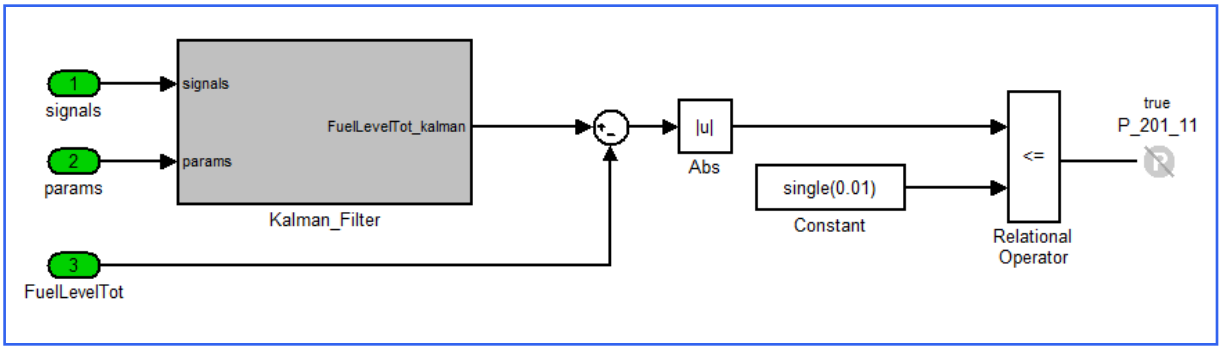

Figure 29: Property verification - AER-01 (AER201_11).

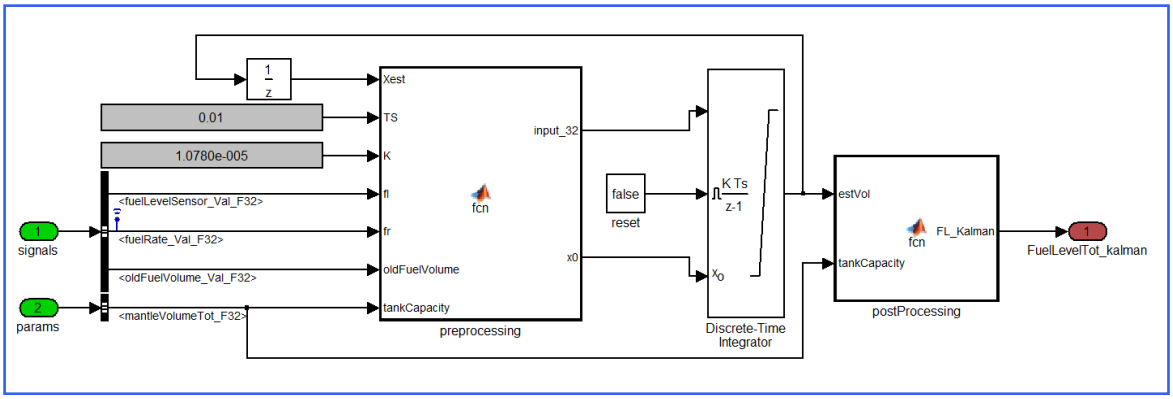

Figure 30: Property verification - Kalman filter.

Following 2 tables contain the code written behind MATLAB function pre-processing and post-processing for verification of requirement.

Code behind preProcessing function block

```
function [input_32,x0] = fcn(Xest,TS,K,fl,fr,oldFuelVolume,tankCapacity)

%covert current raw fuel level (fl) from percentage to litres
rawLevel = tankCapacity * fl * 0.01;

%compute the startup state (x0) for the Kalman filter
if ( (abs(rawLevel - oldFuelVolume) > ((10/100)*tankCapacity)) || (fl > 90) )
   x0_ = rawLevel;
else
   x0_=oldFuelVolume;
end

%scale the fuel rate (fr) from L/hour to M3/sec
fr = fr * 2.7778e-007;

%convert levels from litres to M3
x0 = double(x0_ * 0.001);
rawLevel = rawLevel * 0.001;

% input for Next State
input_32 = double(((K/TS)*(rawLevel - Xest)) - (fr));

%%%%%%%%%%%%%%%%%%%%_end_%%%%%%%%%%%%%%%%%%
```

Table 16: Code behind preProcessing function block.

Code behind postProcessing function block

```
function FL_Kalman  = fcn(estVol,tankCapacity)

% fule estimate in litres (convert from M3)
flEstPct_ = estVol * (1/ 0.001);

% convert to percentage and use max function to avoid division by 0
flEstPct_ = (flEstPct_ * single(100))/ max(tankCapacity,1.1755e-038);

%saturate between 0 and 100% and output %age value in FL_Kalman
if (flEstPct_ > 99.5 )
   flEstPct_ = single(100);
end

FL_Kalman = max (flEstPct_, single(0));
%%%%%%%%%%%%%%%%%%%%%_end_%%%%%%%%%%%%%%%%%%
```

Table 17: Code behind postProcessing function block.

Figure 31 and Figure 32 show how the requirement AER-02 (AER_201_12) is implemented in simulink. "The requirement is to verify that the start-up state for the totalFuelLevel estimated should be the state saved from last shutdown if the stored value and fuelLevel doesn't differ with more than 10% of the total volume or if fuelLevel is above 90% of the useable tank capacity." [14].

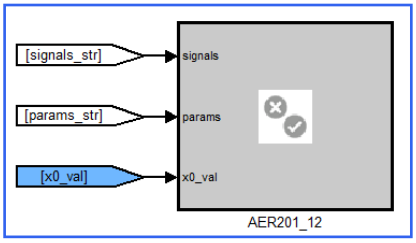

Figure 31: AER-02 (AER201_12).

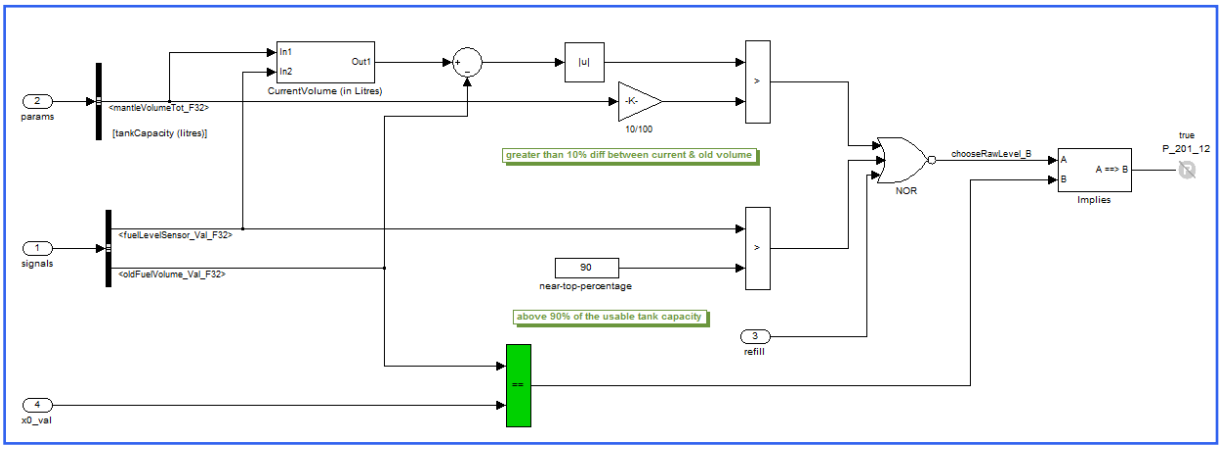

Figure 32: Property verification - AER-02 (AER201_12).

Figure 33 and Figure 34 show how the requirement AER-03 (AER_201_13) and AER-04 (AER_201_14) are implemented in simulink to be verified by Simulink Design Verifier.

"Requirement AER-03 (AER_201_13) is to verify that if a refill of the tank is done while the ECU is on it should be detected by the algorithm if the sensor(s) indicates a 30% increase compared to the estimated volume. The increase should be held at least 5 seconds so that sloshing is ignored." [14].

"Requirement AER-04 (AER_201_14) is to verify that the refill detection should be possible only when the parking brake is applied. The parking brake should be steadily applied for at least 5 seconds before the vehicle is considered to be parked." [14].

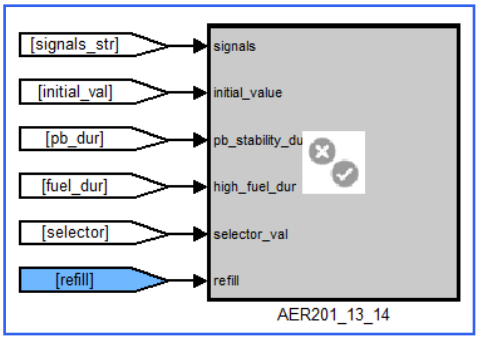

Figure 33: AER-03 (AER201_13) and AER-04 (AER201_14).

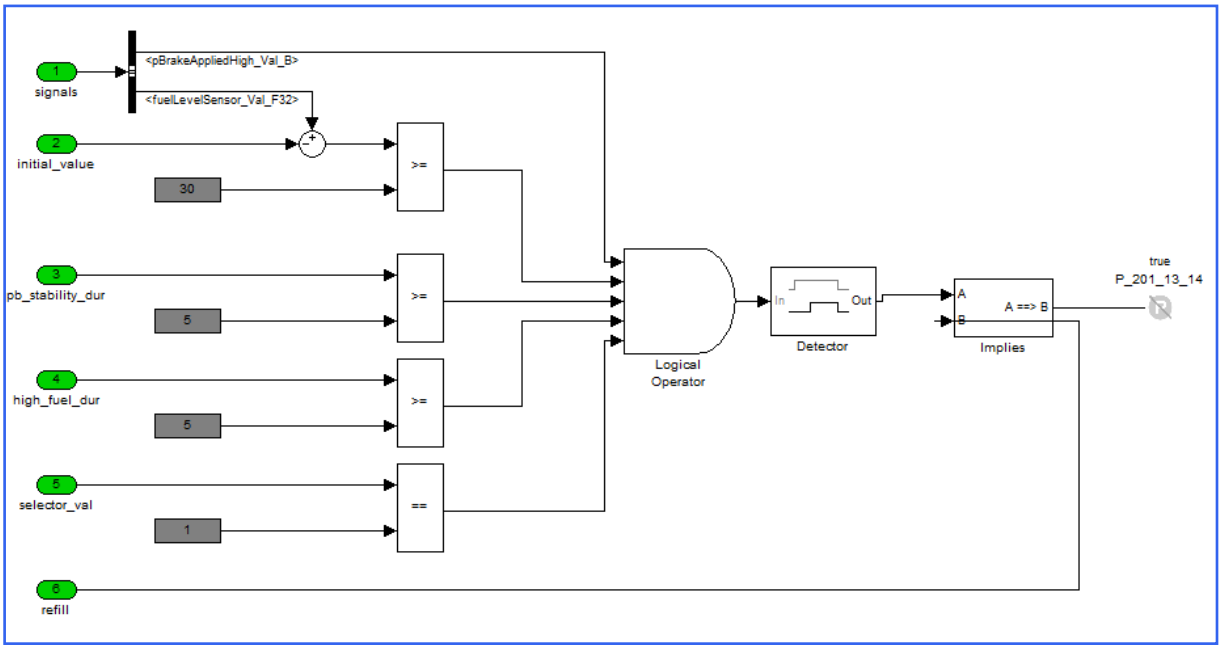

Figure 34: Property Verification - AER-03 (AER201_13) and AER-04 (AER201_14).

Figure 35 and Figure 36 show how the requirement AER-05 (AER_201_15) is implemented in simulink to be verified by Simulink Design Verifier. "The requirement is to verify that if a refill is detected the filter algorithm should not be used, the estimate should instead the value indicated by the fuel level sensor(s) until the refill is done (parking brake released). When the refill is ended the algorithm continues to calculate using the current value from fuel level sensor(s) signal as initial value." [14].

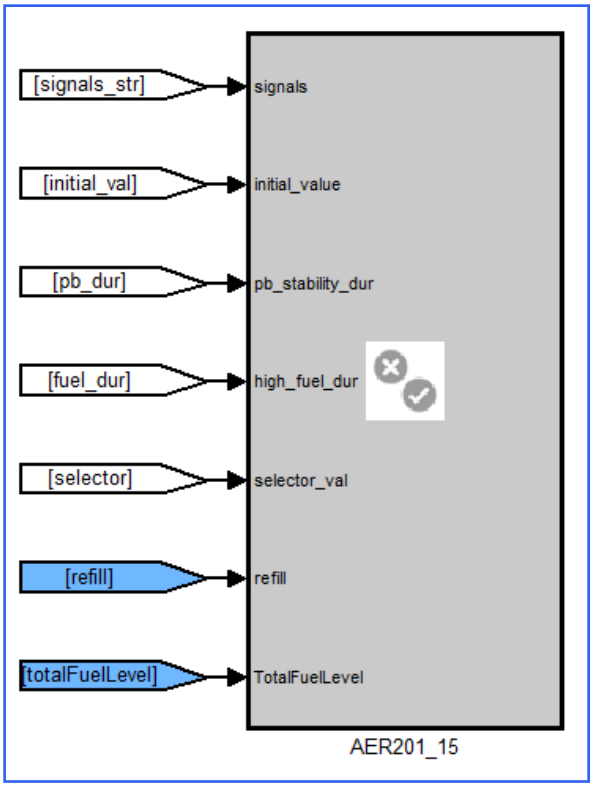

Figure 35: AER-05 (AER201_15).

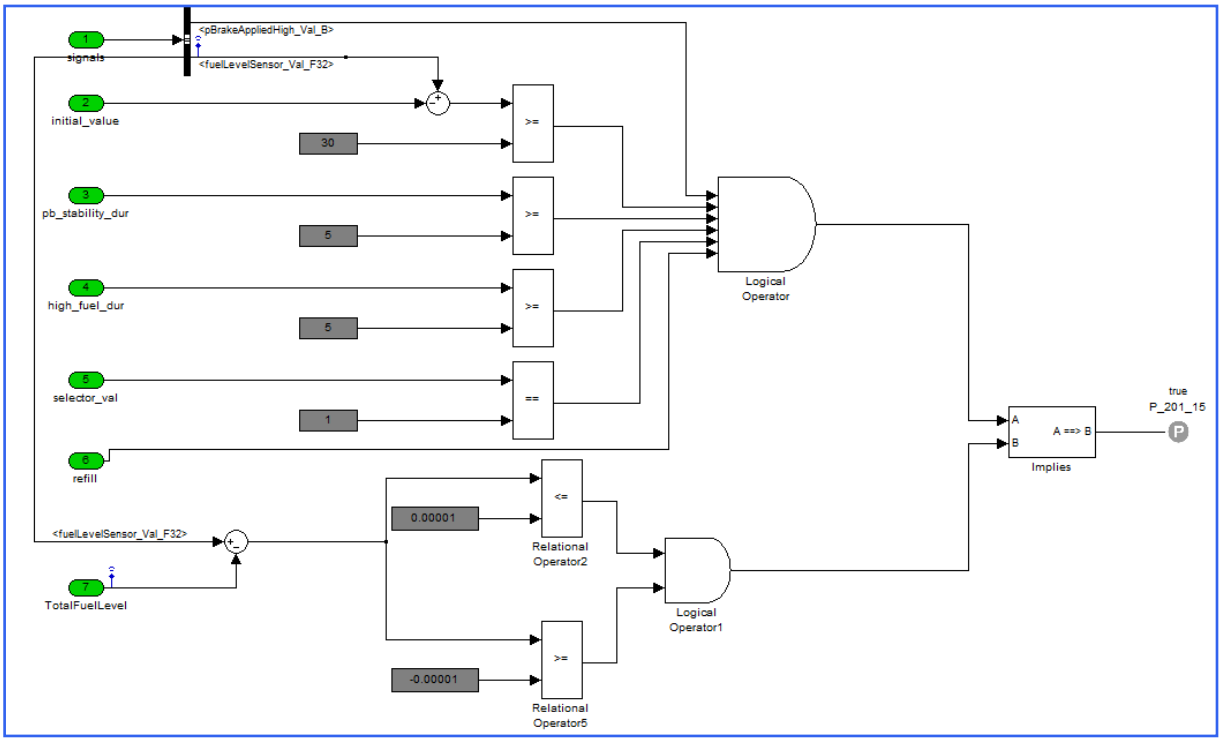

Figure 36: Property verification - AER-05 (AER201_15).

Figure 37 and Figure 38 represent how the requirement AER-06 (AER_202_2) is implemented in simulink. "The requirement is to verify that the lowFuelLevelWarning should be set to 1 (true) when input totalFuelLevel is below a pre-defined level. The level should be 10% for tank sizes equal or below 900liters and 7% for tanks sizes larger than 900liters." [15].

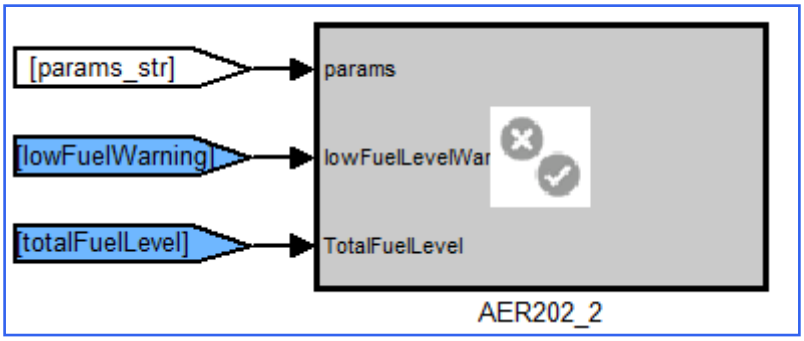

Figure 37: AER-06 (AER202_2).

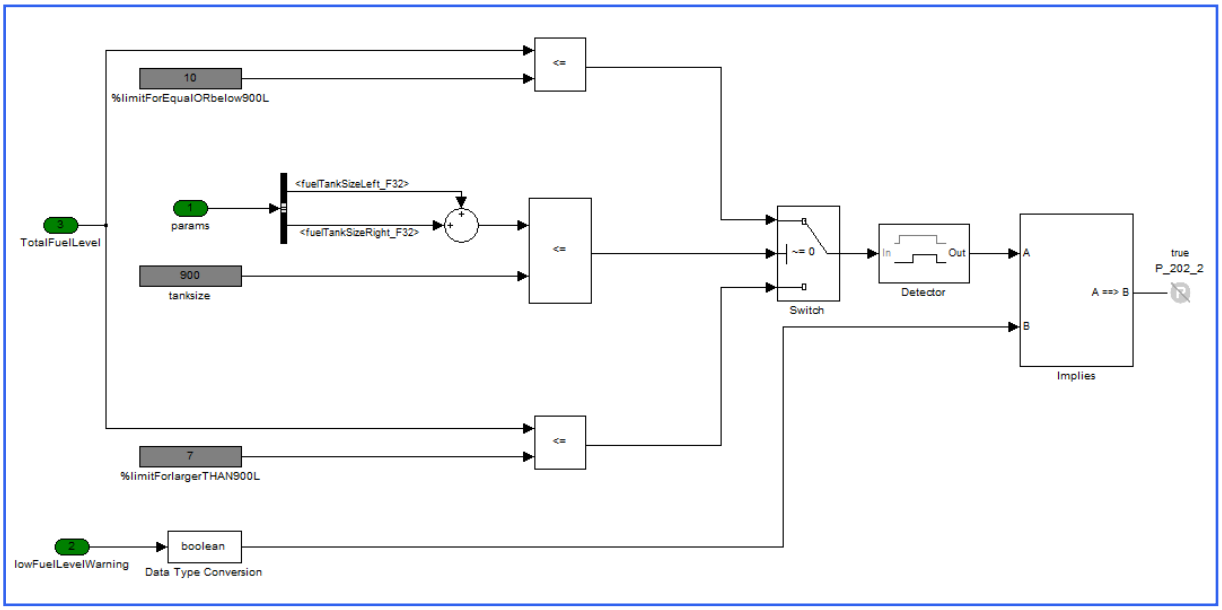

Figure 38: Property verification - AER-06 (AER202_2).

Figure 39 and Figure 40 represent how the requirement AER-07 (AER_202_3) is transformed in simulink model to be verified by Simulink Design Verifier. "The requirement is to verify that the lowFuelLevelWarning should be kept true, once it is activated, until the algorithm is restarted by an ECU shutdown or if the totalFuelLevel reaches above 20%." [15].

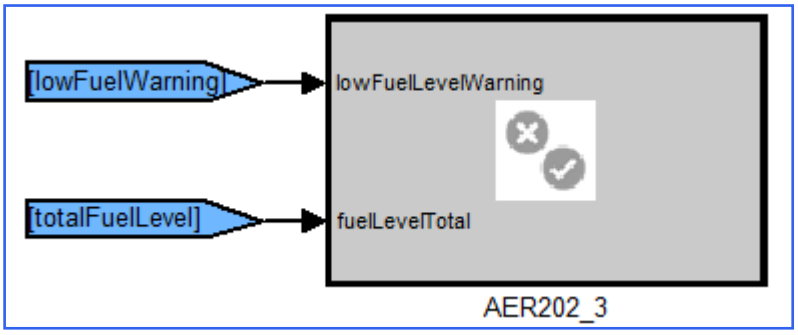

Figure 39: AER-07 (AER202_3).

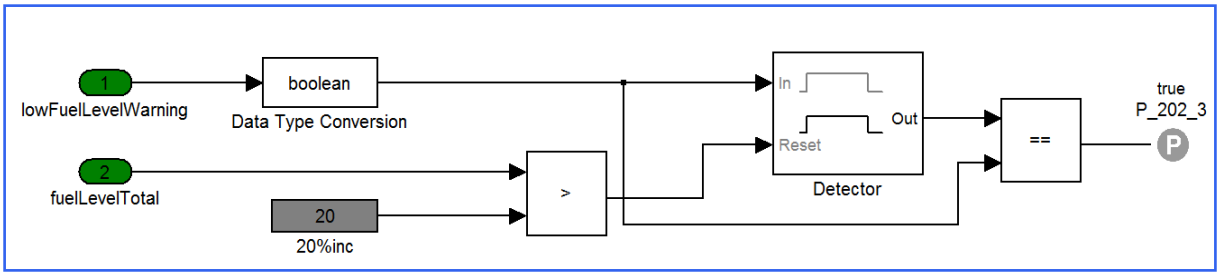

Figure 40: Property verification - AER-07 (AER202_3).

Figure 41 and Figure 42 represent how the requirement AER-08 (AER_202_5) is transformed in simulink model later to be verified by Simulink Design Verifier. "The requirement is to verify that the output signal lowFuelLevelWarning should have initial value 0 (false)." [15].

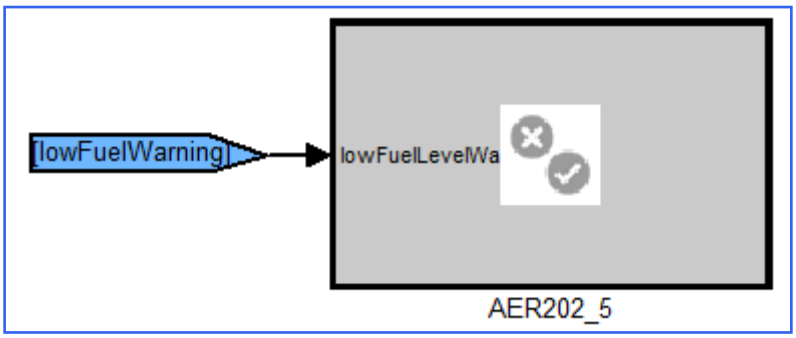

Figure 41: AER-08 (AER202_5).

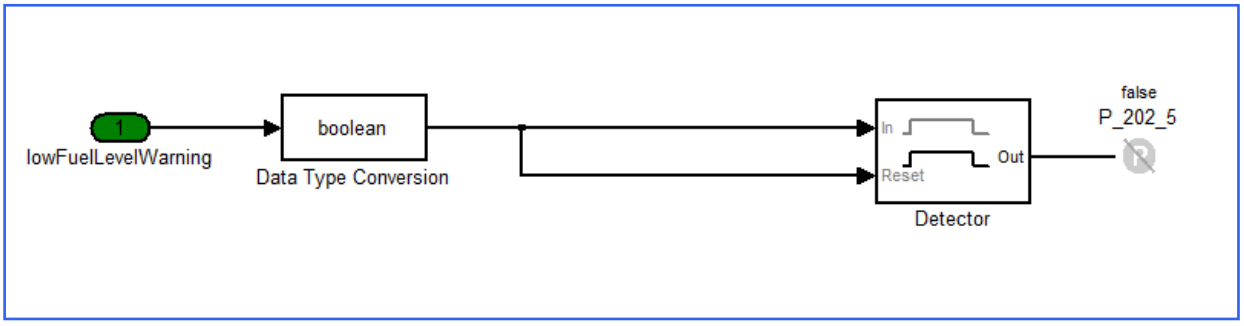

Figure 42: Property verification - AER-08 (AER202_5).

7.5. Results of Property Verification

In Simulink Design Verifier while proving the properties of system either the properties will be satisfied by the system model or there will be some scenario for which the property will be falsified and counter example will be generated.

Simulink Design Verifier checks all possible inputs against all possible paths and after testing all the possible scenarios generate the results whether the property is satisfied or not. If the property is satisfied then a detailed report is generated. And if the property is falsified then counter example is generated by the system along with the Harness model.

Following are the reports and other related artifacts that are generated by Simulink Design Verifier in property proving. First let us consider one of the requirements to show how property proving works. When user starts the property verification for requirement e.g. AER202_03 by clicking on the corresponding button Simulink Design Verifier starts verifying the property and performs a number of steps. If the property is satisfied then the verification block for requirement becomes green as shown in the figure 43 below.

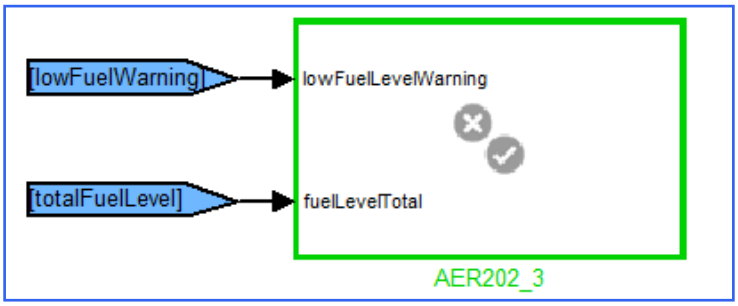

Figure 43: Requirement verification - AER-07 (AER202_3).

Following is the report that is generated when the specified property is satisfied.

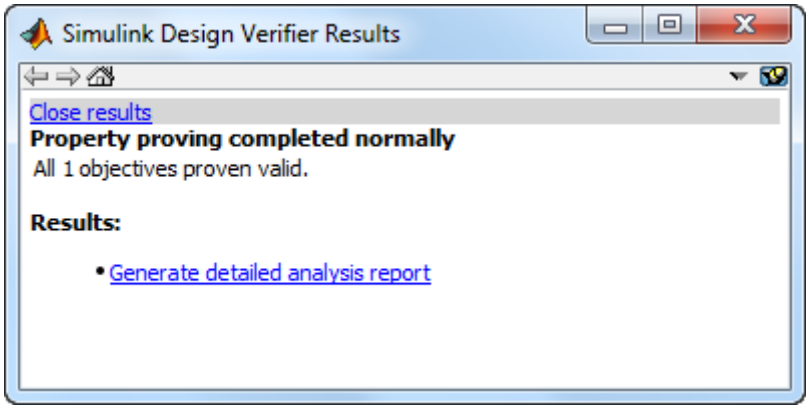

Figure 44: Property proving result window.

By clicking on *Generate detailed analysis report* link present in above figure 44 a detailed report of property verification is generated as shown in the figure 45 below.

Simulink Design Verifier Report

\\global.scd.scania.com\home\Se\121\msunr7\Documents\MATLAB\sldv_output\fuel_verification_0\fuel_verification_0_replacement14.mdl

msunr7

26-Jun-2012 09:56:17

Innehållsförteckning

Kapitel 1. Summary

Analysis Information

Model: fuel_verification_0
Replacement Model: \\global.scd.scania.com\home\Se\121\msunr7\Documents\MATLAB\sldv_output\fuel_verification_0\fuel_verification_0_replacement14.mdl
Mode: PropertyProving
Status: Completed normally
Analysis Time: 13s

Objectives Status

Number of Objectives: 1
Objectives Proven Valid: 1

Kapitel 2. Analysis Information

Innehållsförteckning

Model Information

File: fuel_verification_0
Version: 1.1142
Time Stamp: Tue Jun 26 09:49:19 2012
Author: Ali

Analysis Options

Mode: PropertyProving
Proving Strategy: ProveWithViolationDetection
Maximum Violation Steps: 2 time steps
Proof Assumptions: UseLocalSettings
Test Objectives: UseLocalSettings
Assertions: UseLocalSettings
Maximum Processing Time: 500s
Block Replacement: off
Parameters Analysis: off
Save Data: on
Save Harness: off
Save Report: off

Constraints

Analysis Constraints

Name	Analysis Constraint
input_assumptions/signals_assumptions/fuelLevel	60
input_assumptions/signals_assumptions/fuelRate	30
input_assumptions/signals_assumptions/oldFuelVolume	346
input_assumptions/params_assumptions/Assumption	10
input_assumptions/params_assumptions/Assumption1	450
input_assumptions/params_assumptions/Assumption2	450
input_assumptions/params_assumptions/Assumption3	728

Approximations

Simulink Design Verifier performed the following approximations during analysis. These can impact the precision of the results generated by Simulink Design Verifier. Please see the product documentation for further details.

#	Type	Description
1	Rational approximation	The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic.

Kapitel 3. Proof Objectives Status

Innehållsförteckning

Objectives Proven Valid

#	Type	Model Item	Description	Counterexample
1	Proof objective	AER202_3/P_202_3	Objective: T	n/a

Kapitel 4. Properties

Innehållsförteckning

AER202_3/P_202_3

Summary

Model Item: AER202_3/P_202_3
Property: Objective: T
Status: Proven valid

Figure 45: Property proving detailed report.

Let us consider one of the requirements to show how property proving works in case if the property is falsified. When user starts the property verification for requirement e.g. AER201_12 by clicking on the corresponding button Simulink Design Verifier starts verifying the property and performs a number of steps. If the property is not satisfied by the system model then the verification block for requirement becomes red as shown in the figure 46 below.
Following figures present the reports that are generated when property is falsified.

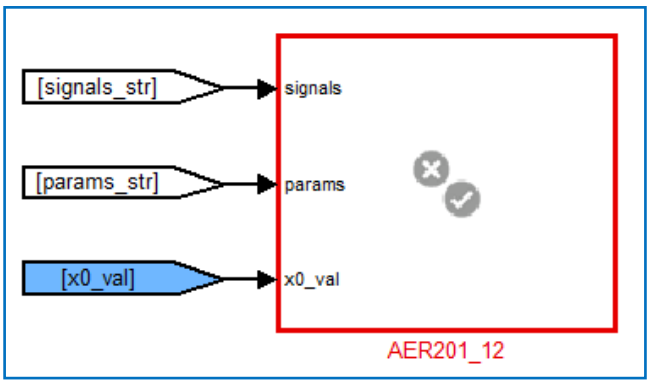

Figure 46: Requirement verification - AER-02 (AER201_12).

Following figure 47 presents the report that is generated when the property is falsified.

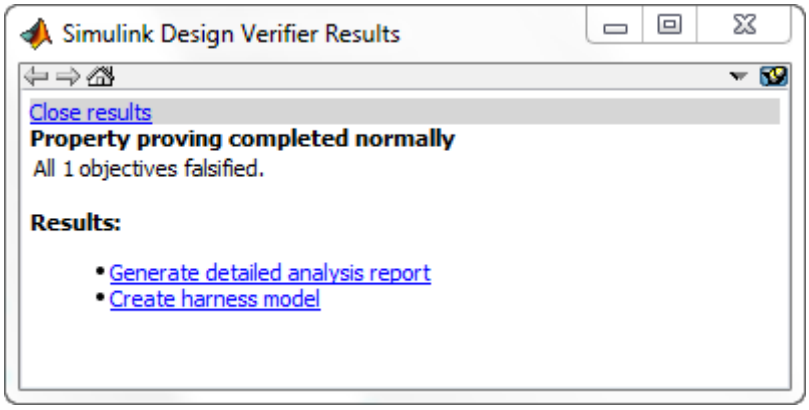

Figure 47: Property proving result window.

By clicking on *Generate detailed analysis report* link present in above figure 47 a detailed report of property verification is generated as shown in the figure 48 below.

Simulink Design Verifier Report

\\global.scd.scania.com\home\Se\121\msunr7\Documents\MATLAB\sldv_output\fuel_verification_0\fuel_verification_0_replacement15.mdl

msunr7

26-Jun-2012 10:16:48

Innehållsförteckning

Kapitel 1. Summary

Analysis Information

Model: fuel_verification_0
Replacement Model: \\global.scd.scania.com\home\Se\121\msunr7\Documents\MATLAB\sldv_output\fuel_verification_0\fuel_verification_0_replacement15.mdl
Mode: PropertyProving
Status: Completed normally
Analysis Time: 0s

Objectives Status

Number of Objectives: 1
Objectives Falsified with Counterexamples: 1

Kapitel 2. Analysis Information

Innehållsförteckning

Model Information

File: fuel_verification_0
Version: 1.1143
Time Stamp: Tue Jun 26 10:11:28 2012
Author: Ali

Analysis Options

Mode: PropertyProving
Proving Strategy: FindViolation
Maximum Violation Steps: 500 time steps
Proof Assumptions: UseLocalSettings
Test Objectives: UseLocalSettings
Assertions: UseLocalSettings
Maximum Processing Time: 500s
Block Replacement: off
Parameters Analysis: off
Save Data: on
Save Harness: off
Save Report: off

Constraints

Analysis Constraints

Name	Analysis Constraint
input_assumptions/signals_assumptions/fuelLevel	[1, 100]
input_assumptions/signals_assumptions/fuelRate	30
input_assumptions/signals_assumptions/oldFuelVolume	346
input_assumptions/params_assumptions/Assumption	10
input_assumptions/params_assumptions/Assumption1	450
input_assumptions/params_assumptions/Assumption2	450
input_assumptions/params_assumptions/Assumption3	728

Approximations

Simulink Design Verifier performed the following approximations during analysis. These can impact the precision of the results generated by Simulink Design Verifier. Please see the product documentation for further details.

#	Type	Description
1	Rational approximation	The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic.

Kapitel 3. Proof Objectives Status

Innehållsförteckning

Objectives Falsified with Counterexamples

#	Type	Model Item	Description	Counterexample
1	Proof objective	AER201_12/P_201_12	Objective: T	1

Kapitel 4. Properties

Innehållsförteckning

AER201_12/P_201_12

Summary

Model Item: AER201_12/P_201_12
Property: Objective: T
Status: Falsified

Counterexample

Time	0
Step	1
Inputs_Signals.fuelRate_Val_F32	30
Inputs_Signals.fuelLevelSensor_Val_F32	48.5275
Inputs_Signals.pBrakeAppliedHigh_Val_B	0
Inputs_Signals.oldFuelVolume_Val_F32	346
Inputs_Params.fuelTankSizeLeft_F32	450
Inputs_Params.fuelTankSizeRight_F32	450
Inputs_Params.mantleVolumeTot_F32	728
Inputs_Params.upLowFuelLevelInd_Val_U08	10

Figure 48: Property proving detailed report.

By clicking on *Generate harness model* link present in above figure 47 harness model is generated as shown in the figure 49 below. Harness model allows the user to simulate a copy of original model using the test cases or counterexamples that are generated by Simulink Design Verifier.

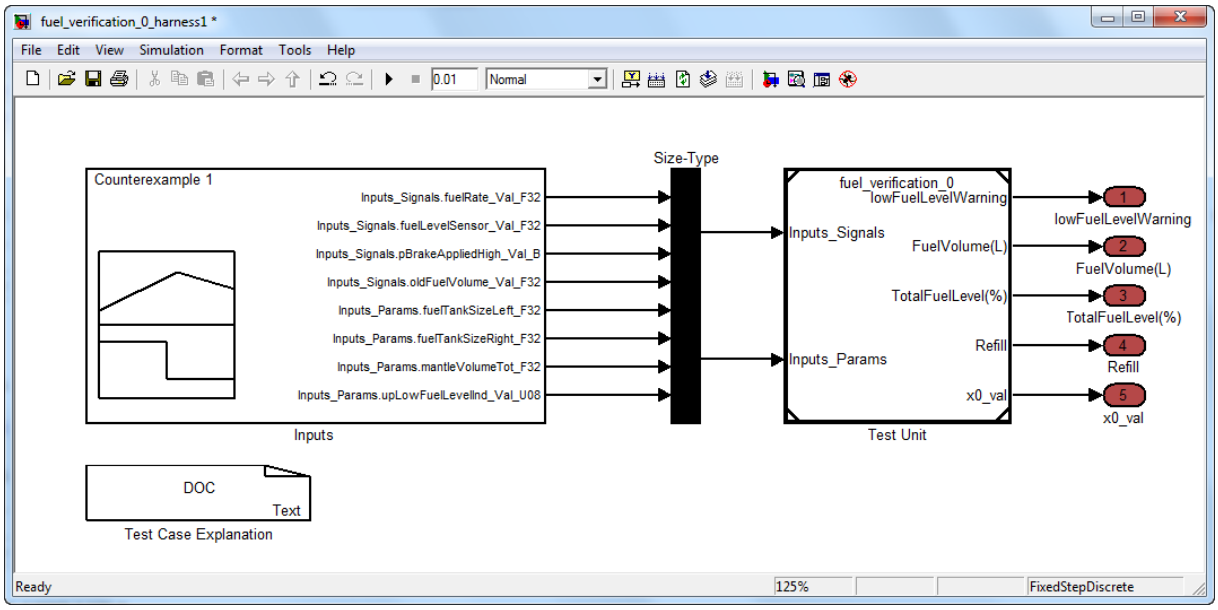

Figure 49: Harness model for AER-02 (AER201_12).

Harness model contains the inputs, size-type, test unit and test case explanation blocks. Inputs are as shown in the figure 50 and the test unit contains a copy of the whole model of fuel level display system.

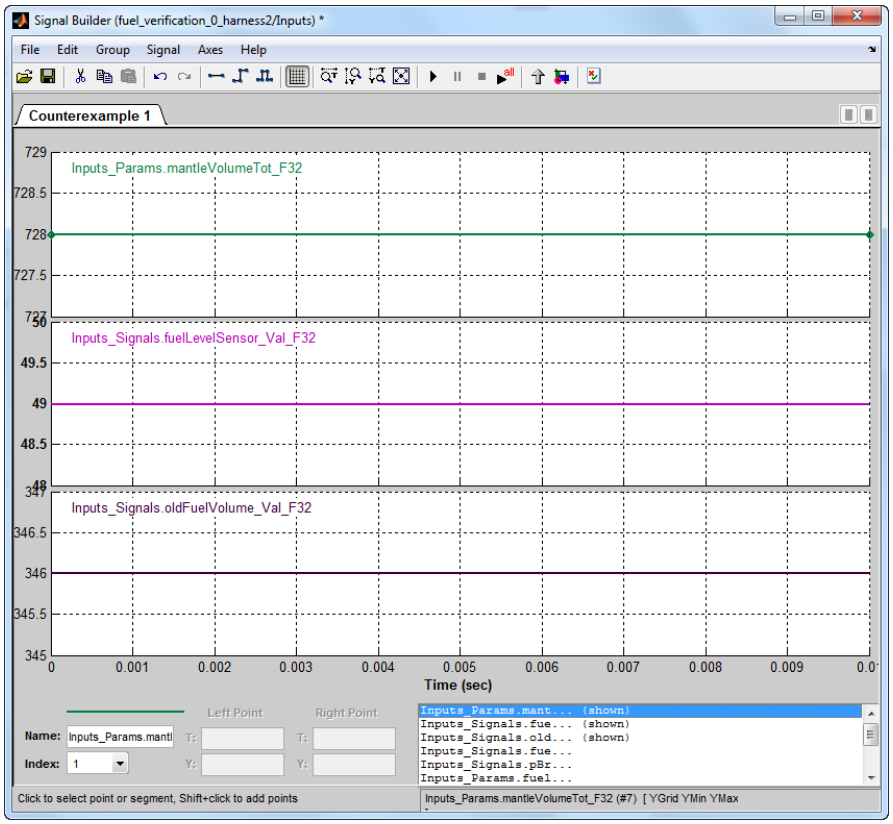

Figure 50: Input signals for AER-02 (AER201_12).

Harness model is only generated in case when objective is falsified. The harness model contains the following items:

Inputs
Inputs block also called input signal builder block contains signals that comprise the test cases or counterexamples that Simulink Design Verifier generated. This block contains signals only for input signals that are used in the model. If an input signal has no effect on the output of the model, that signal is not included in this block shown in harness model. By double-clicking the Inputs block the signal builder dialog box can be viewed and also its signals as shown in the figure 50 can be seen that present the signal builder block and the signals for verification of property. Each signal group represents a unique test case or counterexample. In the signal builder dialog box, each tab can be selected to view the signals associated with a particular test case or counterexample.

Size-Type
Size-type subsystem block is responsible for transmitting the signals from the Inputs block to the Test Unit block. Size-type subsystem block is responsible for verifying that the signals are of the appropriate size and data type as the system in Test Unit block is expecting them.

Test Unit
This Subsystem block contains a copy of the original model that Simulink Design Verifier analyzed. Original model of fuel level display system in Simulink Design Verifier is present in section 9.1.

Test Case Explanation
The test case explanation block contains a document that documents the test cases or counter examples generated by Simulink Design Verifier. By double-clicking this block the description of each test case or the counterexample can be viewed. This block lists either the test objectives that each test case achieves or the proof objectives that each counterexample falsifies.

All the requirements mentioned in table 5 for fuel level display system are proved to be satisfied by the model of fuel level display system in Simulink Design Verifier.

7.6. General Design error detection

This section describes about the verification that is done in Simulink Design Verifier software to identify the design errors in system model without focusing on any specific requirement. The previous section 9 checks whether particular requirements and properties are satisfied by the system model or not, but this section does the verification in general for identifying the design errors like integer overflow errors, division by zero errors, and check specified intermediate minimum and maximum values.
The version of Matlab that is used during this thesis is Matlab R2011b. In this version of Matlab the Simulink Design Verifier software supports the automatic detection of following kind of design errors [15].

1. Integer overflow
2. Division by zero
3. Check specified intermediate minimum and maximum values

Detection of these kinds of design errors is supported automatically in Simulink Design Verifier software. Users can simply select the type of design errors that they want to detect in their model and Simulink Design Verifier software will automatically run the process and check whether there are any design error of selected type in the model or not. Following figure 51 shows the dialog box that can be used to select which kind of design errors users want to detect in their model.

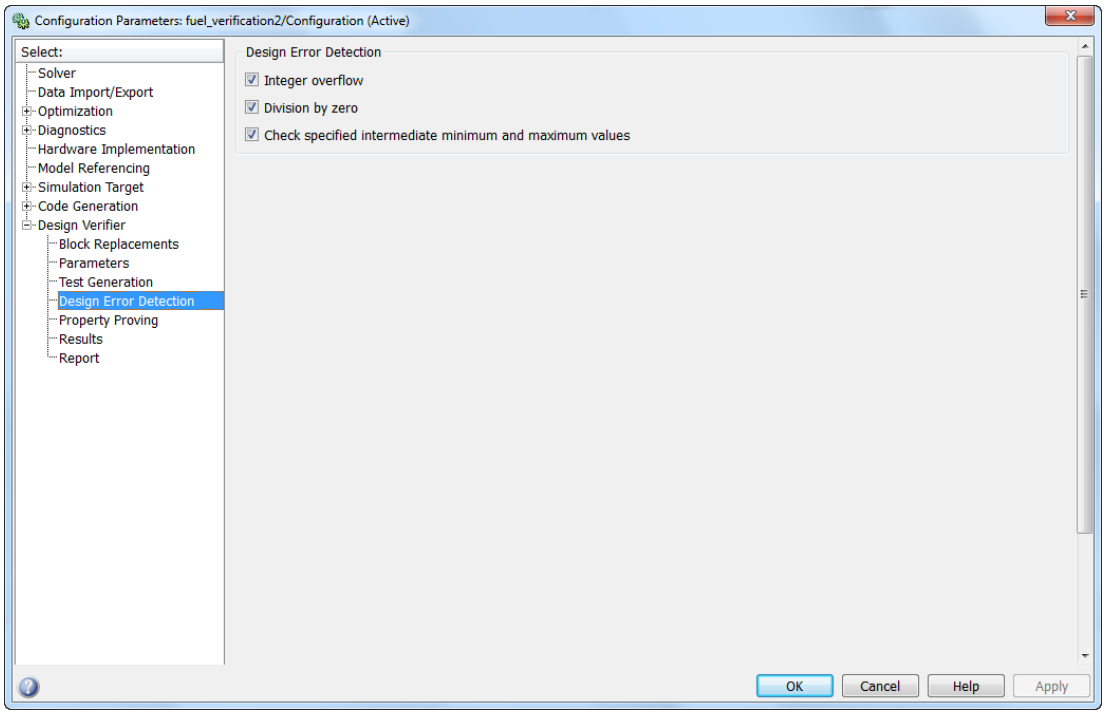

Figure 51: Configuration for Design error detection.

Model of fuel level display system is checked against all the three kind of design errors and it is established that there are no such errors present in the system model of fuel level display system. After checking all type of design error in the model when analysis is started, Simulink Design Verifier generates the analysis report present in figure 53 to figure 55 and highlights the system model with green color as shown in the figure 52 below, which means no overflow or division-by-zero errors are detected during analysis and also the analysis did not detect any intermediate or output signals outside the range of user-specified constraints.

Model can also be highlighted in red, orange or grey color after analysis. Red means that there are design errors of specified king in the system model. Orange represents for at least one objective, the analysis could not determine if there was design error or not. This can happen if the analysis times out, or the software cannot determine if an error occurred or not. System model highlighted in grey show that this particular model was not the part of analysis [15].

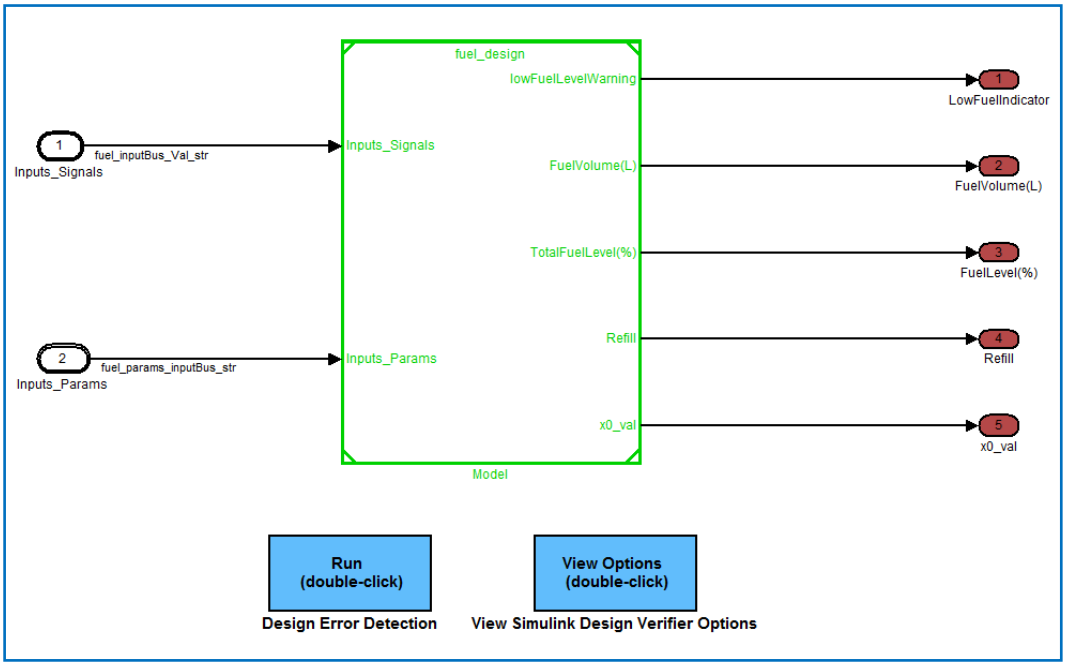

Figure 52: System model after design error detection analysis.

After completing the analysis, Simulink Design Verifier software generates the following dialog box present in figure 53, which represents the overall status of design error detection activity. In design error detection activity of fuel level display system model all the objectives are proven valid and no design error of specified type is identified in the fuel level display system model.

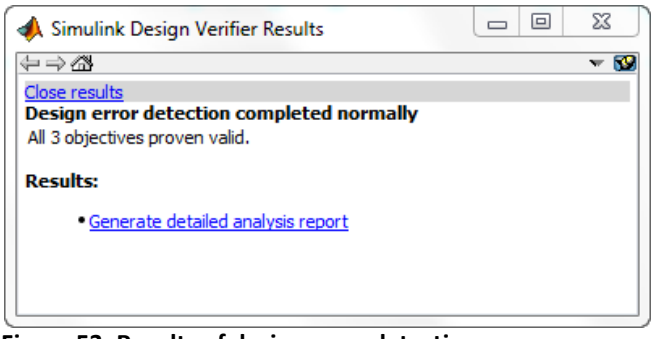

Figure 53: Results of design error detection.

After clicking on the generate detailed analysis report link in the above figure 53 detailed analysis report can be viewed. Following figure 54 and figure 55 represents the detailed analysis report for design error detection in case of fuel level display system model.

Simulink Design Verifier Report

\\global.scd.scania.com\home\Se\121\msunr7\Documents\MATLAB\sldv_output\fuel_verification2\fuel_verification2_replacement3.mdl

msunr7

05-Jul-2012 13:33:33

Innehållsförteckning

Kapitel 1. Summary

Analysis Information

Model: fuel_verification2
Replacement Model: \\global.scd.scania.com\home\Se\121\msunr7\Documents\MATLAB\sldv_output\fuel_verification2\fuel_verification2_replacement3.mdl
Mode: DesignErrorDetection
Status: Completed normally
Analysis Time: 43s

Objectives Status

Number of Objectives: 3
Objectives Proven Valid: 3

Kapitel 2. Analysis Information

Innehållsförteckning

Model Information

File: fuel_verification2
Version: 1.412
Time Stamp: Thu Jul 05 13:23:59 2012
Author: Ali

Analysis Options

Mode:	DesignErrorDetection
Detect integer overflow:	on
Detect division by zero:	on
Check specified intermediate minimum and maximum values:	on
Maximum Processing Time:	800s
Block Replacement:	off
Parameters Analysis:	off
Save Data:	on
Save Harness:	on
Save Report:	off

Approximations

Simulink Design Verifier performed the following approximations during analysis. These can impact the precision of the results generated by Simulink Design Verifier. Please see the product documentation for further details.

#	Type	Description
1	Rational approximation	The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic.

Kapitel 3. Design Error Detection Objectives Status

Innehållsförteckning

Objectives Proven Valid

#	Type	Model Item	Description	Test Case
25	Division by zero	Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/Divide1	Division by zero	n/a
86	Overflow	Model/Algorithm/EvaluateParkingBrake/evaluateParkingBrakeApplied/ SFunction	Overflow	n/a
87	Overflow	Model/Algorithm/FuelLevelEstimationAlgorithm/AlgoritmResetCalc/RefuelDetection/ SFunction	Overflow	n/a

Figure 54: Detailed design error detection analysis report1.

Kapitel 4. Derived Ranges

Signal	Derived Ranges
LowFuelIndicator- outport 1	[0..1]
FuelVolume(L)- outport 1	[0..10000]
FuelLevel(%)- outport 1	[0..100]
Refill- outport 1	[F..T]
x0_val- outport 1	[-Inf..Inf]
Model/Algorithm/EvaluateParkingBrake/evaluateParkingBrakeApplied/ SFunction - outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/Precalculations/Gain2- outport 1	[-3.4029e+036..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/Precalculations/Divide2- outport 1	[-Inf..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/Add1- outport 1	[-Inf..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/Abs- outport 1	[0..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/CompareToConstant/Constant- outport 1	90
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/CompareToConstant/REL_OP- outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/Gain2- outport 1	[-3.4029e+037..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/RelationalOperator2/REL_OP- outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/CompareToConstant/Merge- outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/RelationalOperator2/Merge- outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/LogicalOperator1- outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/Switch1- outport 1	[-Inf..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/AvoidDivideByZero/Constant1- outport 1	[1.1754e-038..1.1755e-038]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/Gain3- outport 1	[-3.4029e+035..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/AvoidDivideByZero/MinMax- outport 1	[1.1754e-038..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/Constant- outport 1	0
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/Gain2- outport 1	[-3.4029e+035..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/KalmanAlgoritm/DiscreteTimeIntegrator- outport 1	[0..10]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/Divide1- outport 1	[0..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/Gain1- outport 1	[0..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/RoundUpwards/Replacement_val- outport 1	100
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/RoundUpwards/CompareToConstant/Constant- outport 1	[99.5..99.501]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/RoundUpwards/CompareToConstant/REL_OP- outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/RoundUpwards/CompareToConstant/Merge- outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/RoundUpwards/Switch- outport 1	[0..100]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/MinMax- outport 1	[0..100]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/Gain1- outport 1	[0..10000]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/Gain3- outport 1	[-3.4029e+035..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/KalmanAlgoritm/UnitDelay4- outport 1	[0..10]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/KalmanAlgoritm/Add1- outport 1	[-3.4029e+035..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/KalmanAlgoritm/Constant- outport 1	[1.0779e-005..1.078e-005]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/KalmanAlgoritm/Constant1- outport 1	[0.0099999..0.01]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/KalmanAlgoritm/Divide1- outport 1	[0.0010779..0.001078]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/KalmanAlgoritm/Divide- outport 1	[-3.6683e+032..Inf]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/ScaleFuelConsumption/Saturation- outport 1	[0..500]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/ScaleFuelConsumption/Gain- outport 1	[0..0.00013889]
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/KalmanAlgoritm/Add2- outport 1	[-3.6683e+032..Inf]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/CompareToConstant4/Constant- outport 1	20
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/CompareToConstant4/REL_OP- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant/Constant- outport 1	10
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant/REL_OP- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/Add- outport 1	[-Inf..Inf]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant1/Constant- outport 1	900
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant1/REL_OP- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant5/Constant- outport 1	7
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant5/REL_OP- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant/Merge- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant1/Merge- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant5/Merge- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/Switch- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/CompareToConstant4/Merge- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/LogicalOperator1- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/RouteOutput/CompareToConstant4/Constant- outport 1	10
Model/Algorithm/LowFuelLevelWarning/RouteOutput/CompareToConstant4/REL_OP- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/RouteOutput/Constant- outport 1	0
Model/Algorithm/LowFuelLevelWarning/RouteOutput/DataTypeConversion1- outport 1	0
Model/Algorithm/LowFuelLevelWarning/RouteOutput/CompareToConstant4/Merge- outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/RouteOutput/Switch- outport 1	[0..1]
Model/Algorithm/FuelLevelEstimationAlgorithm/AlgoritmResetCalc/RefuelDetection/ SFunction - outport 1	[F..T]
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/CompareToConstant/If Action Subsystem/Constant- outport 1	T
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/CompareToConstant/If Action Subsystem1/Constant- outport 1	F
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/RelationalOperator2/If Action Subsystem/Constant- outport 1	T
Model/Algorithm/FuelLevelEstimationAlgorithm/CalculatCurrentVolumeLevels/LevelsRegularTanks/Find_x0/RelationalOperator2/If Action Subsystem1/Constant- outport 1	F
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/RoundUpwards/CompareToConstant/If Action Subsystem/Constant- outport 1	T
Model/Algorithm/FuelLevelEstimationAlgorithm/KalmanObserverEstimation/FuelLevelCalculation/ConvertToPercent/RoundUpwards/CompareToConstant/If Action Subsystem1/Constant- outport 1	F
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/CompareToConstant4/If Action Subsystem/Constant- outport 1	T
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/CompareToConstant4/If Action Subsystem1/Constant- outport 1	F
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant/If Action Subsystem/Constant- outport 1	T
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant/If Action Subsystem1/Constant- outport 1	F
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant1/If Action Subsystem/Constant- outport 1	T
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant1/If Action Subsystem1/Constant- outport 1	F
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant5/If Action Subsystem/Constant- outport 1	T
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/WarningLevel/CompareToConstant5/If Action Subsystem1/Constant- outport 1	F
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/SetState/ SFunction - outport 1	[F..T]
Model/Algorithm/LowFuelLevelWarning/RouteOutput/CompareToConstant4/If Action Subsystem/Constant- outport 1	T
Model/Algorithm/LowFuelLevelWarning/RouteOutput/CompareToConstant4/If Action Subsystem1/Constant- outport 1	F
Model/Algorithm/LowFuelLevelWarning/CalculateWarning/DataTypeConversion- outport 1	[0..1]
Model/Algorithm/LowFuelLevelWarning/RouteOutput/DataTypeConversion2- outport 1	[0..1]

Figure 55: Detailed design error detection analysis report2.

8. UPPAAL Model and Verification Results

UPPAAL is a model checking tool in which real time systems can be modeled, validated and verified. UPPAAL is recommended for the system that can be modeled as timed automatons. Systems in UPPAAL are modeled as states and transitions between the states. Timed automaton in UPPAAL communicate by using the channels and shared data structures. Following figure 56 presents the overall architecture of fuel level display system designed in UPPAAL.

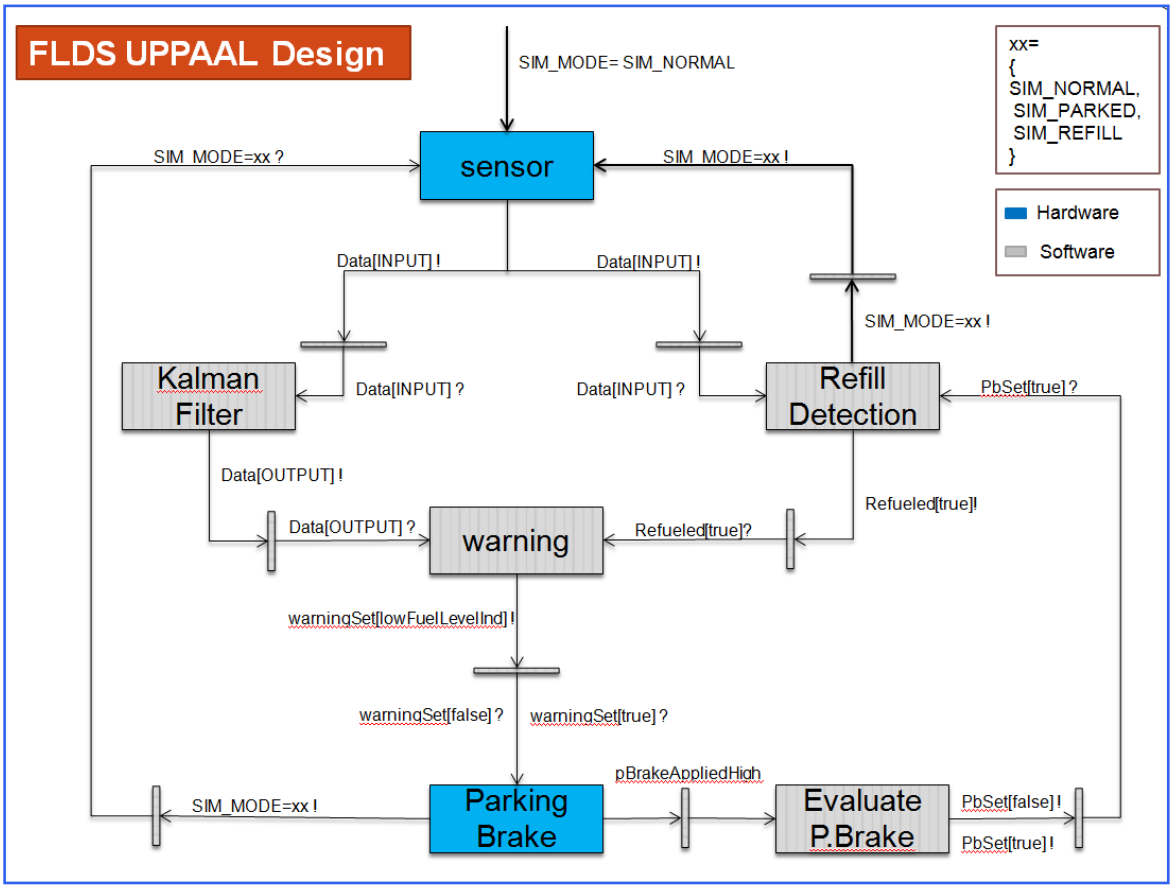

Figure 56: Fuel level display system - UPPAAL Architecture Design.

Model of fuel level display system in UPPAAL consists of six automaton fuel level sensor, fuel level estimation algorithm, evaluate parking brake, parking brake, refill detection and low fuel level warning. Main tree structure of fuel level display system in UPPAAL is presented in figure 57 showing the entire automaton and their local declarations along with global declarations and system declarations sections.

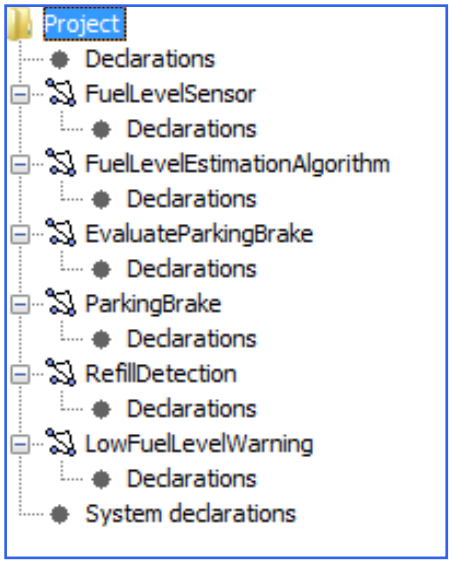

Figure 57: UPPAAL Model Navigation Tree.

Following table 18 contains the global declarations for UPPAAL model of fuel level display system.

// Global declarations for UPPAAL model.

```
const int N=2;
const int INPUT =0;
const int OUTPUT=1;
urgent chan refueled[N];
broadcast chan pbSet[N],data[N],  warningSet[N];

const int SC_FACTOR1 =44; //Scaling Factor, everything multiplied by 44
const int SC_FACTOR2 = 7; //Scalling factor for fuelLevelTot to avoid loss of precision due to rounding of integer
const int TOT_SCALE = SC_FACTOR1 * SC_FACTOR2;

const int LARGE_TANK_LIMIT=900;
const int LOW_FUEL_LEVEL_LARGE_TANKS = 7*TOT_SCALE;
const int LOW_FUEL_LEVEL_NORMAL_TANKS = 10*TOT_SCALE;

//System input Signals (simulink signal variable names are preserved)
const int fuelRate=SC_FACTOR1 * 30;
int fuelLevelSensor= SC_FACTOR1 * 1; //FuelLevelSensor value is multiplied by SC_FACTOR1
const int oldFuelVolume=SC_FACTOR1 * 1;
bool pBrakeAppliedHigh=false;

//System input Parameters (simulink parameter variable names are preserved)
const int fuelTankSizeLeft = 450;
const int fuelTankSizeRight = 450;
const int mantleVolumeTot = SC_FACTOR1 * 728;
const int upLowFuelLevelInd = 10;

//intermediate values
int x0 = 0; //startup state for kalman (just assumed for simulation)
int levelInMainPartRawLitre = 0;
int fuelConsumptionTot;
int newVolumeState;
bool refuelDetected=false;
const int PER_MAX = 100 * SC_FACTOR1;
const int PER_MIN = 0;
const int tanksizes = fuelTankSizeLeft + fuelTankSizeRight;

//system outputs (simulink output variable names are preserved)
```

```
int lowFuelLevelInd = 0; //output warning indicator
int fuelVolume = 0; //output fuel level in litres
int fuelLevelTot = 0; // output fuel level in %age

//system clock (not used)
int sys_x;

//simulation configuration
const int SIM_FL_INCREASE=32*SC_FACTOR1;
const int SIM_NORMAL=10;
const int SIM_PARKED=20;
const int SIM_REFILL=30;
const bool NO_REFILL=true;

int SIM_MODE = SIM_NORMAL;
```

Table 18: Global declarations for UPPAAL model.

Figure 58 presents the automata for fuel level sensor. It handles the different states of fuel level sensor. Fuel level sensor can be in normal state, parked state or refill state. States are also created to keep the fuel level values between 0 and 100. Minimum value of the fuel level can be 0 and maximum value can be 100.

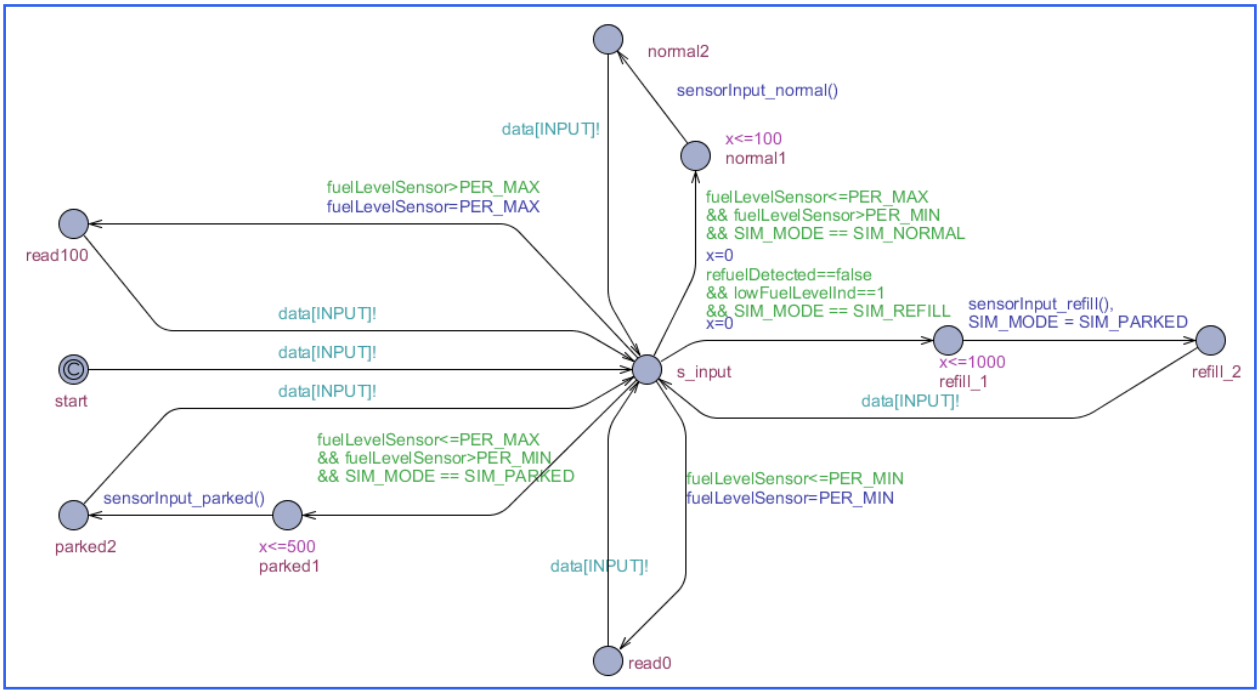

Figure 58: Automata for fuel level sensor.

Table 19 contains the local declarations for fuel level sensor automata.

// Local declarations for fuel level sensor automata.

```
clock x;
void sensorInput_normal()
{
                fuelLevelSensor = fuelLevelSensor - (((((fuelRate*100)/3600))*PER_MAX)/mantleVolumeTot +1);
                /*if(pBrakeAppliedHigh) {SIM_MODE = SIM_PARKED;}
                else if (pBrakeAppliedHigh && lowFuelLevelInd ==1)
                else {SIM_MODE = SIM_NORMAL;}*/
}
```

```
void sensorInput_parked()
{
		fuelLevelSensor = fuelLevelSensor - ((((5*SC_FACTOR1*500)/3600))*PER_MAX)/mantleVolumeTot;
}
void sensorInput_refill()
{
		fuelLevelSensor = fuelLevelSensor + (SIM_FL_INCREASE);
}
```

Table 19: Local declarations for fuel level sensor automata.

Figure 59 presents the automata for fuel level estimation algorithm. It waits for the input from fuel level sensor. When it receives the input from fuel level sensor then it calls three functions currentVolumeLevels(), x_= get_x0(), kalman_output() code of these functions is present in table 20. If refuel is detected then it keeps on doing the same thing. And when refuel is not detected then it calls the three functions currentVolumeLevels(), x_=newState(), kalman_output() code of these functions is present in table 20.

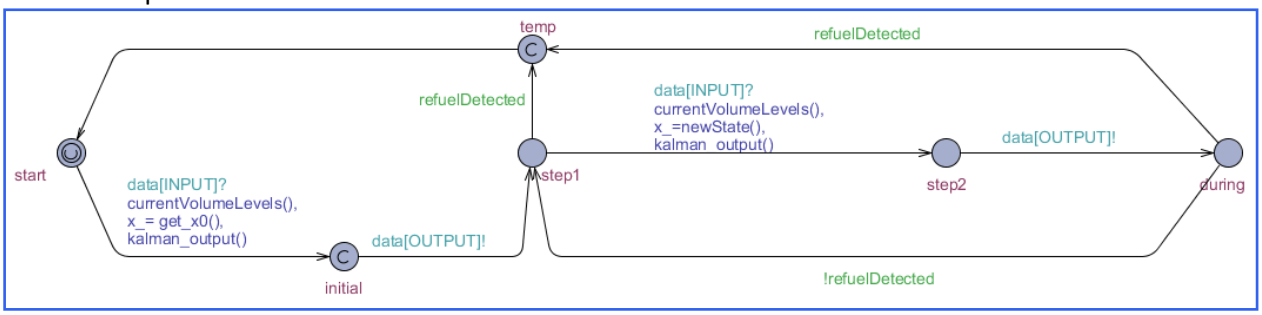

Figure 59: Automata for fuel level estimation algorithm.

Table 20 contains the local declarations for fuel level estimations algorithm automata.

```
// Local declarations for fuel level estimation algorithm automata.
clock x;
int x_;
const int H_TO_SEC = 3600;
const int K_NUM = 5; //numerator  (k = 1.078e-005 [kalman gain])
const int K_DEN=10000; //denumerator
const int GCD = 10000;
const int NEAR_TOP_PERCENTAGE=90;
const int VOLUME_DIFF_ACCEPTED=10;

int newState()
{
		int fr = fuelRate;
		int rfl = levelInMainPartRawLitre;
		if(levelInMainPartRawLitre>=0)
		{
				newVolumeState = ((x_ * GCD) - ((100*fr*GCD)/H_TO_SEC) + (K_NUM*(rfl-x_)))/K_DEN;
		}

		//limit the output volume state between 0 and mantleVolumeTot
		if (newVolumeState<0) {newVolumeState=0;fuelVolume=0;}
		else if
(newVolumeState>mantleVolumeTot){newVolumeState=mantleVolumeTot;fuelVolume=mantleVolumeTot;}
		else {fuelVolume = newVolumeState;}
		return newVolumeState;
}
void currentVolumeLevels()
{
		levelInMainPartRawLitre = (fuelLevelSensor * mantleVolumeTot)/PER_MAX;
}
void kalman_output()
```

```
{
		fuelLevelTot = (fuelVolume * PER_MAX * SC_FACTOR2)/mantleVolumeTot; //convert to a scaled
percentage
}
int get_x0()
{
		int volDiffBetweenEstAndMeas;
		int val = levelInMainPartRawLitre - oldFuelVolume;
		if(val<0)
				volDiffBetweenEstAndMeas = val * (-1);
		else
				volDiffBetweenEstAndMeas = val;

		if (volDiffBetweenEstAndMeas > ((VOLUME_DIFF_ACCEPTED * mantleVolumeTot)/100) ||
(fuelLevelSensor>(NEAR_TOP_PERCENTAGE*SC_FACTOR1)) || (refuelDetected==true))
				x0 = levelInMainPartRawLitre;
		else
				x0 = oldFuelVolume;
		newVolumeState = x0;
		fuelVolume = newVolumeState;
		return x0;
}
int abs(int val)
{
		if(val<0)
		return (val * (-1));
		else return val;
}
```

Table 20: Local declarations for fuel level estimation algorithm automata.

Figure 60 presents the automata for evaluating parking brake. Parking brake is not considered to be applied until it is continuously applied for 50 time units. If the parking brake is continuously applied for 50 time units then the value of parking brake is set to true. When the parking brake is released then its value is set to false.

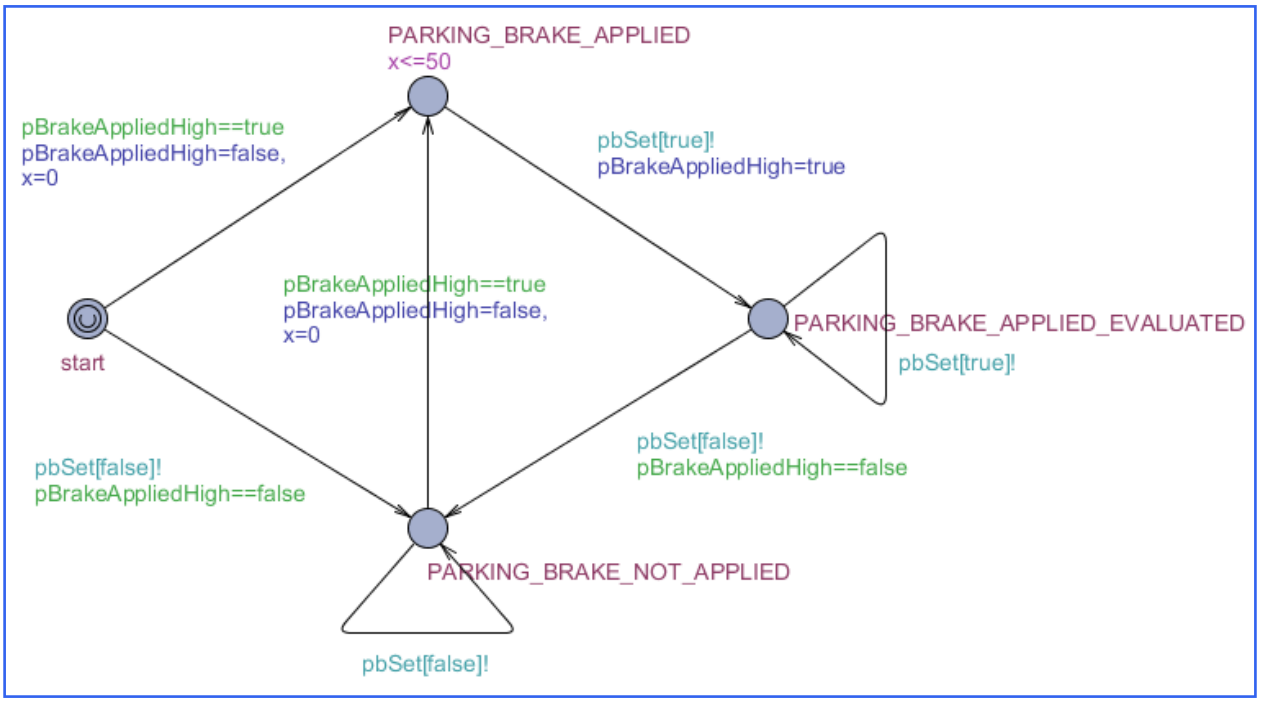

Figure 60: Evaluate parking brake automata.

Figure 61 contains the automata for parking brake. When the low fuel level warning is set to true and parking brake is not applied then it makes pBrakeAppliedHigh=true, x=0 and SIM_MODE = SIM_PARKED. And when parking brake is applied and low fuel level warning is set to false then it makes pBrakeAppliedHigh=false, x=0, SIM_MODE = SIM_NORMAL .

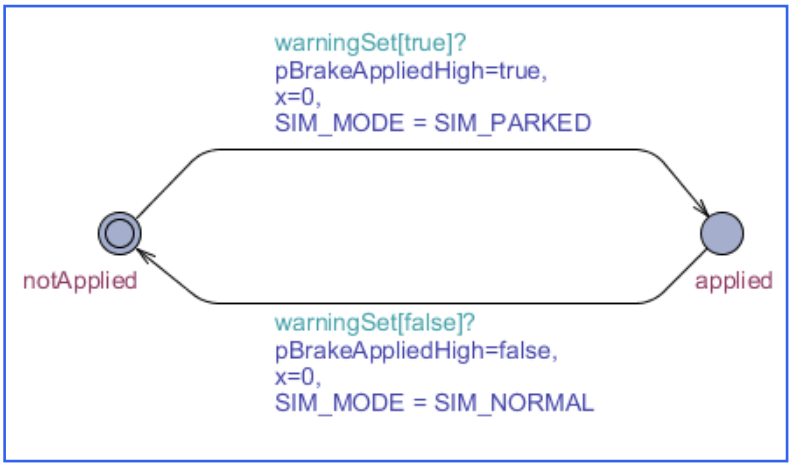

Figure 61: Parking brake automata.

EvaluateParkingBrake and ParkingBrake automaton only have "clock x;" declared in their local declaration sections.

Figure 62 contains the automata for refill detection. Before detecting the refill parking brake must be set to true. When the parking brake is applied only then refill can be detected. If the increased value of the fuel is not maintained until 500 time units then it means there is no refill and system goes back to the initial state. If the increased value of the fuel is maintained until 500 time units then it means there is a refill and startDetaction() function is called. Code for startDetection() function is present in table 21. Refill is considered valid only if the increase in the value is at least 30 percent otherwise there is no refill. And this 30 percent increase must also be held for at least 500 time units to ignore the sloshing otherwise it's not considered as refill and system goes back to the initial state. Condition() and Condition() functions are called at different states, code for both these functions is present in table 21.

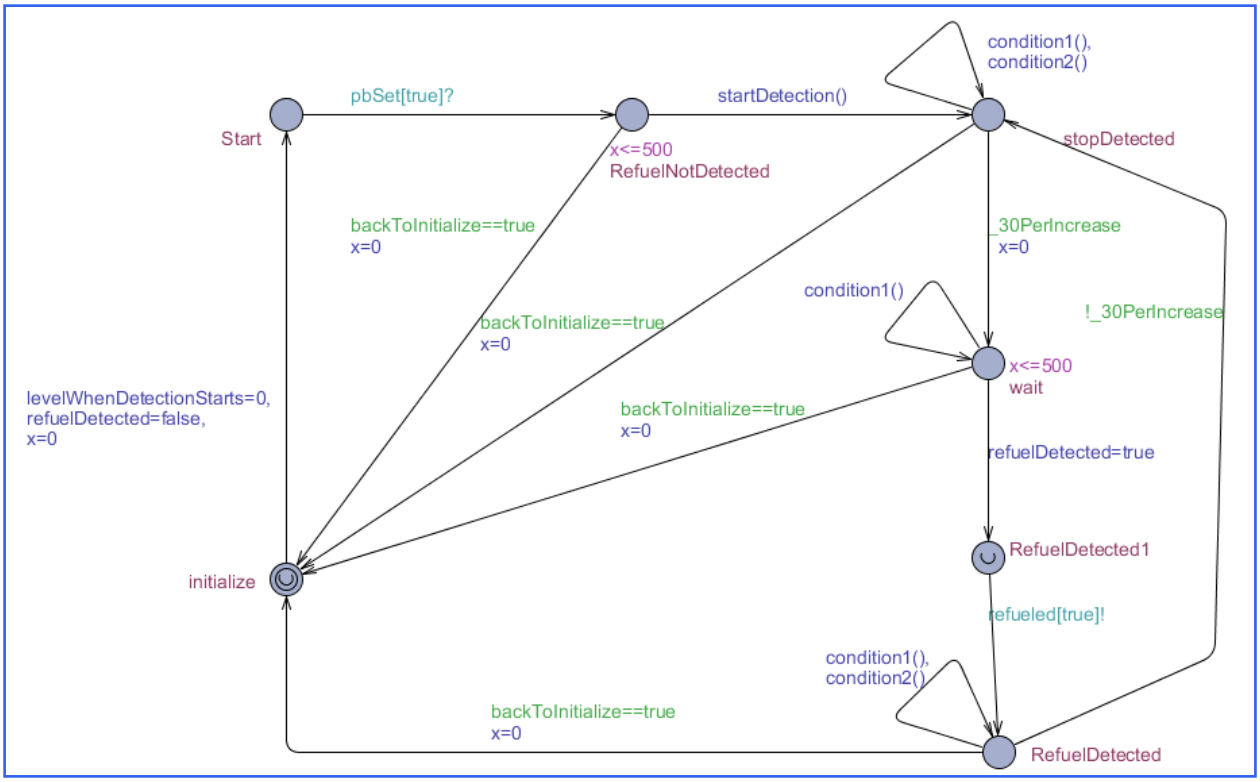

Figure 62: Automata for refill detection.

Table 21 contains the local declarations and code for refill detection automata.

```
// Local declarations for refill detection automata.
clock x;
int levelWhenDetectionStarts = 0;
bool backToInitialize;
bool _30PerIncrease;
void condition1()
{
        backToInitialize = (pBrakeAppliedHigh == false) || (fuelLevelSensor <
((90*levelWhenDetectionStarts)/100));
}
void condition2()
{
        if(fuelLevelSensor - levelWhenDetectionStarts >=(30*SC_FACTOR1))
                _30PerIncrease = true;
        else if(fuelLevelSensor - levelWhenDetectionStarts < (30*SC_FACTOR1))
                _30PerIncrease = false;
        else
                _30PerIncrease = false;
}
void startDetection()
{
        levelWhenDetectionStarts=fuelLevelSensor;
        if(!NO_REFILL)
        {
                SIM_MODE=SIM_REFILL;
        }
}
```

Table 21: Local declarations for refill detection automata.

Figure 63 contains the automata for low fuel level warning. If the value of fuel does below certain predefined value then the warning is activated and if the low fuel level warning indicator is on then

the warning is routed and displayed to the derived and it is kept on until a refill is detected. If the low fuel level warning indicator is off then the warning is not routed. Initial value of the low fuel level warning is set to false.

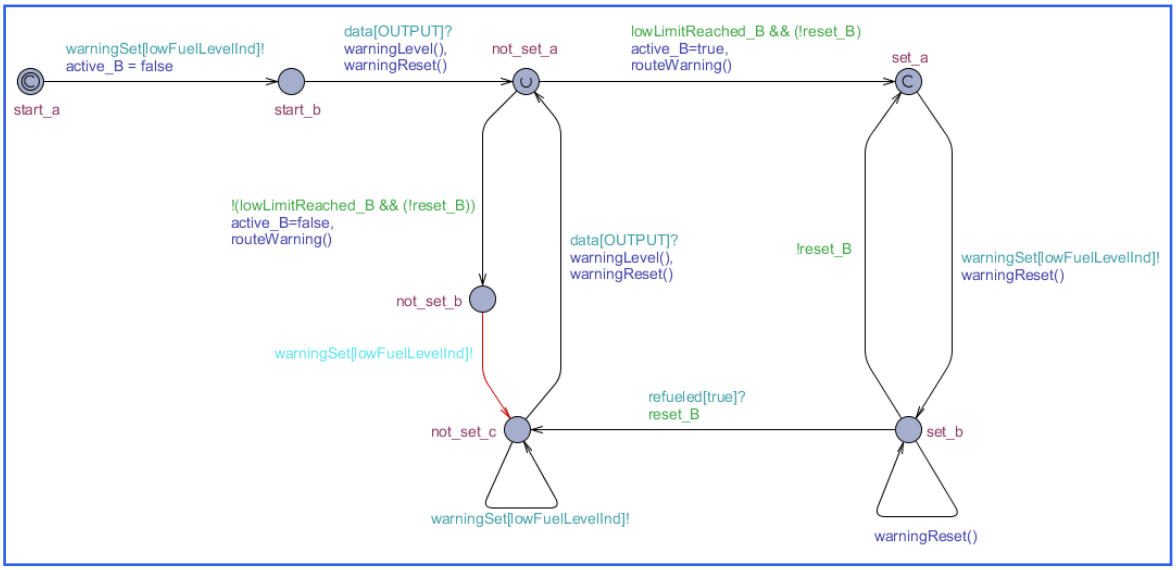

Figure 63: Low fuel level warning automata.

Table 22 contains the local declarations and code for low fuel level warning automata.

```
// Local declarations for low fuel level warning automata.
clock x;
bool lowLimitReached_B;
bool reset_B,active_B;
void warningLevel()
{
if (tanksizes>LARGE_TANK_LIMIT && fuelLevelTot < LOW_FUEL_LEVEL_LARGE_TANKS)
                lowLimitReached_B = true;
else if (tanksizes<=LARGE_TANK_LIMIT && fuelLevelTot < LOW_FUEL_LEVEL_NORMAL_TANKS)
                lowLimitReached_B = true;
else
                lowLimitReached_B = false;
}
void warningReset()
{
                if (fuelLevelTot > 20*TOT_SCALE || refuelDetected)
                {
                                reset_B=true;
                }
                else
                                reset_B=false;
}

void routeWarning()
{
                if (upLowFuelLevelInd ==10)
                {
                                lowFuelLevelInd=active_B;
                }
                else
                {
                                lowFuelLevelInd=0;
                }
}
```

Table 22: Local declarations for low fuel level warning automata.

Table 23 contains the code of template instantiations for fuel level display system.

```
                                        // Template instantiations.
sensor = FuelLevelSensor();
kalman = FuelLevelEstimationAlgorithm();
warning = LowFuelLevelWarning();
eval_pb = EvaluateParkingBrake();
parkBrake = ParkingBrake();
refillDetect = RefillDetection();

// List one or more processes to be composed into a system.
//system sensor, kalman;                          //For simulation of sensor and filter relationship
//system eval_pb,parkBrake;                       //Enabling parking brake while simulating
system sensor, kalman, warning, eval_pb,parkBrake, refillDetect;      //For verification of LowFuelWarning
//system sensor, kalman, warning;
```

Table 23: Template instantiations.

8.1. UPPAAL – Requirements Verification

UPPAAL has a complete separate component called verifier to check safety and liveness properties. For properties verification UPPAAL explores the state space of a system model on-the-fly. Verification component of UPPAAL also supports an editor where the requirement can be specified to be verified upon the system model. UPPAAL supports a sub set of TCTL (timed computation tree logic) as query language to model the requirements to be verified upon system model.
Safety and liveness properties of fuel level display system are checked and all the requirements of fuel level display system that are mentioned in table 5 are verified on the system model during this master thesis project.

Following table 24 presents the queries that can be performed in UPPAAL verifier. Where p and q are state formulas of the form: (P1.state and x<3).

Property	Description
E<> p	There exists a path where p eventually holds.
A[] p	For all paths p always holds.
E[] p	There exists a path where p always holds.
A<> p	For all paths p will eventually hold.
p --> q	Whenever p holds q will eventually hold.
p imply q	Whenever p holds q holds.

Table 24: Queries supported by UPPAAL verifier.

Following are the requirements of fuel level display system as they are represented in UPPAAL verifier supported language and verified.

A[] not deadlock
The above query checks if there is any deadlock in the system model. And if there is no deadlock in the system model then the result returned against the verification of this property is "Property is satisfied". In case of fuel level display system model there is no deadlock in the system model hence; this property is satisfied by the verifier.

E<> (volDiffBetweenEstAndMeas != ((VOLUME_DIFF_ACCEPTED * mantleVolumeTot)/100) || (fuelLevelSensor!=(NEAR_TOP_PERCENTAGE*SC_FACTOR1)))
imply (x0 ==oldFuelVolume)
Above query verifies that the start-up state for the totalFuelLevel estimated should be the state saved from last shutdown if the stored value and fuelLevel doesn't differ with more than 10% of the total volume or if fuelLevel is above 90% of the useable tank capacity. It covers requirement AER-02 (AER_201_12) mentioned in the table 5. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that AER-02 (AER_201_12) is satisfied by the fuel level display system model in UPPAAL.

E[] (volDiffBetweenEstAndMeas == ((VOLUME_DIFF_ACCEPTED * mantleVolumeTot)/100) || (fuelLevelSensor==(NEAR_TOP_PERCENTAGE*SC_FACTOR1)))
imply (x0 == levelInMainPartRawLitre)
Above query verifies that the start-up state for the totalFuelLevel estimated should be the current fuel volume in the fuel tank (levelInMainPartRawLitre) when the stored value and fuelLevel differ with more than 10% of the total volume or fuelLevel is above 90% of the useable tank capacity. This is not explicitly mentioned in AER-02 (AER_201_12) requirement but is implemented in the fuel level display system. It covers else portion of the requirement AER-02 (AER_201_12) mentioned in the

table 5. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that AER-02 (AER_201_12) is satisfied by the fuel level display system model in UPPAAL.

E<> refillDetect.RefuelDetected

Above query verifies that the refuel can be detected if there is 30 percent increase in fuel level. It covers requirement AER-03 (AER_201_13) mentioned in the table 5. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that AER-03 (AER_201_13) is satisfied by the fuel level display system model in UPPAAL.

E<> refillDetect.stopDetected

Above query verifies that the parking brake is steadily applied for at least 5 seconds and the vehicle is considered to be properly parked. It covers requirement AER-04 (AER_201_14) mentioned in the table 5. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that AER-04 (AER_201_14) is satisfied by the fuel level display system model in UPPAAL.

E<>refillDetect.stopDetected imply refillDetect.RefuelDetected

Above query verifies both requirements AER-03 (AER_201_13) and AER-04 (AER_201_14) together at the same time. It actually verifies that refuel can only be detected if the vehicle is properly parked which means that the parking brake is steadily applied for at least 5 seconds. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that AER-03 (AER_201_13) and AER-04 (AER_201_14) are satisfied by the fuel level display system model in UPPAAL even when they are verified at the same time.

E[] refillDetect.RefuelDetected imply (fuelLevelTot/SC_FACTOR2==fuelLevelSensor)

Above query verifies that if a refill is detected the filter algorithm should not be used, the estimate should instead the value indicated by the fuel level sensor(s) until the refill is done (parking brake released). When the refill is ended the algorithm continues to calculate using the current value from fuel level sensor(s) signal as initial value. It covers the requirement AER-05 (AER_201_15) mentioned in the table 5. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that the requirement AER-05 (AER_201_15) is satisfied by the fuel level display system model in UPPAAL.

E<> fuelLevelSensor<= (10*SC_FACTOR1) imply warning.set_a ll warning.set_b

Above query verifies that there exists a warning state when fuel level is below 10%. It covers the requirement AER-06 (AER_202_2) mentioned in the table 5. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that the requirement AER-06 (AER_202_2) is satisfied by the fuel level display system model in UPPAAL.

E<> refillDetect.RefuelDetected imply not (warning.set_a ll warning.set_b)

Above query verifies that when a refuel is detected then the low fuel level warning is not set. It covers the requirement AER-07 (AER_202_3) mentioned in the table 5. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that the requirement AER-07 (AER_202_3) is satisfied by the fuel level display system model in UPPAAL.

E<> sensor.start imply (lowFuelLevelInd == 0)

Above query verifies that the initial value of low fuel level warning is set to false which means by default it is inactive. It covers the requirement AER-08 (AER_202_5) mentioned in the table 5. Result returned against this query by UPPAAL verifier is "Property is satisfied" which means that the requirement AER-08 (AER_202_5) is satisfied by the fuel level display system model in UPPAAL.

Requirement AER-01 (AER_201_11) mentioned in table 5 cannot be verified by using UPPAAL verifier. This is the actual Kalman algorithm that keeps on running during the system execution, it can only be checked by using the simulator during execution, to see if all the values in variables are correct and are updating correctly. This requirement is verified by analyzing the values during simulation.

9. Results

This section contains the results of requirements verification in both Simulink Design Verifier and UPPAAL. The tables contains the requirements ids of the requirements assigned in this thesis report, reference to the requirement ids that are present in the Scania documents, status of the requirement verification, time taken by the tools to verify the requirement and a comment field to mention the additional information. At the end of the tables, status of general design error detection offered by the tools is also presented.

9.1. Requirements Verification Results of Simulink Design Verifier

Requirement ID	Requirement Reference	Status	Time	Comment
AER-01	AER_201_11	Cannot be Verified		Proof cannot be completed within the available resources.
AER-02	AER_201_12	Verified	02 Seconds	Requirement specification is not consistent with the simulink model.
AER-03	AER_201_13	Verified	238 Seconds	Fuel level greater than 30%. Due to limited resources this requirement cannot be verified for complete range of fuel level sensor values (1-100) at once. Proved for one value of fuel level sensor at a time e.g. 31, 49, 70, 90 or 100.
AER-04	AER_201_14			
AER-05	AER_201_15	Verified	24 Seconds	Fuel level greater than 30%. Due to limited resources this requirement cannot be verified for complete range of fuel level sensor values (1-100) at once. Proved for one value of fuel level sensor at a time e.g. 31, 49, 70, 90 or 100.
AER-06	AER_202_2	Verified	03 Seconds	
AER-07	AER_202_3	Verified	25 Seconds	
AER-08	AER_202_5	Verified	01 Seconds	
Integer overflow		Verified	43 Seconds	No integer overflow errors detected.
Division by zero		Verified		No divide by zero errors detected.
Check specified intermediate minimum and maximum values		Verified		No errors are detected against specified intermediate minimum and maximum values.

Table 25: Requirements Verification Results of Simulink Design Verifier.

9.2. Requirements Verification Results of UPPAAL

Requirement ID	Requirement Reference	Status	Time
AER-01	AER_201_11	Cannot be Verified. It is not possible to specify these types of requirements in UPPAAL verifier for verification. Variables values of Kalman algorithm are validated in UPPAAL simulator.	
AER-02	AER_201_12	Verified	.016 Seconds
AER-03	AER_201_13	Verified	.016 Seconds
AER-04	AER_201_14	Verified	.016 Seconds
AER-05	AER_201_15	Verified	.016 Seconds
AER-06	AER_202_2	Verified	.016 Seconds
AER-07	AER_202_3	Verified	.016 Seconds
AER-08	AER_202_5	Verified	.016 Seconds
Dead lock detection		Verified	120 Seconds

Table 26: Requirements Verification Results of UPPAAL.

10. Analysis of Model checking tools

In this section the model checking tools that are used in this master thesis project are analyzed based upon the experiences that are acquired while performing model checking for fuel level display system of Scania. This analysis is based upon the features supported by the tools for model checking.

10.1. Simulink Design Verifier

Simulink Design Verifier is a model checking tool that identifies design errors in the system model, generate test vectors for model checking, and verify system designs against specified requirements. Simulink Design Verifier software makes use of formal techniques to find the design errors in system model and it does not require extensive tests or simulation runs. Design errors that can be detected by Simulink Design Verifier in the system model include dead logic detection, integer overflow identification, division by zero, and violations of design properties and assertions [8].

Polyspace and Prover Plug-In are the formal analysis engines that are used behind Simulink Design Verifier software as model checkers. In Simulink Design Verifier software blocks and functions are used for modeling functional requirements and safety requirements that are later verified on the system model. If the requirements are satisfied by the system model then the software returns the result true against the requirements verification and if the requirements are not satisfied then counter example is generated, that can be used to identify why the requirements are not satisfied [8].

Simulink Design Verifier also supports the functionality of generating test vector from functional requirements. It also provides the feature that ensures the model coverage by using model coverage objectives which include condition, decision, and modified condition/decision (MCDC) [8].

Simulink Design Verifier software provides model support for fixed-point models and it also provides support for floating-point models, which makes it more useful for the systems that require handling of floating point values. Support for floating point really makes a difference as compared to UPPAAL because UPPAAL does not support the floating point models [8].

Simulink Design Verifier software supports many features of Simulink and Stateflow but not all of them. There are some features and blocks of Simulink, and few features of Stateflow that are not supported by Simulink Design Verifier software. If any one of the unsupported features or blocks is used in system model that is to be verified by Simulink Design Verifier then the model is not verified and error is generated at the initial stage that the model is not compatible, because the compatibility of the system model is automatically checked by the Simulink Design Verifier before starting any analysis on it. Compatibility of model can also be checked independently before starting any analysis [9].

Result of compatibility check can be; model is compatible, model is incompatible or model is partially compatible. In case if the model is compatible, this is the ideal case and analysis is started on the model without any delay. In case if the model is incompatible, analysis is not started and log is generated in order to trace, identify and fix the compatibility issues. Partial compatibility notification is generated if at least one object in model is incompatible. Analysis can be continued with partial compatibility but in such a situation the analysis results might be incomplete or not exactly the same as expected [9].

Simulink Design Verifier software provides a functionality called block replacement, which can be used to define rules to automatically perform block replacement in system model, if there are some unsupported blocks used in the model. Unsupported blocks are replaced by supported blocks, which

have same functionality, on the basis of these block replacement rules. In block replacement Simulink Design Verifier makes a separate copy of model and replaces the unsupported blocks with supported blocks in the copy without changing the original model. However, in the case study for this master thesis project block replacement mechanism is not used, because the blocks and features that are used during the model construction of fuel level display system in Simulink are all those that are supported by Simulink Design Verifier software [9].

10.2. UPPAAL

UPPAAL is a model checking tool for modeling, simulation, validation and verification of real-time systems that are modeled as networks of timed automata. UPPAAL uses computation tree logic (CTL) that is branching time logic for modeling the requirements to be verified on system model.

In UPPAAL behavior of the system under consideration is modeled as concurrent timed automata. Templates in UPPAAL represent generic behavior of system entities, and when the templates are instantiated then they represent precise behavior of entities in system model.

Different basic types of variables are supported in UPPAAL. Four predefined types for variables in UPPAAL are int, bool, clock, and chan. Other types like arrays and records can be defined over these existing types.

Events in UPPAAL can be observed as edges that are enabled in any instance of a template. Timers associated with them act as event sources, and channels can activate an edge which appear as an event. Edges in UPPAAL are referred as transitions in other model checking tools. The concept of an event is implicit in UPPAAL and is coupled with the definition of an edge.

State of the system model in UPPAAL can be seen as the state of all the automaton in system, constraints on clocks, and values of variables. States in UPPAAL can be anyone of three kinds 1) initial, 2) urgent or 3) committed. No time passes when there is a transition through a committed or urgent state, therefore they act as instantaneous intermediaries among states.

It is a obligation for all template in UPPAAL model to have an initial state with initial attributes. Final states are not compulsory but their absence can lead to a deadlock situation.

In UPPAAL edges contain select, guard, synchronization, and update options that can be specified for each edge. An automaton can initiate an edge that is individually synchronized with some other automaton in system model. Transition is permitted in UPPAAL only if a condition is satisfied and synchronization channel can be used to stimulate this by a message. Channels in UPPAAL provide a 1-to-1 communication relationship between two edges, but there also exist channels that are called broadcast channels that provide one-to-many relationship among edges. Updates in UPPAAL are responsible for making the change in variables values as a consequence of transition.

With the use of channels, processes can be synchronized that permits simultaneous operating instantiations of automaton. One-to-many synchronization can be achieved on broadcast channels. When two processes are synchronized then both edges are activated at once. States in UPPAAL do not maintain any history. If it is required by the states to maintain history it can be achieved by utilizing variables or flags.

In UPPAAL there is no support for floating point values, which makes modeling of the systems that require handling of fractional point values very tricky. Mathematical tricks are required to model such a system in UPPAAL, even after doing that, it cannot be guaranteed that the system will produce

100 percent the identical behavior as expected. Automatons in UPPAAL are timed and time has explicit meaning and is very critical.

There is no concept of composition of states in UPPAAL. States cannot be composed within other states. Templates and functions in UPPAAL are parameterized that can be declared to be either call by value or call by reference.

10.3. Tools Comparison

Following table 27 presents some major points of comparison of Simulink Design Verifier and UPPAAL.

Simulink Design Verifier	UPPAAL
Simulink blocks, state flows and Matlab functions are used for system modeling.	Timed automata, states and transitions are used for system modeling.
Supports floating point models.	No support for floating point models.
Simulink blocks, state flows and Matlab functions are used for requirements specification.	Subset of TCTL (Timed computational tree logic) is used for requirements specification.
Code can be generated from the simulink model directly.	In current version of UPPAAL no support for code generation from UPPAAL model.
Require more time for property proving	Property verification is very efficient
Limited block support in Simulink Design Verifier against blocks offered by Simulink for system modeling.	All features are offered by UPPAAL designer are supported in verifier.

Table 27: Tools comparison.

11. Relevance of Model checking with respect to ISO26262

ISO (International Organization for Standardization) is a worldwide organization working in preparing international standards.

ISO 26262 is a specific standard of ISO about Road Vehicles – Functional Safety. ISO 26262 has 9 sub parts of which Part 6 (Product Development at the Software Level) and Part 8 (Supporting Processes) mainly talks about the use of verification in product development. Part 6 (Product Development at the Software Level) highlights the different methods for verification and ASIL's associated to them and whether the method is recommended to be used in the systems of that specific ASIL or not [5].
Part 8 (Supporting Processes) generally describes the verification activity in general in product development. Main purpose of verification is to guarantee that the system under consideration satisfy all the requirements [6].

Automotive Safety Integrity Level (ASIL)
According to ISO 26262, the risk of each hazardous event is evaluated based on 3 factors

- Severity (impact of possible damage or injury)
- Exposure (frequency of the situation)
- Controllability (avoidance of damage through timely reactions of the persons involved)

Table 28, 29 and 30 are used to determine the values for severity, exposure and controllability.

	Class			
	S0	**S1**	**S2**	**S3**
Description	No Injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries

Table 28: Classes of severity [4].

	Class				
	E0	**E1**	**E2**	**E3**	**E4**
Description	Incredible	Very low probability	Low probability	Medium probability	High probability

Table 29: Classes of probability of exposure regarding operational situations [4].

	Class			
	C0	**C1**	**C2**	**C3**
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable

Table 30: Classes of controllability [4].

Table 31 is used to determine the value for ASIL based upon the values of severity, exposure and controllability.

Severity class	Probability class	Controllability class		
		C1	**C2**	**C3**
S1	**E1**	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	**E1**	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	**E1**	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Table 31: ASIL determination [4].

[5] Depending on the values of severity, exposure and controllability, the appropriate safety integrity level for a given function or hazard is calculated by using the ASIL determination table. ASIL has 4 levels (A, B, C, D) with D representing the most critical and A the least critical level. QM (quality management) in ASIL determination table represents that no requirement to comply with ISO 26262 [4]. Four levels of ASIL can have one of three values (++, +, o).

- "++" represents that the method is highly recommended for the specified ASIL.
- "+" represents that the method is recommended for the specified ASIL.
- "o" represents that the method has no recommendation for or against its usage for the specified ASIL.

[4] ISO 26262 says that the software architectural design shall be verified according to [6] clause 9, and by using the software architectural design verification methods listed in the following table 32 to demonstrate the following properties:

- Verify the compliance of software architectural design with the software safety requirements.
- Check the compatibility of software architecture design with the target hardware.
- Verify that all the design guidelines are properly followed.

Methods of Verification	ASIL			
	A	**B**	**C**	**D**
Walk-through of design	++	+	o	o
Inspection of design	+	++	++	++
Simulation of dynamic parts of the design	+	+	+	++
Prototype generation	o	o	+	++
Formal Verification	o	o	+	+
Control flow analysis	+	+	++	++
Data flow analysis	+	+	++	++

Table 32: Software architecture design verification methods.

According to ISO 26262 standards formal verification has no significance for the systems of either ASIL A or ASIL B. Formal verification is only recommended for the systems of either ASIL C or ASIL D.

Scania has not yet formally assigned the ASIL to their systems but it will be done in the near future because research work is being carried out to check the functional safety of their systems. Therefore in case of fuel level display system that is used as a case study in this thesis project if the ASIL assigned to this system is either ASIL C or ASIL D only then model checking is relevant for this system and if the ASIL of this system is either ASIL A or ASIL B then there is no need for performing model checking activity for this particular vehicle control system according to ISO 26262 standard.

Verification can be divided into two sub categories 1) Formal Verification 2) Simple Verification. Formal verification is the process of proving or disproving system properties using formal methods (i.e., mathematically precise, algorithmic methods). A formal proof of a property guarantees that no simulation of the specified system will violate the property which hence eliminates the need for writing additional test cases to check the property. Propositional logic, symbolic simulation, model checking, theorem proving and floating point verification are different methods for formal verification. Methods for simple verification are review, walk-through, inspection, simulation, engineering analyses, demonstration, and testing. Model checking is one of the methods for formal verification. Model Checking is very good alternative to simulation and testing for validation and verification of systems. Given a system model and properties that the system model is supposed to satisfy, the model checking tools explore the complete state space of system model to see if the mentioned system properties are satisfied by the system model or not. Either the given properties are verified by the model checking tool or the model checker generates counter examples.

12. Conclusion

In this master thesis project initially the academic and industrial world are explored to find model checking tools that can be used for model checking. After performing the systematic literature review for existing model checking tools in academia and industry, two model checking tools 1) Simulink Design Verifier and 2) UPPAAL are finalized to be used during this master thesis project. After that one of Scania's vehicle control systems called fuel level display system is selected for performing the activity of model checking by using the selected model checking tools. All the requirements that the model of fuel level display system is supposed to satisfy are listed. Fuel level display system is considered to be the well-documented, non-trivial best-practice example at Scania. Selection of this application is based upon the documentation, test data, requirements and state machines available. Moreover this particular system has already been used in many other research oriented tasks and other master thesis as well.

Model of Scania's Fuel Level Display System is created in Simulink Design Verifier and UPPAAL. Requirements of the fuel level display system are verified on the designed system model. All the functional requirements are verified on the system model. There is one requirement that could not be model checked. The rest of requirements are all satisfied. Moreover system models are also verified against general design errors like integer and fixed-point overflows, division by zero, dead logic detection, deadlock detection, violations of design properties and assertions violations.

Model checking tools Simulink Design Verifier and UPPAAL are compared with each other depending upon their advantages and disadvantages. There are certain advantages and disadvantages associated with each tool for performing model checking in any of them. Eventually the choice of model checking tool is dependent upon the system to be verifier and it's characteristics for which model checking is being performed.

In future if Scania wants to perform model checking for their other vehicle control systems, Simulink Design Verifier is a feasible option. Scania already has the model of the vehicle control system available in Simulink so if Simulink Design Verifier is used as a model checking tool, then it will require less time for model checking, because only the properties are to be modeled and systems models are already available. Existing system models of vehicle control systems in Simulink can be used for model checking with minor modifications. On the other hand in any other tool models of the systems will have to be created first.

After analyzing the relevance of model checking according to ISO 26262 standards it is concluded that model checking has no significance for the systems of either ASIL A or ASIL B. Model checking is recommended by ISO 26262 for the systems of either ASIL C or ASIL D.

Model checking tools are provided with the system model and the properties of the system that the model is supposed to satisfy. Model checking tools verify whether the properties are satisfied by the system model or not. If the properties are satisfied then the system model is verified against the specified requirements and if the properties are not satisfied then model checking tools generate counter examples that can help identifying why the properties are not satisfied and where are the faults in system model.

Model checking should be used to complement the testing process instead of replacing the testing process with it. Model checking can help ensure the system to be functionally feasible for the users and environment. Modeling checking can help identifying the design faults in the system model that cannot be detected with other testing techniques. With model checking design faults are detected at the early stages of system development, where they are easier to fix and require less time, less effort and less resources for fixing the design errors, instead of encountering the errors during the final

stages of system development, where it is very hard to fix them and require lot of time, effort and resources. Since the software applications in systems are becoming more and more important, so the model based development is gaining more importance and model checking techniques are the best solution to ensure the quality of the systems and to guarantee that there are no design errors in the system model and the system model satisfies all the specified requirements

13. Future work

After working on this master thesis future work can be suggested in different areas; one from the point of view of Scania, the other in general in the field of model checking and improvements in the tools that are used in this master thesis project for model checking.

During this master thesis project model checking of fuel level display system of Scania is performed. There are a lot of other systems at Scania for which the model checking is needed to be done in order to have the models of all the systems checked formally against the design errors and specified functional requirements. Other systems at Scania for which model checking activity can be performed are Gearbox Management System, Articulation Control Systems, Engine Management System, Brake Management System, Suspension Management System, Locking and Alarm System, All Wheel Drive System, Instrument Cluster System, Tachograph System, Visibility System, Air Processing System, Body Work System, Bus Chassis System, Audio System, Crash Safety System, Automatic Climate Control, Auxiliary Heater System Water to Air, Auxiliary Heater System Air to Air, Clock and Timer System, Road Transport Informatics Gateway, and Road Transport Informatics System etc. These systems are relatively more safety critical as compared to the fuel level display system. According to ISO 26262 standard model checking is more relevant for systems that are more safety critical.

Generally in the domain of model checking as future work it is recommend to write automated transformation rules for automatically transforming the model from Simulink Design Verifier to UPPAAL and vice versa. Later on it is suggested to implement some plug-in or small tool based upon these rules that will be able to transform the model automatically created in Simulink Design Verifier to UPPAAL model and vice versa.

There are some limitations of Simulink Design Verifier that can be improved to make it a better model checker. Simulink Design Verifier software can be extended to support the features and blocks of simulink that are not yet supported by Simulink Design Verifier software. Also in UPPAAL there are lots of places for improvement. UPPAAL can be extended to support fractional values; new data types can be introduced to support fractional values in UPPAAL. UPPAAL can also be extended to generate deployable code automatically from the UPPAAL model. Feature of plotting the values on a graph will also be a nice addition in UPPAAL that can help better understand and analyze the results.

References

1. Christel Baier and Joost-Pieter Katoen (2008), Principles of model checking. The MIT Press. Cambridge, Massachusetts. London, England. ISBN 978-0-262-02649-9. Chapter 1. Page 1-16.
2. http://anna.fi.muni.cz/yahoda/ [Accessed: 17 March, 2012].
3. http://www.scania.com/scania-group/scania-in-brief/index.aspx [Accessed: 27 June, 2012].
4. International Standard – ISO26262 Road Vehicles Functional Safety - Part 3 (Concept Phase) first edition 15 November, 2011.
5. International Standard – ISO26262 Road Vehicles Functional Safety - Part 6 (Product Development at the Software Level) first edition 15 November, 2011.
6. International Standard – ISO26262 Road Vehicles Functional Safety - Part 8 (Supporting Processes) first edition 15 November, 2011.
7. Kim G. Larsen, Paul Pettersson, and Wang Yi (1997). Uppaal in a Nutshell. International Journal on Software Tools for Technology Transfer.
8. Mathworks, Simulink Design Verifier user guide. Version Matlab R2012a. Page 21-23.
9. Mathworks, Simulink Design Verifier user guide. Version Matlab R2012a. Page 81-124.
10. Mathworks, Simulink Design Verifier user guide. Version Matlab R2012a. Page 145-188.
11. Mathworks, Simulink Design Verifier user guide. Version Matlab R2012a. Page 277-314.
12. Mathworks, Simulink Design Verifier user guide. Version Matlab R2012a. Page 419-432.
13. Pressman, Roger (2009). Software Engineering: A Practitioner's Approach, 7th Edition. McGraw-Hill.
14. Scania Technical Product Data 1949329, Allocation Element Requirement – AER Fuel Level Estimation: AE201.
15. Scania Technical Product Data 1949330, Allocation Element Requirement – AER Low Fuel Level Warning: AE202.
16. Sumit Nain, Moshe Vardi (2007). Branching vs. Linear Time: Semantical Perspective. Automated Technology for Verification and Analysis, pp. 19-34
17. Uppaal 4.0: Small Tutorial. 16 November 2009.

Acronyms and abbreviations

Acronym or Abbreviation	Definitions
MDH	Mälardalen University
GSEEM	Global Software Engineering European Master
IEC 61508	International Electrotechnical Commission
TÜV SÜD	Technischer Überwachungs – Verein (Technical Examination/Monitoring Association)
EN 50128	Software for railway control and protection systems
AE	Allocation Element
AER	Allocation Element Requirement
LTL	Linear Temporal Logic
TCTL	Timed Computation Tree Logic
GUI	Graphical User Interface
MCDC	Modified Condition Decision Coverage
COO	Coordinator
CAN	Controller Area Network
ECU	Electronic Control Units
EMS	Engine Management System
ICL	Instrument Cluster
ISO	International Organization for Standardization
ASIL	Automotive Safety Integrity Level

Table 33: Acronyms and abbreviations.