2-1-2018

Formal verification of automotive embedded UML designs

Ghada Bahig

Formal Verification of Automotive Embedded UML Designs

By

Ghada Moussa Bahig

Department of Computer Science

The American University in Cairo

Thesis Dissertation

In Partial Fulfillment of the Requirements of Doctor of Philosophy in Applied Sciences

Thesis Advisor: Dr. Amr El-Kadi

July 2017

Dissertation written by

Ghada Moussa Bahig

M.S., American University in Cairo, Egypt, 2000

B.S., American University in Cairo, Egypt, 1997

Approved by

_____ , Chair, Doctoral Dissertation Committee

_____ , Members, Doctoral Dissertation Committee

_____

_____

_____

Accepted by

_____ , Chair, Department of Computer Science

_____ , Dean, College of Sciences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my advisor, Professor Amr El-Kadi who has the attitude and the substance of a genius: he continually and convincingly conveyed a spirit of adventure about research and scholarship, and an excitement about teaching. Without his guidance and persistent help and support, this dissertation would not have been possible.

I would like to thank my committee members, Professor Ashraf Salem, Professor Sherif Hammad, Professor Mohamed Shalan and Professor Sherif Gamal Aly who supported and guided me to defend my dissertation. Their constructive comments opened my eyes to challenges and paved a way for me to work more on a research area that I adore. Dr. Sherif Hammad, you have helped shape who I am today. Dr. Ashraf, you were my driving force at Mentor to focus and finish my PhD. Thanks for your guidance and support.

I would also like to thank my parents who gave me confidence and support. They always encouraged me to finish and wanted to help in any way they can so that I can finish my dissertation. I would not be who I am if it were not for their support. I have learned and continue to learn so many from them and I love them to death.

I would also like to thank my lovely daughter Nada who was trying to encourage me via challenging my directions to her and comparing the completion of this work to how she will complete stages in her life. I was not happy when she did that but it was a way to enforce my persistence to finish this so that I can be a role model for her.

I would also like to thank my husband who always emphasized that I need to finish my PhD.

Finally, I would like to thank God almighty for answering my prayers to keep going and not to give up.

.

ABSTRACT

American University in Cairo

Formal Verification of Automotive Embedded UML Designs

By

Ghada Moussa Bahig

Thesis Advisor: Dr. Amr El-Kadi

Software applications are increasingly dominating safety critical domains. Safety critical domains are domains where the failure of any application could impact human lives. Software application safety has been overlooked for quite some time but more focus and attention is currently directed to this area due to the exponential growth of software embedded applications. Software systems have continuously faced challenges in managing complexity associated with functional growth, flexibility of systems so that they can be easily modified, scalability of solutions across several product lines, quality and reliability of systems, and finally the ability to detect defects early in design phases. AUTOSAR was established to develop open standards to address these challenges. ISO-26262, automotive functional safety standard, aims to ensure functional safety of automotive systems by providing requirements and processes to govern software lifecycle to ensure safety. Each functional system needs to be classified in terms of safety goals, risks and Automotive Safety Integrity Level (ASIL: A, B, C and D) with ASIL D denoting the most stringent safety level. As risk of the system increases, ASIL level increases and the standard mandates more stringent methods to ensure safety. ISO-26262 mandates that ASILs C and D classified systems utilize walkthrough, semi-formal verification, inspection, control flow analysis, data flow analysis, static code analysis and semantic code analysis techniques to verify software unit design and implementation. Ensuring software specification compliance via formal methods has remained an academic endeavor for quite some time. Several factors discourage formal methods

adoption in the industry. One major factor is the complexity of using formal methods. Software specification compliance in automotive remains in the bulk heavily dependent on traceability matrix, human based reviews, and testing activities conducted on either actual production software level or simulation level. ISO26262 automotive safety standard recommends, although not strongly, using formal notations in automotive systems that exhibit high risk in case of failure yet the industry still heavily relies on semi-formal notations such as UML. The use of semi-formal notations makes specification compliance still heavily dependent on manual processes and testing efforts. In this research, we propose a framework where UML finite state machines are compiled into formal notations, specification requirements are mapped into formal model theorems and SAT/SMT solvers are utilized to validate implementation compliance to specification. The framework will allow semi-formal verification of AUTOSAR UML designs via an automated formal framework backbone. This semi-formal verification framework will allow automotive software to comply with ISO-26262 ASIL C and D unit design and implementation formal verification guideline. Semi-formal UML finite state machines are automatically compiled into formal notations based on Symbolic Analysis Laboratory formal notation. Requirements are captured in the UML design and compiled automatically into theorems. Model Checkers are run against the compiled formal model and theorems to detect counterexamples that violate the requirements in the UML model. Semi-formal verification of the design allows us to uncover issues that were previously detected in testing and production stages. The methodology is applied on several automotive systems to show how the framework automates the verification of UML based designs, the de-facto standard for automotive systems design, based on an implicit formal methodology while hiding the cons that discouraged the industry from using it. Additionally, the framework automates ISO-26262 system design verification guideline which would otherwise be verified via human error prone approaches.

Chapter 1. Introduction

Software plays a major role in almost all industries nowadays from cooking in our kitchens, to driving our cars, to working in our offices. Some of these systems are safety critical which means that failure of the software could cause hazardous consequences on human life. Safety Critical Computing (SCC) aims to optimize system safety in the design, development, use, and maintenance of software systems and their integrations with safety critical hardware systems in an operational environment. In fact, one of the very first seen ambiguities of SCC is the way it is viewed across industries and regulatory bodies. Some ongoing research efforts address safety based on measuring how well the system does exactly what it is intended to do while others view safety as designing a system that is able to handle cases when a system does not work as expected. In the later context, safety engineers assume that any system will fail and then they work through the consequences to ensure that they are well handled through inductive and deductive techniques.

Attention to safety software engineering started when failures in embedded critical systems led to critical failures. In March 2008, a Medtronic heart pacer device was reported to be vulnerable to remote attacks [1]. In 2003, an electrical blackout took place in North America for hours and it was reported that key phase 1 events started with a software system failure [2]. In the 1980s, a bug in the code controlling a radiation therapy machine was found to be the reason why at least 5 patients died due to administering incorrect volume of the radiation during treatment sessions [3]. A good number of such failures are also attributed to incompliance to specification, a glitch in an automaker‘s software design and testing approach in airbags design resulted in the recall of 47,401 vehicles in the US and a further 3,099 in Canada and Mexico [4]. Other reported incidents took place in space exploration, medical, electric power transmission, financial, telecommunications, military, media, and automotive domains.

In automotive systems, it is crucial to ensure design correctness from compliance to specification perspective as early as possible. Safety standards put strict processes that involve manual reviews and requirements traceability in all software life cycle to ensure specification compliance. Industry still heavily relies on manual reviews and processes which is impractical since specification is still captured in informal and semi-formal notations which open the door for requirement specification ambiguity. In recent years, software costs increased exponentially due to the increasing number of software enabled features in a car. A modern car can contain up to 90 Electronic Control Units (ECUS), 11 networks and might host one million lines of code (LOC) [85]. This increases software complexity and with it, the probability of failures. The task of verifying software to detect failures is becoming more and more difficult, time consuming and critical. A good number of failures are attributed to incompliance to specification.

1.1 Existing Approaches

Existing approaches that target software/system safety include:

1. Dependency on standards and processes enforced by regulatory committees to ensure software safety. Regulatory agencies such as ISO publish software safety standards. ISO-26262 is the automotive standard that is based on IEC 61508. This is the functional safety standard across electrical and electronic E/E systems. Several regulatory entities, such as German law, hold car producers liable for damage to a person because of malfunction of a product. If it was not possible for the malfunction to be detected via the current technical state, the liability is omitted. In this context and within the automotive software and hardware domains, ISO 26262 is considered the technical state of the art. Standards rely on a system of steps to govern and manage functional safety and govern product development on a system, hardware and software level.
2. Code level approaches, such as, static analysis and unit testing coverage. Static analysis is a way of examining a code without executing it. The process depends on analyzing code structure and ensuring the code adheres to industry coding standards such as MISRA-C. Unit testing is also done on the code level. Several

metrics are generated to ensure that unit testing addresses potential issues in the code. Some utilized metrics are: code coverage, cyclomatic complexity and maintainability index.

3. Extensive testing at different levels including white box testing, black box testing, system and integration testing based on a variety of algorithms, such as, random test generation, path oriented, goal oriented, and expert based adhoc test designs. Testing is iterative, incremental and includes several stages beginning with module test, simulation testing, hardware in the loop testing and finally integration testing when all system components (Hardware and Software) are ready.
4. Model driven approaches which rely on modeling a system abstraction and being able to simulate these abstractions manually or automatically based on designing test cases and finally formal methods but on a very small scale [5][6][7]. Model-driven approach in software engineering is gaining wide ground in both industry and academia. Legacy approaches still focus on implementation unlike model driven approaches, which depends on models in all levels of the software development process. The outcome of this shift has triggered quite a big change in the approach to software development in design, implementation and testing stages. Model based development utilizing the Unified Modeling Language (UML) has driven many researchers to use UML diagrams like state machine diagrams, use-case diagrams, sequence diagrams, etc. to generate test cases and even code. Model-based testing approaches come with a big edge which is increasing productivity as well as quality by changing the focus away from testing to much earlier stage of the software development process. Additionally, generating test cases are becoming more independent of any implementation of the design
5. Formal methods which had traditionally not been widely adopted due to several barriers. To name some, entry cost is high (education, legacy methods migration), problem space scalability shortcomings, and insufficient tool support for formal

methods since most of the existing tools originated from academia as opposed to industrial endeavors and finally lack of expertise/training to formal methods. In truth, it is very hard and unrealistic to assume that an ABS (Anti brake locking system) application engineer will be able to define safety attributes in formal notation to ensure that the system function is safe from a design perspective.

An automotive functional safety standard, ISO-26262 [13], has been published in 2011 to ensure software functional safety. ISO-26262 is a functional safety standard that declares its objectives as: providing an entire automotive safety lifecycle from management, development, production, operation, service, and decommissioning of the product and supports adapting the needed activities during the different lifecycle phases depending on an automotive specific risk-based approach for determining risk classes (Automotive Safety Integrity Levels, ASILs). The standard highly recommends capturing the design in semi-formal notation and also highly recommends the use of semi-formal verification methods to ensure design correctness in ASILs C and D. The use of formal method is only recommended for ASIL D software. In this research, we will present a framework that allows software designers to formally verify a specified software in a semi-formal notation (UML). This complies with ISO-26262 design verification guidelines for ASILs C and D which highly recommends semi-formal verification of the design for ASILs C and D. Several automotive modules were used as case studies. An industrial ASIL B compliant implementation and reported testing/production level defects is used to conduct a comparative analysis and evaluation of the proposed framework. The production level and late testing defects in the industrial use-case can be discovered via our framework at the design stage. The aim of this research is to show that defects identified on the code level during testing and release stages could be identified on the design level via our proposed framework.

Our intent in this research is to address software verification in the early stage of the software lifecycle, namely, the design stage. The research was motivated by the steep

growth of critical software functions in embedded systems, the fact that 50% of defects are introduced by the design stage, cost of finding a defect during testing is much higher than finding it during design, late defects are mostly due to specification incompliance defects, and the birth of AUTOSAR Automotive standard and ISO-26262.

We will present how existing V&V techniques still heavily depend on testing and little effort focuses on pushing the verification to the design stage. Our research aims to address the motivations while addressing the current shortcoming that have discouraged the industry from using formal methods in the design stage of automotive software development. Formal methods have not been widely adopted due to complexity of notations, lack of support and lack of support tools. Automotive suppliers are also looking for non-disruptive techniques that integrate with their used models and design environments so that they do not have to re-invent the wheel for their software development lifecycle.

1.2 Dissertation Organization

The dissertation is organized as follows: chapter 2 explains why the problem is hard or why a solution is needed, chapter 3 discusses state of the art, chapter 4 defines all basic blocks of the framework followed by a description of the framework flow, chapter 5 introduces the case study modules, chapter 6 details the case study results and the comparative analysis with industrial flow for an automotive module and chapter 7 summarizes the conclusion and future work.

Chapter 2. Research Motivation

The surge of electronic systems has led to major ramifications in vehicle engineering. Today's vehicle can have up to 4 kilometers of wiring in comparison to 45 meters in manufactured vehicles in the 50s. Apollo 11 utilized nearly 150 Kbytes of onboard memory in the late 60s to go to the moon and back. Nowadays, a moderate family car can use up to 500 Kbytes in infotainment computer in order to keep the CD player from skipping tracks.

The industry change had its toll on power demands as well as design, which led to major innovative changes in electronic networks for automobiles. Researchers have shifted focus to try to ensure that developed systems are safe, efficient and reliable and could replace entire mechanical and hydraulic applications. Control networks connect electronic equipment in a car just as LANs connect computers. The networks allow communication between the different computers in the vehicle to transfer and share data. The vehicle is now a LAN of connected computers that need to talk to each other to make smart and critical decisions. Traditionally, networks connectivity depended on wiring. However, currently, due to the surge in communication within the vehicle, the use of wiring hit a technological wall. Several protocols are now the backbone of existing control and communications networks to accommodate the wall of using discrete wiring. Centralized followed by distributed networks have replaced point-to-point wiring. Figure 1 shows an example of the electronic surge, which triggered the number of systems and applications contained in a modern car network architecture to increase drastically.

Nowadays, car electronics represent more than 30% of the total cost of a car [87]. In a 2008 BMW 5 series, it is estimated that there are up to 80 electronic modules communicating together that is made up of nearly 10 million lines of software code. As car electronic architectures become more and more complex, carmakers outsource the design of electronic modules to automotive electronic suppliers. The design of an

automotive electronic module (hardware, software and mechanical skills) typically consumes 24 months of development and involves tens of team members, both technical and managerial. The attributed software defects of such a project is more than 80% of the total number of defects although software testing takes up to 50% of the time spent on project management and technical activities.

In automotive industry, the engineering processes of the software development life cycle are performed according to the standard *V-model*. The main engineering processes are: Requirements specification and management, global design, component development, integration, and validation. These processes are carried out before each carmaker delivery of the software product. In fact, an incremental-type design process is initiated between the carmakers and their suppliers in order to take the carmaker constraints and requirements priorities into account.

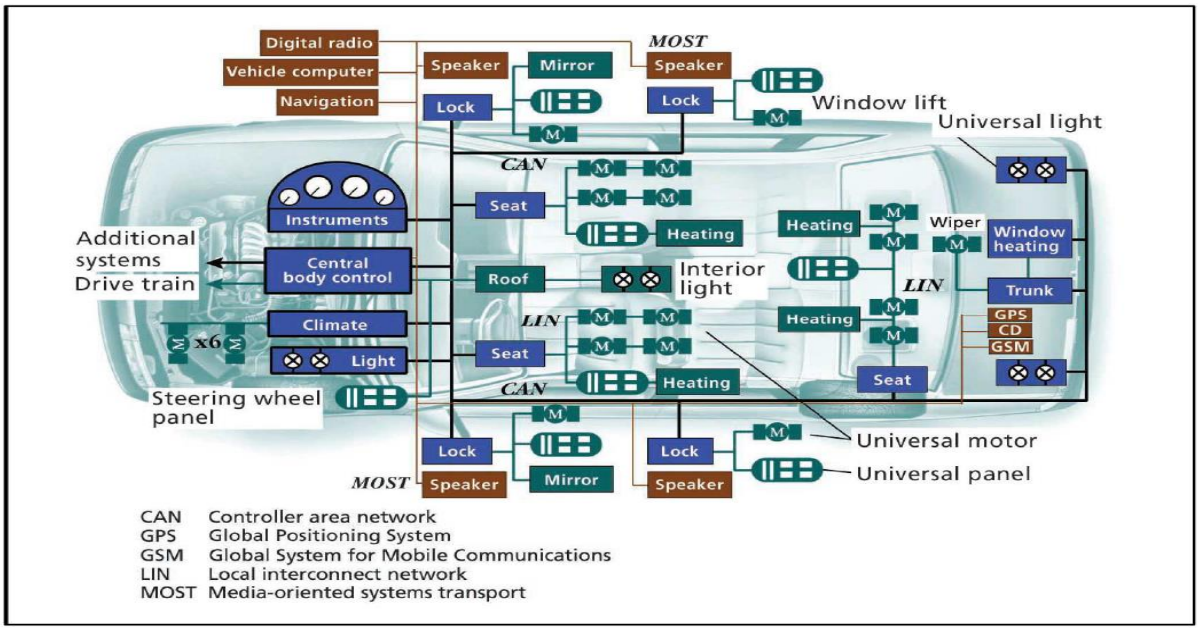

Figure 1 Modern Car systems and Networks

The number of iterations is defined based on the project's complexity and adjusted according to the carmaker inputs and project constraints. Considering a fairly complex

project, the number of iterations between supplier and manufacturer can reach ten. Each iteration (delivery) follows *Verification and Validation* (*V&V*) activities imposed by the supplier and ends with a substantial number of software defects. This number depends on the size (in terms of lines of codes), complexity and maturity of the delivered software.
In automotive industry, both static and dynamic software V&V techniques [88] are practiced in order to ensure that the resulting software product implementation is compliant to the specification and customer needs. Although static techniques are necessary to detect defects earlier in the development process, testing techniques are considered the ultimate techniques in the detection of software bugs. Testing represent up to 90% of the time spent in V&V of an automotive software product. Many automotive industries have invested on automating test execution; however, the test design activity is still manual and completely based on the test engineers' experience.

As the software products become more and more complex, it is impossible to be able to check that the software product responds correctly to all possible test input data. In [89,90], the authors demonstrate that software testing is a NP-Complete (complex) problem and therefore impossible to achieve full coverage of test input data on any software. Moreover, each engineer could have a different perception of the possible and critical test input data based on experience. In automotive industry, a software product is always tested against predefined objectives such as code and specification coverage. Meanwhile, for time and budget reasons, managers could decide to stop testing a software product even if the target coverage rate is not reached due to project timing constraints.

Facing this growing software complexity, carmakers and automotive electronic suppliers are looking for efficient methods to verify and validate software. As the automotive market becomes more competitive, development time reduction and early software defects detection become major drives in the domain. Figure 2 [92] shows how the cost per fault multiplies by 5 in functional testing stage, 10 times in system testing and 50 times in production. The study also shows how the design organization introduces 40% of

the defects introduced in the software and that currently fault discovery during design is only limited to 6%. Additionally, defects introduced in the design stage leaks to the development, testing, UAT, and Production stages. Any introduced solution needs to ensure that defects leakage/slippage from Design to Development stage is minimal and this should be a metric to evaluate design verification approach.

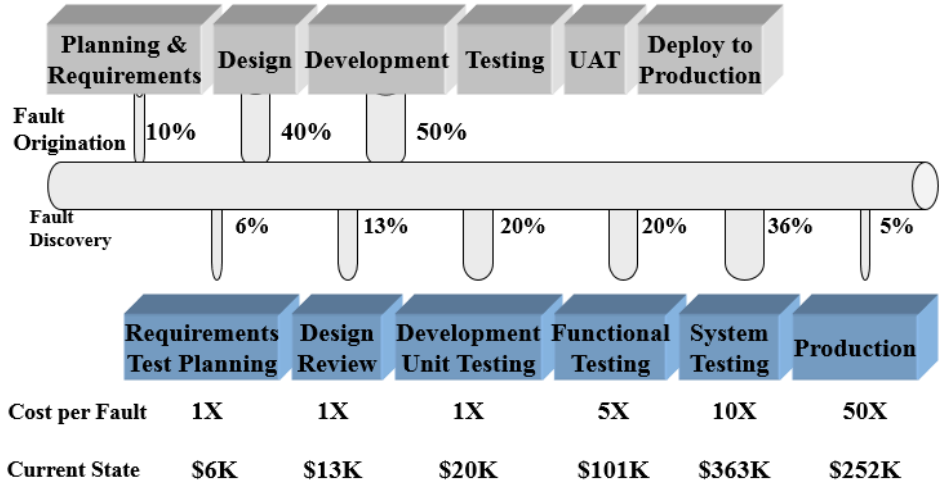

Figure 2 Cost Of Late Testing

2.1 Automotive - Fueling Change Factors

The birth of AUTOSAR and ISO-26262 automotive functional safety standard is a strong proof of how automotive suppliers are committed to enhancing their software to meet these challenges and it is one of the motivations behind the work presented in this dissertation as detailed in next sections. In [91], the author shows that bugs are mainly introduced during the first stage of the software development life cycle and reports that around 90% is introduced in requirements analysis, design and implementation activities. The cost of correcting a bug in the late stages of the software development lifecycle becomes dramatic in comparison to early detection of the defect. It is inevitable to propose methodologies that target early detection of defects in the first stages of the software lifecycle. The overall goal of electronic embedded system design is to balance production costs with development time and cost in view of performance and functionality considerations. In other words, engineers are encouraged to shorten the

overall design and validation cycle without compromising quality, reliability, and cost targets.

According to a released study commissioned by the Department of Commerce's National Institute of Standards and Technology (NIST 2002), software bugs adds a cost overhead on the U.S. economy of about $59.5 billion annually(0.6 percent of the gross domestic product). The study also confirms that a third of these bugs could be improved via an improved Validation and Verification activities that allow the early detection of defects. An estimated 22.2 billion dollars could be saved via finding a higher rate of bugs in the same development activity that introduced them. Currently, over half of all defects are not found until the last testing activity in the development process (validation test) or during post-sale software use (operational life). The current automotive software growth, the need to decrease cost while enhancing quality and the explicit target of discovering faults in the early design stage as opposed to late testing stage are several motivations behind the research introduced in this dissertation.

2.2 What is AUTOSAR?

AUTOSAR (AUTomotive Open System ARchitecture) is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers and tool developers. AUTOSAR's birth is motivated by the following goals:

1. Management of E/E complexity associated with growth in functional scope
2. Flexibility of product modification, upgrade and update
3. Scalability of solutions within and across product lines
4. Improved quality and reliability of E/E systems

Our research is aligned with objectives 3 and 4 where we focus on improving quality of design and implementation via addressing the current shortcoming of formal methods that discouraged the automotive industry from using them[81].

AUTOSAR-standard relies on component based software design model in the design stage of the vehicular system. Software components are used in the design model and are linked through abstract component named the virtual function bus [86].

The basic unit in the application in the software development life cycle is a software component. The automotive application is now a structure of components that have different types of interfaces to talk to each other. The components can be re-usable within some applications. AUTOSAR standards describes standardized interfaces for all the application software components that are needed to build any automotive application. This ensures that there is still freedom in the functionality that is contained within the component as long as the component has standardized interfaces and could be plugged in/out of existing systems[81].

VFB or virtual function bus is the bridge that aims to connect the different software components in the AUTOSAR design model. This special component is responsible for connecting the application software components as well as handling the data flow between them. The virtual function bus is AUTOSAR's approach to model all hardware and system within a vehicular system. The approach allows the focus to be on the application as opposed to the structure of the software via the designers[81].

The presence of the virtual function bus has allowed the software components to not be aware about the other components that they communicate with. The output of every software component is given to the VFB, which dispatches the information via ports of the input of the software components that require this data which is feasible due to standardized interfaces of the software components which defines the input and output ports as well as the data format of the information that will be exchanged via the components [81].

This approach makes it possible to validate the interaction of all components and interfaces before software implementation. This is also a fast way to make changes in the system design and check whether the system will still function[83].

To support the Autosar-methodology, the consortium developed a metamodel to allow designers to describe their systems based on this metamodel. A formal description of methodology related information, which is modeled in UML was given. The benefit below are a result of this definition:

- The structure of the information can be clearly visualized
- The consistency of the information is guaranteed
- Using XML, a data exchange format can be generated automatically out of the meta-model and be used as input for the methodology.
- Easy maintenance of the entire vehicular system

2.2.1 AUTOSAR Layered Architecture

Figure 3 depicts the AUTOSAR Interfaces[80].

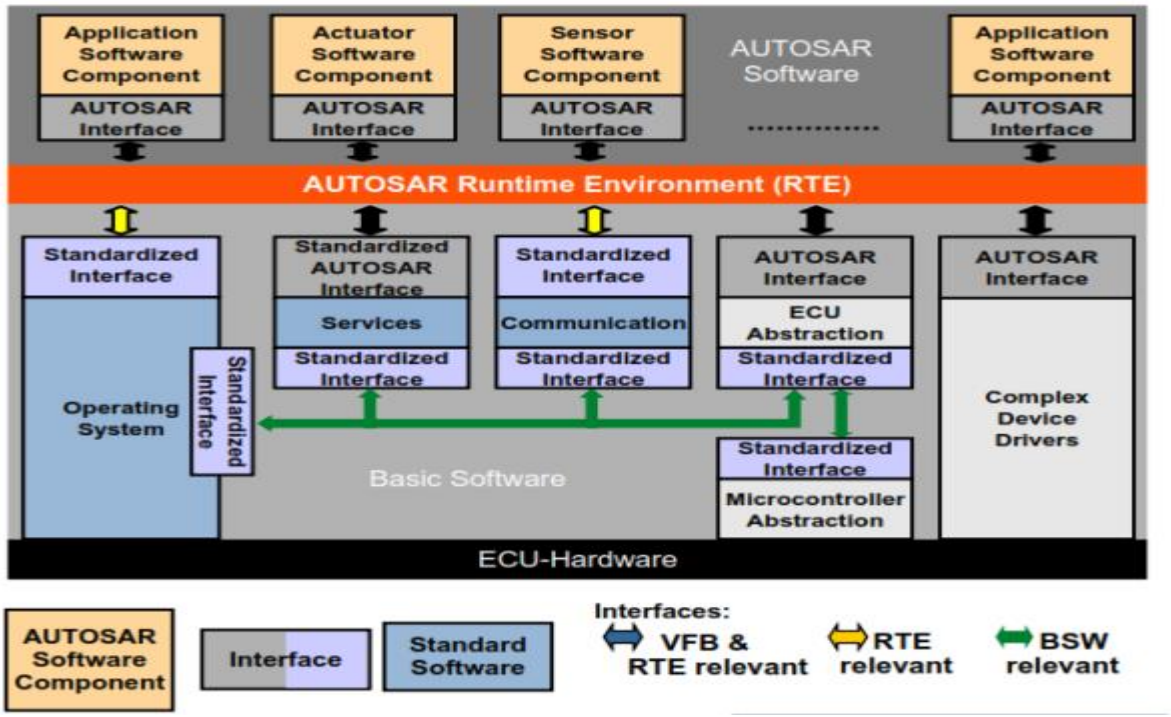

Figure 3 AUTOSAR Interfaces

Classification of Interfaces:

There are three different types of interfaces in Autosar Layered Architecture [80].

1. Standardized Autosar Interfaces:

   A Standardized AUTOSAR Interface is an AUTOSAR Interface standardized within the AUTOSAR project.

2. Standardized Interfaces:

   A software interface is called Standardized Interface if a concrete standardized API exists (e.g. OSEK COM Interface Com_ReceiveSignal & Com_TransmitSignal which are called by RTE module)

3. Autosar Interfaces:

   An AUTOSAR Interface describes the data and services required or provided by a component and is specified and implemented according to the AUTOSAR Interface Definition Language. An AUTOSAR Interface is partly standardized

within AUTOSAR, e.g. it may include OEM specific aspects. The use of AUTOSAR Interfaces allows software components to be distributed among several ECUs. The RTEs on the ECUs will take care of making the distribution transparent to the software components.

2.2.2 AUTOSAR Structure

Figure 4 shows the AUTOSAR layers while explaining the aim of each layer [80].

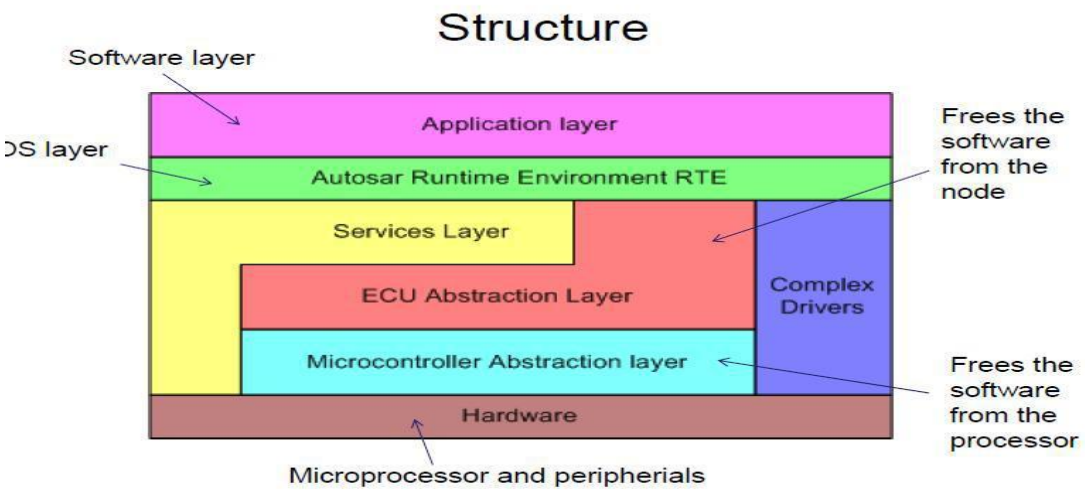

Figure 4 AUTOSAR Layered Architecture

Microcontroller Abstraction layer aims to free the software from the processor, ECU abstraction layer frees the software from the physical ECU properties, is the lowest software layer, and it contains internal drivers which are modules that have direct access to peripherals and microcontroller. ECU Abstraction layer interfaces the drivers. It also contains external drivers. It offers interfaces for access to peripherals regardless of their location in the microcontroller (Internal or external) and their connections (port pins, interfaces) and mainly makes higher software layers independent of the ECU hardware layout. Services layer is the highest layer of the Basic Software and offers operating system functionality, network communication and management services, memory services, diagnostic services, ECU state management, mode management, logical and

temporal program flow monitoring. The RTE is a layer that provides communication services to the application software. The software communication with each other via services in the RTE[80].

2.3 ISO-26262

ISO-26262 is a functional safety standard that publishes its objectives as [13]:

- Provides an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases;
- Provides an automotive specific risk-based approach for determining risk classes (Automotive Safety Integrity Levels, ASILs);
- Uses ASILs for specifying the item's necessary safety requirements for achieving an acceptable residual risk; and
- Provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved.

Our research fulfils recommendations made by the standard in the validation and verification activities of the design recommendation. The steps recommended by the standard include semi-formal verification of the design and formal verification of the design. Our proposed framework support these guidelines. Test derivation guidelines recommend checks to be based on requirement of analysis, boundary conditions and equivalence partitioning. We will show that our formal framework support these guidelines while checking the model to report any violation.

The standards has been published in 10 sections, namely, vocabulary, management of functional safety, concept phase, product development at the system level, product development at the hardware level, product development at the software level, production and operation, supporting processes, ASIL oriented and safety oriented analysis and finally guidelines on ISO 26262.

ISO 26262 automotive safety lifecycle envelopes the entire lifecycle all the way to production. This incorporates the necessity of having a safety manager who manages the evolution of a safety plan and develops a set of measures inclusive of a safety review, audit, and assessment. These measures are intended to be the framework for developing any E/E system [13].

ASIL is a primary element in ISO 26262 compliance. The ASIL is determined at the beginning of the development process. The expected functions of any system are specified and analyzed in comparison with potential hazards. The ASIL asks the question, "If a failure arises, what is the side-effect on human lives, whether it be the driver or pedestrians?" The risk estimation is established based on several probabilities, including the probability of exposure, the possible controllability by a driver, and the possible outcome's severity if a critical event occurs, leads to the ASIL. The ASIL is not related in any way to the technologies utilized within the system. It only focuses on any potential harm that may come to the driver or road users in case the system fails. [13].

Every safety requirement has to be assigned an ASIL value, which can be A, B, C, or D, with D having the most stringent safety critical processes and strictest testing regulations. ISO 26262 standard related all guidelines and recommendations based on the ASIL level and identified the least set of testing requirements based on ASIL level as well. This governs the approaches that should be utilized for test once the ASIL level is determined based on a system level safety goal, which describes what the system should do to ensure safety [13].

In the example of a windshield wiper system, the analysis of the safety of the system will render that the potential loss of wiper function can impair the visibility of the driver and thus lead to a critical injury to the driver or a potential pedestrian. In this case, a high ASIL level is assigned to the system. The system development will have to follow all the guidelines in the standard that are applicable to this ASIL level. This guidance is meant to be in addition to existing safety practices. Existing measures to manufacture automobiles

could already be utilizing a good number of the approaches recommended by ISO-26262. The publishing of the standard just aims to standardize the practices across the industry. [13].

Hardware qualification is also part of the standard and it lists two main objectives in this area. The first is to show how the individual parts are part of the big system and to define failure modes and asses them. Regular existing qualification can be done for elementary hardware components. Complex hardware components has to go through the analysis phase and the ASIL level assignment phase and testing based on assigned ASIL level. The hardware qualification is done via testing the part as a unit in different environmental and operational conditions. Numerical methods are then used to analyze the results and grouped into a qualification report that also documents the testing process, any assumptions and different input categories. [13].

The activities to qualify a component is documented in ISO26262. It can be summarized as defining functional requirements, the utilization of resources, and analyzing software behavior in case of failure or overload situations. Whenever an existing software component is qualified, the process to integrate it to an existing system or re-use it becomes much simpler. The re-use aspect is really encouraged in AUTOSAR and simplified via ISO 26262. AUTOSAR encourages the use of well-established entities that have been used in several projects and ISO 26262 describes how to easily qualify such entities for re-use. Example of such entities can include operating systems, libraries, databases or even driver software. [13].
The qualification via the standard for these entities would be to check their behavior under normal conditions and abnormal ones (inducing faults to see system reaction). Any Software defects are analyzed from a data path and runtime perspective as well and addressed throughout the design process [13].

Existing components whether they are hardware or software components can also comply with ISO 26262 via “proven in use” argument. This is a special clause in the ISO where it

describes means to comply a component via proving that it has been used long enough by other components with no reported failure. This was included in ISO 26262 to ensure that existing systems that have been in production with no incidents for a long time does not have to comply with the guidelines for safety in development life cycle. It makes no sense to ask a module that has been deployed in several cars for years to apply standard guidelines. The compliance of such components is established by proving that they have been utilized in real world and can be shown to be defect free and reliable. Combining components that are certified based on the new standard guidelines with those that have been deployed for a long time is believed to reduce the overall complexity of the system. [13].

A major challenge in any adoption of a new standard such as ISO 26262 is how to apply the standard to existing processes. Usually, this is initiated via a pilot project to evaluate the delta and the effect of the process on existing processes. Existing pilots tend to show that ISO-26262 is similar to existing processes as the industry was already safety oriented. Industry already saw the advantages of evaluating risks and doing program safety analysis throughout a project and starting with the early phases of the project definition to account for hazard analysis [13].

In summary, ISO 26262 could be seen as a standard that pushes for early understanding of program goals and impacts, analyzing these goals and impacts from the start of the project, linking the program to a correct ASIL accordingly and finally fulfilling these requirements through ASIL guidelines all the way to production. [13].

Testing is critical in the development life cycle as described in ISO 26262. It is crucial that systems react reliably towards testing scenarios. It must be shown that system behavior always stay within a safe limit that is identified during the analysis phase of the system even when exposed to expected and unexpected human or environmental inputs. It is expected that increasing the test quality of the system will increase the performance of the product, its quality as well as its reliability and recall rate. It is well known that the

cost of finding an issue in production is far less than finding it in the field. The best scenario would be to find the issue in the design stage where the cost is much less. [13].

The standard includes understanding of the fact that the above can be accomplished via software tools. The tools could be used to automate a guideline or task within the development lifecycle of the component. The standard describes a complete section on tool qualification where a tool is evaluated based on a Tool Confidence Level Metric [13].

The inputs and outputs of any tool decide the use-cases that will be used to test the tool. Once the tool is put under test, the output is used to determine the Tool Confidence Level (TCL). The TCL and ASIL determine the level of qualification required for the software tool. Two specific parts are used to determine the confidence level:

- The possibility of a malfunctioning software tool and its erroneous output can lead to the violation of any safety requirement allocated to the safety-related item or element to be developed
- The probability of preventing or detecting such defects in its output

The Tool Confidence Level can be TCL1, TCL2, TCL3, or TCL4, with TCL4 being the highest level of confidence and TCL1 being the lowest level of confidence [13].

2.3.1 Tool Qualification

ISO 26262 puts in place requirements to qualify tools that help in the product development lifecycle or that adopt technologies that are recommended via the standard. Requirements include the necessity to define an ASIL level and the tool must have a user manual, unique Id, version number, some installation guide document, the needed installation environment and details of the features of the tool. ISO 26262 requires the following tool qualification work products:

- Software Tool Qualification Plan
- Software Tool Documentation

- Software Tool Classification Analysis
- Software Tool Qualification Report

STQP or software tool qualification plan needs to be put in place in the early development of any element or item that is impacted via the safety plan. It mainly targets two areas, namely, showing that there is a plan in place to qualify the tool and enumerating the use-cases that show that the tool has been categorized with a correct ASIL with a good degree of confidence. STQP plan needs to include a unique tool identifier, a version number, predefined ASIL level, use-cases, features, user manual as well as the needed environment to run the tool.

In order to define Tool Confidence Level (TCL), a Software Tool Classification Analysis (STCA) was put in place to guide the assignment of the TCL factor. Two main aspects define the TCL, the first is the Tool Impact (TI) and the second is the Tool Error Detection (TD). TCL is defined based on these two values as described in the ISO-26262 standard.

Tool impact can either be classified as TI0 or TI1. If the tool supplier can provide details of why the tool mal-function can never affect a safety requirement, then the tool impact can be assigned TI0. If no such argument can be given then the tool impact is assigned as TI1.

In the case where a tool generates documentation and the documentation has a typing defect, then this mild issue does not trigger or cause a safety requirement incompliance in any way. It would be safe to assign such a tool an impact factor of TI0. In the case where a tool can potential effect the behavior of a system in any way based on its output, then an impact factor of TI1 is assigned.

On the other hand, Tool error detection can be assigned a range between TD1 and TD4. The assignment is based on the confidence level of the tool. A high confidence level tool is assigned TD1. TD2 is assigned for tools with moderate confidence level, and TD3 is assigned for tools with low confidence level. If the tool could potentially mal-function

and this can be detected via random cases as opposed to systematic ones, then TD4 is the assigned value.

In static analysis tools case, a TD2 (moderate confidence degree) is assigned to the tools. This is because static analysis tools detect a subset of defects that can exist in the model/design. The tools cannot report all violations in a design model. As a result, TD2 indicates that additional testing or tools are needed to ensure that the model is correct which is interpreted into a TD2 or a moderate confidence level in the tool.

A TCL factor can be assigned once a tool has already been evaluated from an Impact (TI) and error detection (TD) levels. TCL can range from 1 to 4. It is possible that a tool can be assigned different TCLs depending on different exercised use-cases. In which case, the highest TCL value is the one used for the tool. The above classification needs to be done for every tool. [13].

Finally, a qualification report that contains the outcome of the qualification activities and the proofs showing that the assignments were done properly and the all qualification guidelines have been met. Any unexpected outcome should also be well captured in the report.

The standard also supports tool qualification based on the usage history of a tool. If the tool has been used extensively then a high confidence factor could be assigned in the qualification endeavor. This definitely will help existing suppliers from a cost and time perspective in tool qualification since their tools have been used extensively in projects. With that said, the tool must show that qualification is done for every safety requirement before being used in developing any safety item. In such case, the tool must show that:

- It was historically used for a similar objective and similar use-cases
- The tool has not gone through major specification updates
- The tool has not caused a previous safety violation in previous safety requirements.

If Tool X was used to validate Car A ABS (Anti Brake System) system. Tool X has not violated any safety requirement for this module in car A. In this case, the standard allows using Tool A for ABS system in Car B given that it is a similar Car and the ABS system will be used in that car with a similar manner. [13].

2.3.2 ISO26262 Architectural Design level Guidance

The first objective of this subphase is to develop a software architectural design that realizes the software safety requirements. The second objective of this subphase is to verify the software architectural design.

The software architectural design represents all software components and their interactions with one another in a hierarchical structure. Static aspects, such as interfaces and data paths of all software components, as well as dynamic aspects, such as process sequences, state machines and timing behavior, need to be described.
In order to develop a single software architectural design both software safety requirements as well as all non-safety-related requirements have to be fulfilled. Hence in this subphase safety-related and non-safety-related requirements are handled within one development process.

The software architectural design has to provide the means to implement the software safety requirements and to manage the complexity of the technical safety concept [13]. Inputs to this phase are:

1. Software safety requirements specification
2. Safety plan
3. Verification Plan
4. Other supporting documents/resources include [13]:
    - Technical safety concept

- System design specification
    - Design and coding guidelines for modelling and programming languages
    - Guidelines for the application of methods (from external source)
    - Software tool application guidelines
    - Qualified software components available

To ensure that the software architectural design captures the information necessary to allow the subsequent development activities to be performed correctly and effectively, the software architectural design shall be described with appropriate levels of abstraction by using the notations for software architectural design listed in Table 1 [13].

Table 1 ISO26262 Recommended Design Abstraction Notation

Methods		**ASIL**			
		A	**B**	**C**	**D**
1a	Informal notations	++	++	+	+
1b	Semi-formal notations	+	++	++	++
1c	Formal notations	+	+	+	+

Adherence to design guidelines through verification methods is shown in Table 2. Our research aims to empower the ISO-26262 guidelines with tools to achieve semi-formal and formal verification of the design.

Table 2 Methods for the verification of the software architectural design

Methods		ASIL			
		A	B	C	D
1a	Informal verification by walkthrough of the design-a	++	+	o	o
1b	Informal verification by inspection of the design-a	+	++	++	++
1c	Semi-formal verification by simulating dynamic parts of the design-b	+	+	+	+
1d	Semi-formal verification by prototype generation / animation	o	o	+	+
1e	Formal verification	o	o	+	+
1f	Control flow analysis-c, -d	+	+	++	++
1g	Data flow analysis-c, -d	+	+	++	++

a Informal verification is used to assess whether the software requirements are completely and correctly refined and realized in the software architectural design. In the case of model-based development this method can be applied to the model.

b Method 1c requires the usage of executable models for the dynamic parts of the software architecture.

c Control and data flow analysis can be carried out informally, semi-formally or formally.

d Control and data flow analysis may be limited to safety-related components and their interfaces.

Chapter 3. Literature Survey

This chapter introduces the research's state of the art. The existing software defects as well as the existing verification and validation techniques used in the software engineering cycle.

3.1 Software Defects

In order to claim there is an attempted solution that targets improvements in embedded automotive software safety, it is imperative to capture problems that this solution need to address and the software defects that any solution needs to target. IEEE defines software defect to be a software related discrepancy between a computed, observed, or measured value and condition and the true, specified, or theoretically correct value or condition [8]. To guide the measures for software defects corrective actions in some industries, software defects in more than one domain / standard gets classified as negligible, significant and catastrophic defects. Measures to identify problems in a typical software program that is millions of lines of code are crucial. It is reported that the average embedded device has 1 million line of code and doubling each year [9], a modern passenger jet, such as Boeing 777 depends on 4 million lines of code[10], cars average 400 million lines of code so far[9]. Researchers have attempted to classify software defects that should be addressed to guarantee software reliability and deterministic behavior in more than one way.

Lutz classified defects based on existing identified defects in software safety critical embedded projects as shown below [11]:

1. Program Faults

- Internal Faults (Syntax programming and language's semantics). These are coding defects that happen internally in the software. Not many of these internal software defects appear during system testing and concludes that existing process flows

discover most of these defects and that there should be no focus on these defect categories.

- Interface Faults (Interactions with other systems components)
- Functional Faults (operating faults; omission or unnecessary operations; conditional faults; incorrect conditions or limit value). These usually happen due to behavior not satisfying the functional requirements. These defects are the most frequently occurring software ones and could be easily detected via formal approaches and verification approaches.
- Behavioral faults (incorrect behavior that does not conform to requirements). These usually present half of the faults uncovered during system testing and could be easily detected via formal approaches and verification approaches.

As classified above, functional and behavioral faults represent the bulk of the defects and it is recommended that they be addressed via formal methods.

Analysis of program faults categories in Voyager and Galileo spacecraft projects conclude the following software defects root causes [11]:

- Interface defects are mainly driven by communication errors between the members in the development team or communication errors between the development team and other teams. Additionally a primary defect reason for interface defects is misunderstood hardware and software interface specifications. A typical identified use-case was the misconception of the initial state of relays or unexpected timing patterns that were not explicitly indicated in the specifications.
- Functional faults were observed to be mainly due to defects in identifying requirements or implementing them. An example of this was reported as being assumed condition or limit values that were not explicitly identified as requirements and were incorrectly assumed.

Conclusion was that program faults category of defects is mainly caused by problems with understanding/mapping requirements within the software [11].

2. Human Errors

- Coding or editing errors
- Communication errors within a development team
- Errors in recognizing requirements
- Errors in deploying requirements

3. Process Flaws

- Utilized methods are old (Inspections and ad-hoc based testing approaches)
- Inaccurate or incomplete specifications that results from lack of communication between programmers and designers.
- Incomplete or missing interface specifications between software and hardware engineers.
- Inadequate requirements documentation that lacks complete description, which lead to misunderstanding those requirements. Requirements not identified/understood (Inadequate design)

The above classification was based on existing two spacecraft's projects and identified 387 software defects in their software development, namely Voyager (18,000 lines of code) and Galileo (22,000 lines of code). Table 3 lists a summary of program faults classification in both projects:

Table 3 Fault Classification in Spacecraft's Project

	Program Faults		Safety related program faults	
Faults Types	Voyager (134)	Galileo (253)	Voyager (75)	Galileo (122)
Internal Faults	1% (1)	3%(7)	0%(0)	2%(3)
Interface Faults	34%(46)	18%(47)	36%(27)	19%(23)
Function and	65%(87)	79%(199)	64%(48)	79%(96)

	Program Faults		Safety related program faults	
behavior Faults				

The above numbers show us that the focus should be on Interface, functional and behavioral faults, as they are mainly the bulk percentage for the defects identified in both projects.

Other existing research efforts focused on software design, implementation, and testing problems [11] which they enumerated as follows:

1- Omission: The failure of a system to generate an output to an input.

2- Value: The failure of a system to produce the correct output to a given input although the output was generated in the correct time requirement.

3- Timing: The failure of a system to generate the correct output towards an input in the required time interval/constraint

4- Byzantine: Any failure that causes an invalid input-output combination.

Edward A. Lee focuses on the problems with embedded software and identifies them as follows [12]:

1. Resource limitations (limited memory, small data word sizes, and relatively slow clocks). Although there has been huge progress in semiconductor industry in the past, embedded industries fall short of utilizing designs that utilize the new artifacts. Examples include,

- Rarely see embedded development utilizing object oriented techniques, such as inheritance and polymorphism.
- Processors used for embedded systems rarely use memory hierarchy techniques that make use of virtual memory spaces to deliver faster execution using caches.
- Automated memory management (Allocation, de-allocation and garbage collection) is rarely utilized in embedded system.

2. Most software systems abstract away time via ordering within the software system development. In embedded software systems, integrations of software and hardware takes place. Physical systems are concurrent and temporal. Actions and reactions happen all the time simultaneously and concurrently thus temporal properties are crucial to embedded safety systems. To the contrary, time is abstracted away and replaced by ordering. Languages such as C/C++ and Java allow definition of the order of actions but not the timing. The lack of timing in the core abstraction is a flaw in embedded systems. Embedded frameworks such as Simulink (Mathworks), TinyOS(Berekley) and SCADE (Esterel Technologies) have no threads or processes.

3. Developers find it extremely difficult to debug communication between threads. As a result the behavior of concurrent systems is always not fully comprehended or defined which puts their reliability to question. The only attempt to control the interaction is based on mutex and semaphore technologies to try to control parallel access, which are methods that have been defined in the 60s. Most of the time, there are race conditions in the software that are manifested in production as opposed to in testing phase of the software which causes a system to be non-deterministic.

Bugs introduced because of misusing sempahores or mutex are very difficult to troubleshoot and almost impossible to be identified during testing. It is possible that a program can be running correctly for years and then a flaw that is introduced at design time is uncovered. Current concurrent development shortcomings are due to lack of proper concurrent software engineering processes (Good reviews or specifications, proper testing, and proper planning of concurrent systems design). It is possible to improve this via formal methods although it is believed that the program itself in such cases will be difficult to understand which impacts the reliability factor of the software.

The main dilemma is to be able to capture concurrent systems abstractions while retaining understandability of the programs and design. In such case, these abstractions need not be much more difficult compared to general non concurrent system. [12].

Software *V&V helps* the product designers and test engineers to confirm that a right product is build right way throughout the development process and improve the quality of the software product. It makes sure that, certain rules are followed when developing a software product and also makes sure that the developed product fulfills the required specifications. This reduces the risk associated with any software project up to certain level by helping in detection and correction of faults, which are unknowingly done during the development process.

The standard definition of verification is: "Are we building the product RIGHT?" e.g. *verification* is makes sure that the software product is developed the right way. The software must confirm to its predefined specifications, as the product development goes through different stages, an analysis is performed to ensure that all required specifications are met.

The *verification* part of *V&V* comes before validation and incorporates *software inspections*, *reviews*, *audits*, etc. During the *verification*, the *work product* (the ready part of the software being developed and various documentations) is reviewed / examined by one or more persons in order to find and point out the bugs in it. The *verification* helps in prevention of potential bugs.

The standard definition of validation is: "Are we building the RIGHT product?" e.g. a software product must do what the customer expects it to do. The software product must functionally do what it is supposed to, it must comply with any functional requirement set by the customer. *Validation* occurs at the end of the development process in order to determine whether the product complies with specified requirements. *Validation* starts after verification ends (after coding of the product is completed). *Testing* methods are basically carried out during the *validation*.

3.2 Software Verification and Validation Techniques

Whatever the size of project, software *V&V* greatly affects software quality. Software that has not been verified has little chance of working. Defects could lead to an operational failure (bug) or non-compliance with a requirement. The objective of software *V&V* is to

reduce software Defects to an acceptable level. The *V&V* techniques must be applied at each stage in the software process. It has two major objectives

1) Discovery of bugs in a product and

2) Assessment of whether or not the product is useful and useable in an operational situation.

V&V must establish confidence that the software is fit and safe. Confidence is certainly subjective and depends on many factors such as software criticality which is very high in automotive domain. The *V&V* consists of numerous techniques and tools, often used in combination with one another. Processes such as ISO-26262 wrap the software development process and utilize all existing V&V techniques via recommendations and guidelines.

Software *V&V* both use *static* and *dynamic techniques* of product checking to ensure that the resulting software product matches with its specifications and that the software product as implemented meets the expectations of the customer. In fact, *dynamic techniques* involve the execution of the software product under test, whereas *static techniques* do not.

- *Static techniques* (*Review* and *Proof*) are concerned with analysis of the static product representation to discover defects throughout all stages of the software life cycle. It may be complemented by *tool-based document* and *code analysis*.
- *Dynamic techniques* (*Testing*) are concerned with exercising and observing product behavior. The product is executed with test data and its operational behavior is observed.

3.2.1 Process Based Approaches

Domain specific regulatory bodies put down process measures to guide any industry in its software engineering process in order to govern software under development and put strict measures in different software engineering cycles that aim to minimize software defects that are caused by process flaws as a result of miscommunications, ambiguities, or misunderstandings. There are over 250 standards and the list below shows some of the existing software engineering standards that are available [14]:

1. AECL CE-1001-STD REV.2: Standard for Software Engineering of Safety Critical Software
2. ANSI/AAMI/ISO 14971: Risk Management - Part 1: Application of Risk Management to Medical Devices
3. ANSI/AAMI/IEC 62366:2007: Medical devices - Application of usability engineering to medical devices
4. ANSI/IEEE 7-4.3.2: Application Criteria for Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations
5. ANSI/ UL 1998: Software in Programmable Components
6. BS EN 50128:2001: Software in Programmable Components
7. EIA SEB6A: System Safety Engineering in Software Development.
8. IEC 60880: Software for Computers in the Safety Systems of Nuclear Power Stations
9. IEC60950-1 Amd.1 Ed 2.0: Information technology equipment
10. IEC 61508-1/2/3/5/6: Functional Safety of electrical/electronic/Programmable electronic safety related systems
11. IEC 62304: Medical device software
12. IEEE 1228: Software Safety Plans
13. ISO IEC TR 15026: Systems and software engineering. Systems and software assurance/
14. ISO / IEC 27002:2005: Information technology – Security management
15. MIL-STD-882D: System Safety Program Requirements
16. RTCA DO-178B: Software considerations in Airborne Systems and Equipment certifications
17. SAE AS9006: Aerospace Software Supplement for AS9100A
18. ISO26262: Road Vehicles Functional Safety

There have been several attempts to evaluate standards and argue for/against their effectiveness in software engineering. One of these attempts was the SMARTIE (Standards and Methods Assessment Using Rigorous Techniques in Industrial Environment) which was a collaboration aiming to provide an objective assessment of an

existing standard since it argues that there is currently more than 250 standards in the market. [14]

The study reports that many standards are not really standards at all. Many "standards" are reference or subjective requirements, suggesting that they are really guidelines (since degree of compliance cannot be evaluated) and recommend that organizations with such standards should revisit their goals and revise the standards to address the goals in a more objective way. More generally, they found that standards lack objective assessment criteria, involve more process than product, and are not always based on rigorous experimental results. Thus, their recommendation was that software engineering standards be reviewed and revised. The resulting standards should be cohesive collections of requirements to which conformance can be established objectively. Moreover, there should be a clearly stated benefit to each standard and a reference to the set of experiments or case studies demonstrating that benefit. Finally, software engineering standards should be better balanced, with more product requirements in relation to process and resource requirements [14]. In summary, SMARTIE project findings were:

- Standards define a best practice; however there is no consensus about what is best practice.
- Standards heavily over-emphasize a process rather than a product.
- Software standards try to assure product quality through a good development process.
- The standards outline a set of mandatory requirements. However these requirements are not clear or precise, leading to the standards becoming 'codes of practice' or 'guidelines'.
- Standards prescribe, recommend or mandate the use of technologies that have not been objectively validated.

Standards are too big, usually extremely large documents that address the complete system development life-cycle. This makes them hard to apply.

3.2.2 Static Techniques

The main aims of static analysis is to find improper practices in the code without executing it. The improper practices are based on historic findings or practices that have led to bugs in existing systems. Static analysis is very popular and used through the automotive industry and recommended by ISO-26262 in all ASIL levels. They do uncover issues in the model structure, data or control flow as well as ensuring syntax accuracy. There are several techniques that can be categorized as static analysis ones. Sections below give a brief on each.

3.2.2.1 Review

Some product output is presented to other project members, managers, technical engineers, customers or end-users to review the output and provide feedback based on experience. The review usually ends with an approval conditional some feedback to be implemented or a rejection. If the product in the review is rejected, then another review involving the same attendees shall be organized. A review can be utilized to check any work product during the project lifecycle including requirements or specifications. Four types of reviews have been introduced via IEEE (IEEE Std. 610-1990) to conduct software verification, namely, technical review, inspection, audit and walkthrough.

These reviews are all "formal reviews" in the sense that all have specific objectives and explicit rules of procedures. They expect to identify defects and discrepancies of the software regarding the original specifications, plans and standards.

3.2.2.2 Technical review

A technical review is intended to assess a review item, which could be source code or a document, to ensure that the item in review conforms to specifications, complies with standards or procedures, any previous required change was properly included, no new issues were introduced as a result of any requested change.

3.2.2.3 Walkthrough

A Walkthrough is usually the first attempt to evaluate a project element such as a document, design, some model or even source code. The objectives of the walkthrough includes early identification of potential defects and proposing solutions towards these defects. It is also possible to consider the walkthrough endeavor as an educational one for team members and to avoid future defects of similar nature via different team members.

3.2.2.4 Inspection

Inspection can be used for the detection of defects in detailed designs before coding and during the coding stage. Inspection may also be used to verify test cases. A study done by Fagan (Fagan 1986) has shown that inspection could detect over 50% of the total number of defects introduced in development stages. IEEE (IEEE Std. 610-1990) considers that inspection is a more rigorous alternative to walkthrough, and is strongly recommended for software with stringent reliability, security and safety requirements.

3.2.2.5 Audit

In order to ensure that requirements, standards, procedurers, coding guidelines, licensing and contractrual agreement compliance is adhered to, an audit is conducted in an independent fashion. The audit is usually done via members that are not part of the development team.

3.2.2.6 Proof

A proof is a logical expression ensuring that software is correct. Testing on the other hand only shows that a specific input can generate a specific output. Alternatively, proof shows that inputs given a set of pre-conditions will result in defined post-conditions being met. Proof is usually based on formal techniques that is based on mathematical equations being solved. Any requirement is mapped to a mathematical equation and checkers are launched to check if given a set of values and pre conditions happen, this equation can never violate post conditions. Another name for this approach is formal verification. Proof techniques are usually shown to ensure specification compliance in comparison to actual design or code.

Proof techniques are often used on critical software products. They often have precise and logical specifications with no loopholes and they require being highly reliable, since failures in this kind of products may lead to deathly consequences. Some areas where proof techniques which have been successful are for the specification and verification of safe and critical products such as aircraft avionics, nuclear power plant control and patient monitoring. Automotive engineers are not familiar with proof techniques contrary to aeronautic or defense engineers. Software testing is a widespread V&V technique in automotive industry. Proof techniques are not widely used in automotive industry. In fact, the difficulty of expressing software requirements in the mathematical form necessary for formal proof has restricted a wider application of this technique.

3.2.2.7 Tools

As part of existing attempts to address safety in software, researchers / industry and conformance bodies started identifying best or to be avoided practices that every programmer should abide by. MISRA, the Motor Industry Software Reliability Association was started in the early 1990 and was primarily concerned with safety aspects of electronic systems. Initially, the project was expected to develop guidelines for vehicle based software. One of the major outputs of this effort was MISRA-C which was an attempt to develop an embedded C programming standard/guideline that addresses shortcomings in C language or practices that could lead to a software failing and thus impacting or influencing safety.

Software tool vendors take such guidelines and attempt to automate the rules validation across the software under development. Such tools are called static analysis tools since the analysis of source code takes place without execution. Because static analysis does not require execution of the code, analysis for defects and vulnerabilities is done throughout the software development process, and analysis conducted across all code paths.

Static analysis is simply looking for signatures of defects or patterns that have caused defects historically in already developed programs. Sometimes, these patterns are vague. Static Analysis main flaw is that it can give a good number of false positives where it reports many violations that are false or meaningless. Static analysis also looks for patterns that are already known. So, any new defect cannot be identified via static analysis tools. Research is ongoing on static analysis to extend its scope and make it a reliable step in the software development cycle [20].

Static analysis tools started emerging in the late 70's. The first generation of such tools started with the Lint tool. Lint was well perceived by developers and project managers when it was first released. Developers were able to run a tool that automatically detects software defects in the early stages of implementation and as a result, it was very easy form them to correct such defects. This gave developers confidence in their code quality before release. Lint was utilizing SAT or Boolean satisfiability as well as path simulation. It also used compilers to be able to detect defects. Lint is seen to be the first usable static tool. [15]

As with the rest of the tools conducting static analysis, Lint tool was never designed to detect issues that can lead to run-time problems. Its main objective was to indicate code constructs that could potentially be problematic or code constructs that could lead to portability issues so that developers can fix them in the code. Problematic code or non-portable code could be viewed as code that is correct from a semantics and syntax perspective but could potentially behave in a way that was not intended by the developer due to its structure or composition. The dispute about marking problematic code is that most of the time the code would work without changing it, the same for compiler warnings. As a result, this tends to be ignored and the analysis capabilities of the tool's reported output would not be efficient due to high noise rate where only one issue out of 10 is a real defect. [15]

As a result, developers ended up wasting time trying to analyze which of the reported violations are real and which are not. Developers were asked to do manual review to analyze the output of the static tool analysis. The overhead of doing output analysis manually was exactly why static tools was introduced in the first place. As a result, Lint was never deployed and trusted on a massive scale in the industry and it only survived some success in a few organizations. Some of the initial Lint tool releases are still in use by product development in some organizations until today. [15]

Static analysis remained for almost 20 years as a myth in identifying defects as opposed to an actual dependable tool. A new generation of static analysis tool were released in early 2000, Stanford Checker, which was seen to offer good value to make it a reliable tool for defect detection. Unlike first generation tools which were only looking for matching patterns, this tool utilized path coverage and was able to reveal more defects that had run time failure indications. The tools ran on entire project code bases as opposed to individual files. This switched the focus from, problematic code constructs to defects that had run-time implications. The main theme of the new tool developers was to understand the code composition, use complex technologies, namely, path analysis, and inter procedural analysis to comprehend the program flow between functions in a complete system. [18][19]

Although second generation tools were adopted by organizations, they still failed to strike a balance between reliability and scalability. Some tools were accurate when it comes to a subset of defect types but failed to work with systems that had millions of lines of code. Other systems ran faster but ended up with output like Lint Tool where many false positives were reported. The tools did show defects at a reasonable ration but only when you restrict the input parameters during the execution of the tool. The dilemma of trying to balance between accuracy and scalability to avoid false positives remained. This was the problem that caused first generation tools not to be widely adopted and they remained in second-generation tools, which also reduced the rate of their adoption. So, in a nutshell, the technologies used in second generation tools were more advanced, the

results were still far from what developers can claim as an accurate output. Second generation tools also endured many issues due to the heterogeneous build and development environment. The development and build environments are not standard and are different in every organization which led to great pains, time and cost to integrate any tool to existing build and development environments.[15][16]

Most recently, a new generation of tools emerged that are based on SAT solvers and path analysis technologies. SAT is described as defining if variables of some formula can be assigned in a way to make the formula end up being evaluated to true. It also tried to find if no assignment exists that could lead the formula evaluation to be true. This would imply that a function, which would be represented via the formula, is false given all input parameters and variable assignment. In such case, the formula is declared as unsatisfiable; else, it is satisfiable. The conclusion was that such static analysis tools were able to find real defects and minimize the false positives. There was also a claim that the underlying used technology can allow for further enhancements in static analysis [17].

Furthermore, some further programming language specific static analysis concepts were introduced. In automobile design, the UK-based Motor Industry Software Reliability Association (MISRA) mapped their concerns of safety in software into a set of documentation limitations. Knowing that most automotive development happens in C, they collected the pitfalls of C language and published guidelines in order to make automotive programming in C safer.[21]

The result of UK's MISRA association endeavor was a guidelines document to aid any developer in using the C language. The guidelines, which was published in 1998, were later given the acronym, MISRA C. It was a 70-page document that described all C language pitfalls based on existing systems failures. [21]

MISRA C is comprised of 127 rules, 93 are mandatory and must be satisfied by any automotive software developer and the remaining are recommendations. In order for any

developer to confirm to MISRA C, they must show that they do not violate any of the 93 mandatory rules. Developers should also make every attempt to confirm to the advisory rules as well. Therefore, briefly, either you are MISRA C compliant fan or you are not. [21] The guidelines do not deal with issues related to invalid algorithms. It has no impact on programming style and no constraints that can stop a developer form writing stupid code. It will just ensure that your code has avoided known pitfalls of C language. [21]

Other tools out there also help in identifying code anomalies in different categories via static tools. There are tools to ensure that code is compliant to standards, not redundant, does not contain any division by zero, does not have constructs that can cause run-time exceptions, does not cause memory leaks and finally does not mis-use variables in any way. [22]. Such tools parse the source code and tries to find any of the above categories error patterns in the code. Static analysis now include control and data flow analysis, interface analysis, information flow analysis, and path analysis of software code. Nowadays, static analysis tools can identify a good number of development defects but there remains a good number of defects that cannot be detected via static analysis tools. [22].

There exists a wide range of tools for code written in C or C++. FlexeLint2 is a Unix based tool that checks C/C++ source code to find bugs, constructs that are not portable, inconsistent code constructs or redundant code. inconsistencies, non-portable constructs, and redundant code. Reasoning3's Illuma is a static tool that detects bugs in applications written in C/C++. Development teams send their code base to Reasoning3, which conducts the static analysis, analyses the tool output to filter away false positives and generates a report to be sent back to the development team. Illuma The tool focused on detecting bugs that can cause applications crashes or corruption in data such as NULL pointer dereferencing; out of bounds array access; memory leaks; bad de-allocation; and uninitialized variables.

Two static analysis tools provided by Klocwork4 are also in use by several organizations. InForce conducts automated inspection of source code to provide code metrics that can be used to identify defects, opportunities for code optimizations or security flaws. GateKeeper inspects the source code architecture and provides an assessment report that shows the cons and pros of the architecture, which reveals en evaluation of the code. The evaluations address the quality of the code, defects that are hidden, and code that is hard to maintain. Metrics also shows interdependency between modules, cyclic relationships within modules, code files that exhibit high risk, potential logical defects, and areas for improvement [22].

PREfix [22] analyzed the code to establish a call graph of the program. PREfast [22] tool is a "quick" version of the PREfix tool where specific PREfast analyses revolves on trying to identify matches in an abstract syntax tree representation of the C/C++ program in order to identify programming defects. PREfix/PREfast are used in the industry to detect defects, such as dereferencing of a NULL pointer, variables that are not initialized, using uninitialized memory, and freeing memory or resources twice.

3.2.3 Dynamic Techniques

Require model execution where they evaluate the model based on its execution behavior. Most dynamic V&V techniques require model instrumentation, the insertion of additional code (probes or stubs) into the executable model to collect information about model behavior during execution.

Software testing, a V&V dynamic technique is a widespread technique in automotive industry. In Johnson Controls, software testing represents up to 90% of the total time spent in verifying and validation a software product [93]. Moreover, in the academic research, the traditional focus of software *V&V* techniques has been the *software testing*. In fact, *testing* approaches are widely studied in academic research and deployed in software industry.

3.2.3.1 Testing

Testing is involved in every stage of software life cycle, but the testing done at each level of software development is different in nature and has different objectives. Unit Testing is done at the lowest level. It tests the basic unit of software, which is the smallest testable piece of software, and is often called “unit”, “module”, or “component” interchangeably. When more than one unit is combined together in a test, integration testing is performed. The test is conducted on external interfaces to the individual units as well as interfaces between these units. The test is often done on both the interfaces between the components as long as it can assessed from the unit under test.

System Testing focuses on end-to-end testing of an entire system rather than focus on internal components of the system. System testing makes a statement on the overall quality of the software. It is usually based on functional requirements in specifications of a system. System testing also can cover nonfunctional requirements such as maintainability, reliability and security of the system.

Finally, acceptance testing is conducted when a complete system is handed over to customers or users and aims to ensure that the system is functioning as opposed to trying to find defects.[23]

Currently there are two major activities to ensure quality in systems. The first is static analysis, which targets non-execution defects using several discussed techniques such as: inspection of the code, analysis of the program, symbolic analysis, or model checking. Dynamic analysis on the other hand focuses on methods to ensure system software quality during actual executions using actual and under real or simulated conditions. Inputs Synthesis, testing procedures and automating the generation of test environments are examples of some techniques used in dynamic analysis. Static and Dynamic techniques complement each other as one involves execution of a system and one does not. [23]

Input test cases and test results analysis depend on the testing strategy in question. The testing strategy is decided based on testing data flow. Every testing technique reveal different quality aspects of a system. Functional and Structural testing are two main categories of testing techniques.

Functional Testing is conducted when the system under test or software to be tested is seen as a black box with no concerns regarding its internals or interactions amongst its components. Test cases in functional testing are primarily based on specifications of requirements or design of the system under test. Results are sometimes called oracles or gold models. The results usually include the original requirement that is tested, the desired output in accordance to the specification, and the actual results. In functional testing, the emphasis is on the external behavior of the system.

Structural based testing of a software system means viewing the system as white box (transparent) where you see all the internal system details and need to test all internals of that system. Test cases are based on how the software got implementation or on the implementation itself. The main objective is to know specific constructs in the system and aim to test them to verify that they operate as planned by the implementation. Example include specific statements, specific program path or branch. The results of any test are compared with the planned expected results of the implementation. Evaluation of the tests are based on metrics such as coverage percentage of the statements within the system being tested. There are several metrics used in structural testing such as coverage of branches, coverage of data-flow, and coverage of paths in the system. Briefly, structural coverage is concerned with the internal composition of the system as opposed to its external behavior.

Traditionally, testing has been performed using adhoc and intuitive techniques. Testing still remains the biggest part of software development life cycle. Testers utilized both structural and functional techniques based on intuition in their testing cycles. There were no techniques, methods or theory to design testing in an efficient, automated and

structured way. Goodenough and Gerhart founded the main theorem of testing in a paper to propose a Theory of Test Data Selection. This was the first work that got published that tried to lay a theoretical foundation for testing. It categorized test data for test cases as effective if it uncovers program defects. If the selected test data does not uncover defects then this test data selection is not effective. They emphasized on statement coverage in their work. Their foundation led to various successful research on testing methods theory. Later, Huang added in his research the importance of having every statement in the program executed at least once during the testing cycle. He also emphasized that statement coverage does not guarantee that all defects will be detected. He described a term, "edge strategy", which aims to exercise every edge in a diagraph of the program at least once.

Subsequent research introduced probe insertion technique and path coverage, which appeared in 1976. Howden explained that test data needs to be selected to ensure that every unique path of a program is visited at least once. He elaborated in his work that the total number of paths in a program could be infinite and suggested that in practice a subset of program paths (or a superset) needs to be tested. Several studies were later made to evaluate the efficiency of path testing and to define an upper bound that limits the value of the subset of test cases to ensure reliable path coverage testing.

Functional testing also lacked any solid theory behind it although it was widely used in industry and academia. In the first research that tried to lay a theoretical foundation behind functional testing, Howden introduced the term design functions, which is code surrounded by comments which describe the intent of the function. He described how systematic design techniques could be utilized to design functional tests.

Further research addressed theories behind structural testing via introducing the term, domain defect. Domain defect was described as a subset input to a program that triggers an invalid path to be taken. Domain defects were described to be potentially triggered via branch statements that have incorrect predicates or invalid computations that have an

impact on predicates that are used within a branch statement. White and Cohen defined some guidance on the selection of test data to uncover domain defects. Their work described general reasons whey testing could be successful or not and proposed research direction [56].

In 1985, Rapps and Weyuker introduced data flow analysis for structural testing in their published research. They proposed guidelines on the test data selection to achieve data flow analysis. They argued that path coverage criteria could let defects go uncovered. They proposed new path selection criteria based on data flow analysis and discussed relationships between the criteria. Their work laid the foundation to select test data based on dataflow analysis techniques.

Richardson and others recommended an approach that revolves around test case selection based on specifications. Generally, functional testing that is based on specification was focused on manual hand selecting test cases rendering functional testing as simply based on selective criteria. Automation of test cases for structural testing was possible which led to the advantage of having a reliable and complete functional testing as opposed to heuristics based. Their research started using formal methods via utilizing formal specifications to empower a testing methodology that blends specification and implementation methods. Their work was the first attempt to merge structural and functional testing with formal specifications.

Boolean algebra appeared as a backbone for a testing method in the 90s. The intent was to use it to simplify, convert and analyze specifications. Boolean algebra was used to ensure that specification is consistent and complete which can definitely have a great impact on testability of the specification. Functional requirements were represented using decision tables, which makes it easier to design tests and to implement programs as well. The proposed approach was based on using both Karnaugh-Veitch charts and decision tables' Boolean algebra based techniques to capture functional requirements. This was the initial attempt to select data for test cases based on boolean algebra.

Other research that focused on improving the testing theory revolved around defining metrics that can be put in place to ensure software reliability. Traditionally, reliability was based on failure cases during the test cycle. . This of course required a big amount of data collection, post analysis of data, experience to interpret the analyze data and computations to translate the results. In the late 90s, a new method was introduced to calculate reliability based on coverage during testing. The program was mapped into a graph and every function is a node in the graph. The reliability was calculated for every node in the graph based on the number of times the node got executed during system testing. The higher this number, the more the reliability factor for this node. The node reliability value is then used to compute an overall reliability value for the entire system. Since existing research was concerned with coverage analysis, it was concluded that extending such methods with reliability analysis will increase the overall reliability of the entire system.

In 1997, a framework was proposed for functional and structural testing based on probability. Bernot and others concluded that they could ensure a high level of confidence on the correctness of a system and provide a reliability metric via selecting input test cases data based on generating data distributions that are domain specific. They proposed using techniques such as integer intervals, Cartesian products, unions, and sets that are defined inductively. Other research in the same year proposed using formal notations to describe system architecture in order to automate tests for complex systems. [56]

Another interesting research in 1997 used formal architectural description for rigorous, automatable method for integration test of complicated systems. The authors described CHAM formal language to capture the behavior of the system. The system interesting use-cases or behavior was mapped into graphs to capture all the possible behaviors of the system. The graphs were then shrunk after determining communications between entities in the system. The reduced set of graphs were then used to generate integration tests with

the coverage metric in mind or as an input parameter. The baseline of generation is assumed to be a set of reduced graphs that capture architectural features of a system.

Later research started focusing on 'off the shelf' software entities. Briefly, re-usable components that have already been design, verified and proven reliable. The work started to be based on UML (Unified Modeling Language) which gained huge momentum in the industry. Hartmann and others at Siemens worked on testing UML components via combining generation and execution of a test in a UML modeling tool (e.g. Rational Rose). A system is modeled into UML components and interactions. Test cases are then generated and run against the model to ensure correct behavior. They also proposed generating test cases from state charts and prototyped an environment, TnT, to evaluate their approach based on use-cases [25].

Component based testing approach was introduced as well. Beydeda and others suggested mapping component into a graphical representation. Testing was described as complex when a component lacks its source code in UML. The research suggested a way to merge structural and functional testing. Component is represented graphically, component-based software flow graph (CBSFG), to simplify the specification and implementation details captured in the component. The graphical representation was then used to generate test cases. Existing structural testing approaches were described to be possible on this graphical representation to classify test cases based on data flow analysis. The main components are still tested with functional techniques. [23]

A multitude of techniques were proposed in testing theory for test case generation. Examples include random generation, generation based on identified paths, generation based on identified goals, and intelligent approaches. Fault distribution is used in random test case generation to aid the generation of test cases. Control flow and data flow analysis are utilized to generated test cases in path-based approaches. Some of these methods are static while others are dynamic. Goal-oriented methods define test cases to ensure that a specific goal is taken. A goal could be a statement, a condition or even a

branch. Complex computations are used in intelligent techniques to generate test cases. [24]

3.2.3.2 Model Based Techniques

Model Driven Engineering, MDE for short, aims to raise the level of abstraction in program specification and increase automation in program development. The idea promoted by MDE is to use models at different levels of abstraction for developing systems, thereby raising the level of abstraction in program specification. An increase of automation in program development is reached by using executable model transformations. Higher-level models are transformed into lower level models until the model can be made executable using either code generation or model interpretation.

A model is specified in some model notation. Some model languages are tailored to a certain domain, such a language is often called a Domain Specific Language. A DSL can be visual or textual[68].

As in each software engineering approach quality is an important aspect of MDE. Quality in MDE can be checked, or ensured, with three different techniques: model validation, model checking, and model-based testing.

MDE is often confused with Model Driven Architecture (MDA). MDA can be seen as OMG's vision on MDE [42]. The MDA focuses on the technical variability in software, i.e. how to specify software in a platform independent way.

In a nutshell, MDE is a software engineering paradigm that focuses on creating and exploiting models, aka, abstract representations of the knowledge and activities that govern a particular application) rather than on the computing (or algorithmic) concepts, or platform setup/dependencies.

The MDE approach was driven by the need to increase productivity by maximizing compatibility between systems (via reuse of standardized models), simplifying the

process of design, and promoting communication between individuals and teams working on the system (via a standardization of the terminology and the best practices).

A modeling paradigm for MDE is considered effective if its models make sense from the point of view of a user that is familiar with the domain, and if they can serve as a basis for implementing systems. The models are developed through extensive communication among product managers, designers, developers and users of the application domain. As the models approach completion, they enable the development of software and systems.

Main components of MDE paradigm involve the following:

- Standardized or domain specific modeling languages to formalize the application structure, behavior and requirements of an application in a specific domain.
- Executable model: The ability to execute the application at the model level to identify problems/issues in the application structure and behavior before worrying about platform and programming languages dependencies.
- Transformation rules: These map the application model into language/specific and platform specific variant of the application model
- Transformation engine: Accepts as input the model and the transformation rules and generates language/platform specific variance of the application.

Using Formal Methods (FMs), which have rigorous mathematical foundations, for system development is extremely needed in the current era, especially for safety critical systems where formal proving is needed for safety or security requirements. On the other hand, Model-driven Engineering (MDE) is considered to be developing as a new model in software engineering. MDE is based on meta-modeling and model mappings in software development, and adds means to build links between domains that are similar or different[34]. It is now essential to use formal methods in system engineering, particularly in the early phases of the development process. An abstract representation of the system as a model can be utilized to ensure that the system under development fulfils the specified requirements (via simulation and model-based testing), and ensures specific properties using formal analysis (validation & verification). Indeed, there are several cases to prove the relevancy and importance of formal methods in industrial applications

and yielding very good results, many professionals and industry engineers are still hesitant to adopt formal methods. Formal methods still suffer from lack of training, which is mainly due to complex and mathematical notations that formal techniques use rather than abstract graphical notation which is more lightweight and natural for an application for a system engineer such as the Unified Modeling Language (UML) [42]. The absence of support tools for the development during the life cycle activities and the lack of integration among existing formal methods techniques and languages is also a major reason why industry has been discouraged to use formal methods.

MDE technologies with a bigger attention on automation and architecture render higher degrees of abstraction in system development by advocating models as first-step artifacts to support, analyze, verify, and eventually compile into code or into other models. Meta-modeling is a crucial concept of the MDE architecture and it is designed as a way to empower a language or formalism with an abstract notation, so as to separate the abstract syntax and semantics of the language from its alternate concrete notations.

Although the basic elements of the MDE are still expanding, some MDE fundamentals are part of the meta- modeling/programming frameworks such as Model-integrated Computing (MIC) [34], OMG MDA (Model Driven Architecture) [34], Microsoft Domain-Specific Languages (DSLs) tools (as part of the Visual Studio SDK) [34], and Eclipse/EMF [43].

Modeling languages that are metamodel based are being specified and accepted for different domains [64]. The languages address the lack by third-generation languages to ease the platforms complexities and be able to capture domain notions efficiently [34]. Inspite of the fact that meta-model based definition of a language abstract syntax is well grasped and utilized by many meta-modeling environments (GME/MetaGME, EMF/Ecore , XMFMosaic/ Xcore , AMMA/KM3, etc.), the definition of semantics for this languages class is a crucial and pending issue. Metamodeling environments are capable of coping with the most complex syntax and mapping to other models issues.

They still lack standardization and accurate support in order to give the (possibly executable) semantics of metamodels[71]73[], which is usually given in natural language. This entails that the majority of the currently employed metamodel-based languages (such as the UML) are still not sufficient for efficient analysis of models due to their strong semantics lack, which is vital for formal models analysis aided by tools. [34]

Software languages became a basic pillar in system development. Processes of Language engineering have been taken into consideration in many facets of software engineering [44]. Regarding the metamodeling paradigm of MDE for (software) language engineering, many designs have been given, which focuses on the fact that language descriptions have unique forms in different technical domains (e.g. metamodels, schemas, ontologies and grammars). Ideally, multiple languages (from a variety of technical domains) need to be integrated together on a system level approach in most software development endeavors. The engineering of a language needs to address several angles of a language: constraints, structure, textual representation, graphical representation, parser, compiler, mapper to other notations and the ability to capture dynamic behavior via executability features. Research tends to address only one of these aspects [34].

It is only recent that communities for formal methods have started to use metamodels and MDE platforms for their tools. Examples of these efforts include but are not limited to: An Event-B based metamodel and an Event B EMF based framework [34] which give a frontend that is EMF based to Rodin platform. Rodin platform is an Event-B Eclipse-based IDE that enables refinement and mathematical proof of models based on Event-B. Maude Formal Tool Environment [45] is a logic language that is executable and is suitable for object oriented systems. It delivers tool support in order to reason about specifications and a connector that is an Eclipse plug-in which allows the Maude environment to connect to other metamodeling frameworks such as KM3 which uses ATL (the ATLAS Transformation Language) transformations [46].

A transformation language GReAT (Graph Rewriting And Transformation language) which utilized the graph transformations based concepts and metamodeling within graph communities [47] has been designed with the model transformation area of Model Integrated Computing in mind. Several tools support it which aims to grant rapid prototyping and transformation tools. ITU language utilized this metamodel [48] where the authors put forward a methodology that is semi-automatic and reverse engineering based which support the derivation of a metamodel from a formal syntax definition of an existing language.

A comparable method which aims to arch model and grammer was developed by other authors in [49] and in [50]. A forward engineering process approach that aims to derive a a concrete textual notation from an abstract metamodel [34] was also developed. More recently, work in [51] shows how to apply metamodel-based technologies for the creation of a language description for the Sudoku game. Notations and tools have been developed within the ASM community to enable specification and analysis [52].

Foundation Software Engineering Group developed an Abstract State Machine Language (AsmL) at Microsoft. The aim was to develop an executable specification that is rich and based on Abstract State Machines theory, integrates with .NET framework and object oriented but AsmL does not offer a semantic structure to target the ASM method [53]. ASM popular tools also include CoreASM, TASM (Timed ASMs), extensible execution engine developed in Java, a simulator-model checker for reactive real-time ASMs , an encoding of Timed Automata in ASMs[54] , and ability to specify and verify properties based on First Order Timed Logic (FOTL) on ASM models. Several model-to-text tools are available for this flow[72].

Other endeavors allow the derivation of a language metamodel from language grammar. Examples include Ecore metamodel EMFText [55], KM3 and TEF (Textual Editing Framework) metamodel using TCS [46][58] (Textual Concrete Syntax) and Xtext [57]. Textual grammar and metamodel overview is given in [63]. Other more sophisticated

model-to-text tools that can generate text grammars from MOF specific based repositories also exist [59][60]. Such tools render the MOF- based repository content (known as a MOFlet) in text format while complying to syntactic rules (grammar). The tools are designed to be automatic and work with any MOF model in order to produce their target grammar based on a set of defined patterns so as they do not allow detailed customization of the generated language.

Work in [35] shows how object oriented software engineering flow can be on top of graphical notation and formal methods using algebra and object-Z as specification language. The flow uses UML as a modeling framework and Java as an implementation language. The work in [75] shows an approach where formal notations are introduced in safety critical software systems. Additionally, [61] introduces a transformation technique that is based on a metamodel. The technique is based on structural mapping between UML and B formal specifications in order to generate formal B specifications from UML diagrams. Most of the approaches revolve around translating graphical models into formal specifications. Work in [36] proposed an MDE-based approach to integrate several formal techniques. In the work presented by [34], formal models are introduced into MDE as domain specific languages based on constructing their meta-models. A set of transformation rules are then constructed and finally model to text rules are developed so that the models can be compiled into code. MARTE to LOTOS case study was applied on the framework with a main goal of showing different formal notations and how they can be translated into software in the software development life cycle but the approach fails to framework semantics.

The work presented in [62] discusses the broad challenges of integrating tools and interoperability of tools within a framework that is based on MDE principles. Further research focuses on the semantics specification of languages based on meta-model so that it is possible to have executability of the model within current meta-modeling frameworks such as Kermeta[74]. Similar effort with the same aim is presented in [65]

where the authors describe a framework called M3Actions framework which targets the support of EMF models operational semantics.

The work presented in [66] specified semantics of modeling languages that are visual based on Maude Formalism. A translation approach is discussed in [67] regarding the application of ASMs in order to specify MDE style execution semantics and a translation approach as well [69]. The research proposed semantic bridging to reputable formal models of computation (such as data flow , FSMs, and discrete event systems) built upon AsmL. This is done via the use of a transformation language, namely, GME/GReAT. The approach that is proposed presents sets of semantic units that are pre and well defined for potential translation/mapping endeavors. There are two cons to this approach [34]: the first being that the user needs to be aware of language semantics from scratch and based on a set of notations that are still new and did not exist previously. Secondly, in heterogeneous systems, defining language semantics as composition of some selected primary semantic units for basic behavioral categories is not always achievable. This can be due to complex behaviors, which might not be possibly reduced to existing set of combination [70].

In MDE, Automatic code generation or program synthesis techniques have been viewed in recent research endeavors to help solve the predicament of ensuring software safety by completely automating the coding phase. A code generator takes as input a domain-specific high-level description of a task (e.g.,a set of differential equations) and produces optimized and documented low-level code (e.g., C or C++) that is based on algorithms appropriate for the task (e.g., the extended Kalman filter).

This automation is claimed to increase developer productivity and is claimed to prevent the introduction of human based coding defects. Ultimately, however, the correctness of the generated code depends on the correct-ness of the generator itself. This dependency has led several monitory agencies to require that development tools be qualified to the same level of criticality as the developed software. [76]

Model-based design and automated code generation have become popular, but substantial obstacles remain to their widespread adoption in safety-critical domains: since code generators are typically not qualified, there is no guarantee that their output is safe, and consequently the generated code still needs to be fully tested and certified. Formal methods such as formal software safety certification can be used to demonstrate safety of the generated code (i.e., that the execution of the code does not violate a specified property) by providing formal proofs as explicit evidence or certificates for the assurance claims. However, several problems remain. For automatically generated code it is particularly difficult to relate the proofs to the code; moreover, the proofs are the final stage of a complex process and typically contain many details. This complicates an intuitive understanding of the assurance claims provided by the proofs. Hence, it is important to make explicit which claims are actually proven, and on which assumptions and reasoning principles both the claims and the proofs rest. Moreover, the complexity of the tools used can lead to unforeseen interactions and thus causes additional concerns about the trustworthiness of the assurance claims.

Recent research to address the previously mentioned dilemma (traceability between code and proof) focus on showing that traceability between the proofs on one side and the certified program and the used tools on the other side is important to gain confidence in the formal certification process. Approaches are currently under development to systematically derive safety cases from information collected during the formal software safety certification phase, in particular the construction of the necessary logical annotations. The purpose of these safety cases is to provide a "structured reading guide" for the program and the safety proofs that will allow users to understand the safety claims without having to understand all the technical details of the formal machinery.

Fault tree analysis is an example of usage to identify possible risks to the program safety and the certification process, as well as their interaction logic, and thus to derive the structure of the safety cases. Generic, multi-tiered argument then gets used that is

instantiated with respect to a given safety property and program. Its three tiers together constitute a single safety case that justifies the safety of the program. The upper tier simply instantiates the notion of safety and the formal definitions for the given safety property while the two lower tiers argue the safety of the program as governed by the property. The lower tiers are constructed individually to reflect the program structure. This can be done systematically because their structure directly follows the course the annotation construction takes through the program. Model driven engineering champions allege that MDE principles and technologies mixed with formal methods drastically increase the existing level of automation within system development and cater for needed and demanded support for formal analysis. [34]

Because of its ability to address software complexity and productivity challenges, Model-based design has become the preferred software engineering paradigm for the development of application software components in central automotive domains such as chassis and powertrain. The core idea is that an initial executable graphical model representing the application software component to be developed serves as the primary representation throughout multiple phases of software development. The executable model is refined and augmented until it becomes a blueprint for the final implementation through production code generation. In addition, executable models can be utilized for various quality assurance activities.

The Simulink product family is a popular tool chain for Model-Based Design. Simulink and Stateflow support graphical modeling with time-based block diagrams and event-based state machines, and Real-Time Workshop Embedded Coder supports embedded code generation. In the recent past, Model-Based Design with code generation has been successfully employed to produce software for safety-critical applications. Examples include application software components of the electromechanical APA steering system [77] for the Volkswagen Tiguan, an urban SUV. Stringent software development methods and techniques are already required to satisfy customer expectations and ensure the essential quality and reliability of any in-vehicle software. [77] However, given the

safety-related nature of some advanced automotive systems, application of techniques above and beyond existing software development practices must be considered for these applications [77]. The requirements imposed by safety standards also have to be met, and the objectives and recommendations outlined therein need to be mapped onto Model-Based Design.

The software development activities for a driver assistance system at Carmeq were evaluated to allow rationalizing such a mapping. In practice, the evaluation of this recent project using Model-Based Design led to consolidated findings that became best practice and will be introduced into guidelines for future projects. In their research, the authors combine these project experiences with more general ideas on using Model-Based Design with Simulink and Stateflow for safety-related automotive applications. The safety standard currently relevant to automotive in vehicle applications is IEC 61508. Part 3 of this international standard, IEC 61508-3 [77], is concerned with software development. In IEC 61508, software failures are viewed as the result of faults systematically introduced during software development. In recognition of this, IEC 61508-3 defines requirements and constraints for the software development and quality assurance processes [77]. The degree of rigor required in these processes depends on the criticality of the software component within the embedded application and is expressed in terms of safety integrity level (SIL) [77].

3.2.3.3 Formal Approaches

Formal methods are perceived differently by industry and engineers, and there are many types of formal methods in software development. Formal technique involves the use of mathematically precise specification and design notations. In its native form, formal development is based on proof refinement to ensure software correctness at each stage in the software development life cycle. [26] Formal methods use mathematical models for analysis and verification at any part of the program life cycle. [27]

Formal methods are mathematical techniques that should be heavily supported by tools for developing software and hardware systems. Mathematical rigor enables users to

analyze and verify these models at any part of the program life-cycle: requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution. The vital first step in a high-quality software development process is requirements engineering. Formal methods can be useful in extracting, clarifying, and defining requirements. Tools can provide automated support needed for checking completeness, traceability, verifiability, and reusability, and for supporting requirements evolution, different viewpoints, and inconsistency of management. Formal methods are used in specifying software: developing a precise statement of what the software will do, while avoiding constraints on how it is to be achieved. Examples of these methods include ASM[72], B, and VDM [27].

Formal methods differ from other design systems through the use of formal verification schemes, the basic principles of the system must be proven correct before they are accepted [27]. Traditional system design has used extensive testing to verify behavior, but testing is capable of only finite conclusions. Dijkstra and others have demonstrated that tests can only show the situations where a system won't fail, but cannot say anything about the behavior of the system outside of the testing scenarios [27]. In contrast, in formal methods, once a theorem is proven true it remains true.

It is very important to note that formal verification does not cancel the need for testing [28]. Formal verification cannot resolve invalid assumptions within the design, but it can aid in identifying defects and in reasoning which would otherwise be left unverified. In several cases, engineers have reported finding flaws in systems once they reviewed their designs formally [28]. Roughly speaking, formal design can be seen as a three step process, following the outline given here [29][30]:

1. Formal Specification: The engineer defines the system using a modeling language. The language has some fixed grammar that allows modeling complicated structures of defined types within the specification.
2. Verification: As previously mentioned, formal methods contrast other specification systems through detailed focus on provability and correctness. By

building a system using a formal specification, the designer is actually developing a set of theorems about his system and proving these theorems are correct. The formal verification is not an easy process since mapping your system into a set of theorems that each has to be proved is complex and ends up resulting a huge number of theorems for small systems. Even a traditional mathematical proof is a complex matter. Given the demands of complexity and Moore's law, almost all formal systems use an automated theorem-proving tool of some form. These tools can prove simple theorems, verify the semantics of theorems, and provide assistance for verifying proofs that are more complicated.

3. Implementation: Once the model has been specified and verified, it is implemented by converting the specification into code. Many tools automatically map formal specifications into code. As the difference between software and hardware design grows narrower, formal methods for developing embedded systems have been developed

Formal methods are viewed with a certain degree of suspicion. While formal methods research has been progressing since 1960's, formal methods are only being slowly accepted by engineers. There are several reasons for this. Most formal systems are extremely descriptive and extensive / thorough, modeling languages have generally been judged by their capacity to model anything. Unfortunately, these same qualities make formal methods very difficult to use, especially for engineers that are not used to modeling a system in formal notations or trained on type theory which is needed for most formal systems[31]. Ultimately, formal methods will acquire some form of acceptance, but compromises will be made in both directions: formal methods will become simpler and formal methods training will become more common.

Formal methods are distinguished from other specification systems by their emphasis on correctness and proof, which is ultimately another measure of system integrity. Proof is a complement, not a replacement, for testing. Testing still remain a crucial part of guaranteeing any system's operability, but it is finite. Testing cannot show that a system operates properly; it can only show that the system works for some tested cases. Because

testing cannot demonstrate that the system should work outside the tested cases, formal proof is necessary [32].

Formally proving computer systems is not a new idea. Knuth and Dijkstra have written extensively on the topic, but their methods of proof still remains to be heavily based on the traditional mathematical methods. In pure sciences, proofs are verified through extensive peer review before publication. Such techniques are time-intensive and less than perfect; it is not unusual for a published proof to contain a flaw. Given the cost and time requirements of systems engineering, traditional proving techniques are not really applicable. [33] Because of the costs of hand verification, most formal methods use automated theorem proving systems to verify their designs.

There has been recent focus on using formal methods in the specifications stage. Specification is a technical agreement in writing between a software engineer and a client to ensure that both have a common understanding of the objectives of the software. The client uses the specification to guide application of the software; the software engineer uses it to guide its implementation. A complex specification may be broken down into sub-specifications, each describing a sub-component of the system, which may then be assigned to other programmers, so that a programmer at one level becomes a client at another [27]. Complex software systems require careful organization of the architectural structure of their components: a model of the system that hides implementation detail, allowing the architect to focus on the analyses and decisions that are most critical to structuring the system to satisfy its requirements [27]. Wright is an example of an architectural description language based on the formalization of the abstract behavior of architectural components and connectors [27].

The purpose of software safety certification is to show that a program complies with its high-level requirements and is safe in the presence of potential hazards. Formal software safety certification is based on formal techniques, which are based on program logics to show that the program does not violate certain constraints during its execution. Most

endeavors depend on creating a safety property, which is an exact characterization of these conditions, based on the operational semantics of the programming language. Each safety property thus describes a class of hazards. A safety policy is a set of first order logic rules (Could be using Hoare logic or other formal notations) designed to show that safe programs satisfy the safety property of interest. A safety predicate that is added to the computed verification conditions (VCs). However, the focus is on the information provided by constructing the annotations, and the details of constructing is left out (i.e., applying the Hoare rules) and proving (i.e., calling the theorem prover) the VCs to the complementary system-wide safety case. Formal software safety certification follows the same technical approach as program verification. A VC generator (VCG) traverses the code backwards and applies the Hoare rules to produce VCs, starting with any safety requirements on output variables [33].

It is required that all VCs are proven by an automated theorem prover (ATP). The figure below details the flow of software certification using formal methods and tags the trusted/untrusted components.[29]

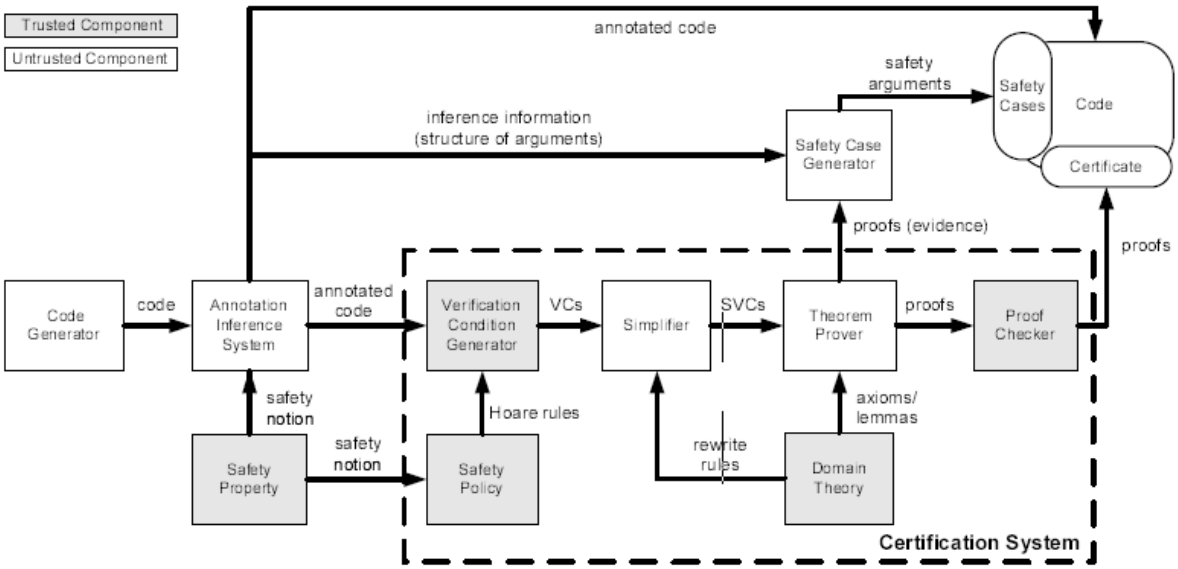

Figure 5 Software Certification using Formal Methods

Formal methods are heavily employed in software design. Data refinement is established based on state machine specification, abstraction functions, and simulation proofs, as its central role in methods like VDM [27], and in program refinement calculi [27]. At the implementation level. Formal methods are also utilized for code verification. Every program-specification pair implicitly asserts a correctness theorem that, if certain conditions are satisfied, the program will achieve the effect described by its specification. Code verification is the attempt to prove this theorem, or at least to find out why the theorem fails to hold. The inductive assertion method of program verification was invented by Floyd and Hoare [29], and involves annotating the program with mathematical assertions, which are relations that hold between the program variables and the initial input values, each time control reaches a particular point in the program. Code can also be generated automatically from formal models; examples include the B-method [27] and SCADE [27]. Formal methods are used in software maintenance [27] and evolution [27]. Perhaps the widest application of formal methods is in the maintenance of legacy code: in some of Microsoft's most successful products, every tenth line is an assertion.

A Formal method has to have a formal notation, semantics based on mathematics, and formal deductive system. Given those requirements, no existing method is truly a formal method. But there are many that are close. Some have mathematical semantics (sometimes partial) but almost no deductive system, such as Z and State charts which are named as conceptual techniques. Others have logic but almost no semantics, such as VDM and Unity; these get labeled as deductive techniques. Still others are defined by an evaluation mechanism (operational semantics or evaluation rules) and are executable specifications. However, executability is not the main drive of formal methods and in some cases gets in the way of their use. Researchers tend to distinguish between state-oriented (e.g. Z and VDM) and behavior-oriented (e.g. Lotos, Unity, RSL) techniques. So it is clear that all formal methods are not created equal, and it is misleading to group all such methods together to decide if formal methods make a positive difference to a software project.[26]

For some organizations, the changes in software development practices needed to support such techniques can be revolutionary. That is, there is not always a simple migration path from current practice to migrate to the use of formal methods, because the effective use of formal methods can require a drastic change right at the beginning of the traditional software lifecycle: how we capture and record customer requirements. Thus, the stakes in this area can be particularly high. For this reason, strong evidence of the effectiveness of formal methods is highly desirable.[26]

Unfortunately, past evaluations of the use of formal methods were not conclusive. The few serious industrial uses of formal methods focused on formal specification alone, with no widespread attempt at formal deduction, refinement or proof [26].

Some researchers report that the use of formal notations does not lead inevitably to improving the quality of specifications, even when used by the most mathematically sophisticated minds. In experiments, the use of a formal notation are claimed to lead to a greater number of defects, rather than fewer. [26]

Meanwhile, evidence of the positive effects of formal methods continues to grow. Some researchers described several instances of their use for safety-critical systems in early 1994 on a joint project between IBM Hursley and the Programming Research Group at Oxford University. A serious attempt was made to quantify the benefits of using Z on the CICS re-specification project, and a proceedings paper provides sanitized graphs and general information. As a result, CICS provided a very good quantitative evidence to support the efficacy of formal methods. However, the public announcements of success have never been accompanied by a complete set of data and analysis, so independent assessment is difficult. [26]

As unscientific support for formal methods has grown, industry has been more willing to use formal methods on projects where the software is safety-critical. Formal methods are

being incorporated into standards and imposed on developers. For instance, the interim UK defense standard for such systems, DefStd 00-55, makes mandatory the use of formal methods. ISO-26262, standard for automotive reliability, also imposes the use of formal methods for software that could lead to safety hazards. Researchers believe such standards formulation without a solid basis of empirical evidence can be dangerous and costly as there is still no hard evidence to show that:

1. Formal methods usage is cost effective and has been used on a real safety project complex project.
2. Using Formal methods increases the reliability and makes the project more cost-effective compared to traditional structured methods with enhanced testing
3. Either developers or users can ever be trained in sufficient numbers to make proper use of formal methods

Moreover, it must be understood how to choose among the many competing formal methods, which may not be equally effective in a given situation [26].

3.2.3.3.1 Formal Methods in Architectural design

As shown in previous sections, formal methods are proposed is ISO26262 and is potentially the only available methodology that could really help in architectural design level safety certification. Formal methods for software development receives much attention in research centers, but are rarely used in industry for the development of (large) software systems. Several reasons contribute to this state:

1- Entry cost to FM is huge (Education, legacy methods migration … etc)
2- Insufficient tool support for FM based rules as most of them are academic based tools as oppose to industrial ones
3- Lack of expertise/training to FM
4- FM notations and flow are hard to understand/adopt by non-mathematicians.

On the other hand, Semi formal methods are widely used in the industry due to many reasons mainly due to MDE (Model Driven Engineering) approaches which encourages:

1- Focus on creating models of a system at each stage in the development lifecycle

2- Automatic model transformations (e.g. to code)
3- Intuitive, and abstract graphical notations
4- Good at abstracting away detail

With the above said, the usage of semi-formal methods (informally defined semantics) cause:

1- Ambiguity
2- Inconsistency
3- Imprecision
4- Unable to be formally reasoned about methodologies

Formal methods can be used in conjunction with informal or semi-formal modeling techniques in software development. In such integrated approaches, formal techniques provide an effective means to check the validity of semi-formal models, thus providing increased quality for both models and implementation. Despite its potential, application of the integrated approach to large scale systems has been limited.

3.2.3.3.2 Benefits of Formal Methods

Formal methods offer additional benefits outside of provability, and these benefits do deserve some mention. However, most of these benefits are available from other systems, and usually without the steep learning curve that formal methods require.

Discipline: By virtue of their rigor, formal systems require an engineer to think out his design in a more thorough fashion. In particular, a formal proof of correctness is going to require a rigorous specification of goals, not just operation. This thorough approach can help identify faulty reasoning far earlier than in traditional design[32][33]

The discipline involved in formal specification has proved useful even on already existing systems.[35]

Precision: Traditionally, disciplines have moved into jargons and formal notation as the weaknesses of natural language descriptions become more glaringly obvious. There is no

reason that systems engineering should differ, and there are several formal methods which are used almost exclusively for notation.[28]

For engineers designing safety-critical systems, the benefits of formal methods lie in their clarity. Unlike many other design approaches, the formal verification requires very clearly defined goals and approaches. In a safety critical system, ambiguity can be extremely dangerous, and one of the primary benefits of the formal approach is the elimination of ambiguity [32].

3.2.3.3.3 Weaknesses of Formal Methods

Bowen points out that formal method is generally viewed with suspicion by the professional engineering community, and the propensity of tentative case studies and advocacy papers for the formal approach would seem to support his thesis [28]. There are several reasons why formal methods are not used as much as they might be, most stemming from overreaching on the part of formal methods advocates.

Expense: Because of the rigor involved, formal methods are always going to be more expensive than traditional approaches to engineering. However, given that software cost estimation is more of an art than a science, it is debatable exactly how much more expensive formal verification is. In general, formal methods involve a large initial cost followed by less consumption as the project progresses; this is a reverse from the normal cost model for software development [31].

Limits of Computational Models: While not a universal problem, most formal methods introduce some form of computational model, usually hamstringing the operations allowed in order to make the notation elegant and the system provable. Unfortunately, these design limitations are usually considered intolerable from a developer's perspective.

Usability: Traditionally, formal methods have been judged on the richness of their descriptive model. That is, 'good' formal methods have described a wide variety of systems, and 'bad' formal methods have been limited in their descriptive capacities. While an all-encompassing formal description is attractive from a theoretical perspective, it

invariably involved developing an incredibly complex and nuanced description language, which returns to the difficulties of natural language. Case studies of full formal methods often acknowledge the need for a less all-encompassing approach [29].

Arguably, many of these failures can be attributed to overreaching on the part of formal methods advocates. This reasoning has led to the lightweight approach to formal specification.

While formal systems are attractive in theory, their practical implementations are somewhat wanting. By attempting to describe all of any system, formal methods have overreached, and generally failed.

3.2.3.3.4 Formal Model Checking of UML State chart Diagrams

UML has become a defacto standard for software industries. AUTOSAR specifications are primarily depending on UML diagrams in its specifications. This, among others, was the main reason why several research endeavors focused on model checking of UML diagrams, namely state charts and sequence diagrams[84].

In [39], a UML state chart system based model is translated into π-calculus. The intermediate π-calculus model is then translated into NuSMV input language based on defined translation rules. NuSMV model checking is then run to evaluate any incompliances or problems in the model. This is a 2 step translation process and it does not show how the UML model developer will interpret the feedback in UML domain.

In [37], the authors translate UML state charts into FSMs (Finite State Machines), FSMs are then transformed into NuSMV input model and NuSMV model checking is finally run on the 2nd level translated model.

In [40], the authors translate UML models to an input language in a self-developed model checker called PAT in such a way that is transparent to users. In particular this approach

utilized PAT as the back end for verification capabilities. PAT is claimed to support several modeling languages including CSP#. The authors tool flow is based on parsing UML XMI (XML metadata Interchange), the object management group standard of exchanging UML diagrams. The authors claim that PAT can address deadlock, reachability, trace refinement relationship. The paper presents the framework but fails to apply the theory to any industrial use-case. It also fails to show how the UML user will get the model checking feedback to reason with it in UML domain.

Similar to [37], authors in [38] translate abstracted UML state chart railway interlocking system model into FSM which is then translated into NuSMV input language and utilized NuSMV checker. Again the work does not show how any counterexamples can be expressed back in UML domain. It also lacks any documentation on how the safety properties are constructed and translated into LTL formulas.

Similarly, [41], transforms UML verification model to PROMELA model which uses hierarchical automata to describe the state machine and its formal semantics and then verifies the correctness of the model using SPIN since SPIN accepts PROMELA based models. The same drawbacks discussed in previous endeavors are also applicable to this effort.

In [98], an approach to formalize UML is shown via transforming UML to Event-B. The transformation only covers UML activity diagram to Event-B models and does not cover state flow diagrams.

In [99], an approach is presented that semi-automatically generates formal specifications from state machine and activity diagrams. The model is translated to text using MERL language amd MetaEdit tool. State machine is transformed into SMV model description and activity diagrams into LTL formulas. NuSMV model checker is then used to verify the specification.

In[100], a method is proposed to map UML state chart to BIR language, which is designed for BOGOR model checking in order to only evaluate the deadlock property.

In [101], Echo verification is employed where the system under test has to be captured with a PVS based formal specification including low level specification capturing pre and post conditions. Additionally, the proof is semi-automated where the complete proof needs to be done under human guidance.

Earlier attempts did not consider minimizing transformation steps due to ISO 26262 tool qualifications recommendations and they address limited category of defects. They also either propose a new low level specification language, limited to architecture as opposed to functional mapping or lack showing how the model checker result can be interpreted via a UML designer. Additionally, the existing endeavors were not evaluated based on an industrial specification that was compared to an industrial implementation of the case study module. Our proposed framework attempts to address these shortcomings

3.2.4 Comparative Analysis of Existing V&V Methods

Table 4 summarizes the existing V&V methods.

Table 4 Summary of V&V Techniques

Static	Dynamic	Formal
• Cause-Effect Graph • Control Analysis • Data Analysis • Interface Analysis • Semantic Analysis • Structural Analysis • Symbolic Evaluation • Syntax Analysis • Traceability Assessment	• Acceptance Testing • Bottom-up testing • Comparison Testing • Compliance testing • Debugging • Execution Testing • Fault insertion testing	• Induction • Inference • Logical Deduction • Proof of correctness . .

Static	Dynamic	Formal
. .	• Functional black box testing • Interface Testing • Boundary Value • Equivalence partitioning • Structural Testing – White box	

Static Analysis in general, aims to identify programming defects and is limited in identifying these category of defects. Existing standards (e.g. ISO26262) mandate the use of static analysis tools as part of the software development life cycle [13]. It is commonly known that static analysis covers only a subset of software programming defects, is usually language and domain specific, and usually produces false defects and sometimes coding limitations that could introduce an implementation maneuver to conform to a defined static rule. Safety standards mandate the use of these tools on all software that needs to be safe. The reason being that all approaches that identify defects or good practices should be integrated into one approach (Standard) to ensure safety. It is certain that static analysis tools do not cover all software defects presented in section 3.1, for example, timing and interface defects and static analysis is not capable to address such category of defects since they are manifested as a run-time behavior defects while static analysis focuses on defects that are outside program execution. In conclusion, static analysis helps in identifying defects in a timely fashion (if compared to manual code inspections) but no software safety could be concluded on software if it claims that it is 100% static analysis bug free software.

Even if static techniques are necessary to detect defects earlier in the development process, they are not sufficient. In fact, these techniques focus on analyzing the static product representation and do not test the product in its real life (dynamic).

Testing implies executing the program on (valued) inputs. Since static techniques (review, inspection …) are useful to evaluate the internal correctness of a software product, testing is the used technique allowing the assessment of its behavior. Even for simple programs, so many test cases are theoretically possible that exhaustive testing would require years to execute. Dijkstra (Dijkstra 1972) calculated that the exhaustive testing of a multiplier of two 27-bit integers taking "only" some tens of microseconds for a single multiplication would require more than 10000 years to be completely tested. Exhaustive testing is a NP-Complete problem from a computational viewpoint. Generally, the whole test set can be considered infinite. In contrast, the number of executions that can realistically be observed must obviously be finite (and affordable). Clearly, "enough" testing to get reasonable assurance of acceptable behavior is a basic need. This basic need points to 2 well-known issues of testing, both technical in nature (criteria for deciding to stop testing) and managerial in nature (estimating the effort to put in testing). Testing always implies a trade-off between limited resources and schedules, and inherently unlimited test requirements.

Formal methods are rarely used in automotive industry, contrary to medical, avionics and railways industries. The main argument of automotive industry managers was the high cost of deploying and using formal methods. As automotive electronic products becomes more and more complex, automotive industry is required to start adapting existing formal methods to their context or developing new ones. Actually, the cost of non-quality (warranty and customer dissatisfaction) exceeds the cost of using formal methods. Now, in automotive industry, semi-formal and formal methods are highly recommended via standards (ISO-26262) to ensure software reliability. Incompleteness and ambiguity are the main characteristics of informal and semi-formal methods. The use of formal specification methods is expected to lead to increased software quality and reliability.

A variety of advantages has been attributed to the use of formal software specifications. These advantages include understanding of specifications, help in the verification of

specifications and automatic generation of the source code and test cases. Management is generally conservative and unwilling to use new techniques whose benefits are not yet established. Given these difficulties in using formal methods, challenges remain in integrating formal methods with the system development existing paths.

Chapter 4. Proposed Approach

The main theme of the proposed framework is to introduce a solution that allows early detection of design bugs via formal verification. Such a framework needs to address existing challenges that discouraged the industry from moving to utilize formal verification and still heavily relying on testing. In order for the framework to address existing shortcomings or challenges, it needs to meet the below criteria:

1- The ability to capture detailed design using xtUML (Executable UML)

AUTOSAR specifications are based on informal notation (English text) and Unified Modelling Language (UML) diagrams (state machine, sequence diagrams ...). The use of informal notation to capture specification has caused ambiguities in the specification that led up to several releases of AUTOSAR standard to clarify such ambiguities. The framework should support capturing conditions in the model that serve as a foundation for generating theorems in formal domain which forces the model designer to question any ambiguities in the specification. Additionally, ISO-26262 guidelines highly recommend using semi-formal notation for capturing the design as shown in Table 1. Based on the above, the framework supports modelling the software in UML extended with behavior to ensure the possibility of exhaustive design verification.

2- Automatic mapping of UML to formal notations

Once the software is modeled in UML, the framework supports automatic translation from UML to formal notation and theorems in formal domain. The objective is to ensure that the framework addresses formal complexities that discouraged the industry from using them as discussed in 3.2.3.3.3.

3- Extend xtUML with Satisfiability conditions

The framework supports capturing specification requirements in UML model so that it serves as the baseline for generating formal theorems, forces the model designer to question any specification ambiguities and adds a separation

between the design elements and requirements to ensure that theorem generation is separate from deign implementation.

4- Formal verification of semi-formal model

   Framework supports model checkers that ensure the design is correct and incompliances to specification are detected. The ability of the framework to check the model formally will allow software suppliers to be ISO-26262 compliant with unit design and implementation verification guidelines as presented in Table 2. Additionally, formal model checking will ensure design requirements are mathematically exhaustively proven as opposed to depending on test cases to verify compliance to specification.

5- Integrate the flow in a well-established xtUML tool.

   The framework integrates to existing UML model IDE which allows rapid proof of concept and integration to existing verification activities.

Our proposed framework is based on several components, namely, a formal framework called SAL (Symbolic Analysis Laboratory)78] , BridgePoint which is an executable UML (Unified Modelling language) integrated development environment (IDE) [96], UML to SAL model compiler to compile UML model and requirements to formal SAL notation, and finally model checkers that validate the generated SAL model against generated theorems. In this chapter, we will introduce the framework flow followed by a brief introduction on SAL, UML and BridgePoint IDE. The framework allows software designers to formally verify a specified software in a semi-formal notation (UML). This complies with ISO 26262 design verification guidelines for ASILs (Automotive Safety Integrity Level) C and D which highly recommend semi-formal verification of the design for ASILs C and D.

4.1 Design Flow

Design flow is initiated by a designer that starts with informal/semi-formal specification document. The designer maps the specification to a UML design augmented with action language to capture behavior. EXecutable UML - xtUML model augmented with satisfiability conditions (Requirements) is the framework starting input. Satisfiability conditions represent requirements that the design should satisfy as captured in a requirement specification. Satisfiability conditions are the

foundation for generating the formal theorems which are used to verify the design. Satisfiability conditions can be captured on variable, state, and transition levels in UML.

xtUML input model which includes satisfiability conditions is fed into a model compiler which parses the UML model elements presented in XML format and constructs object instances of all elements. The objects are traversed and mapped into a SAL formal model based on transformation rules. SAL objects are also stored and linked to their UML counterparts. Executable UML - xtUML model is mapped into a formal SAL model and the UML satisfiability conditions are mapped into SAL formal theorems.

SAL checkers get launched and any generated counterexample is mapped back into UML domain so that the UML designer could fix the detected specification incompliance in UML domain. Figure 6 summarizes the proposed flow. The process is iterative until all theorems can be properly proved.

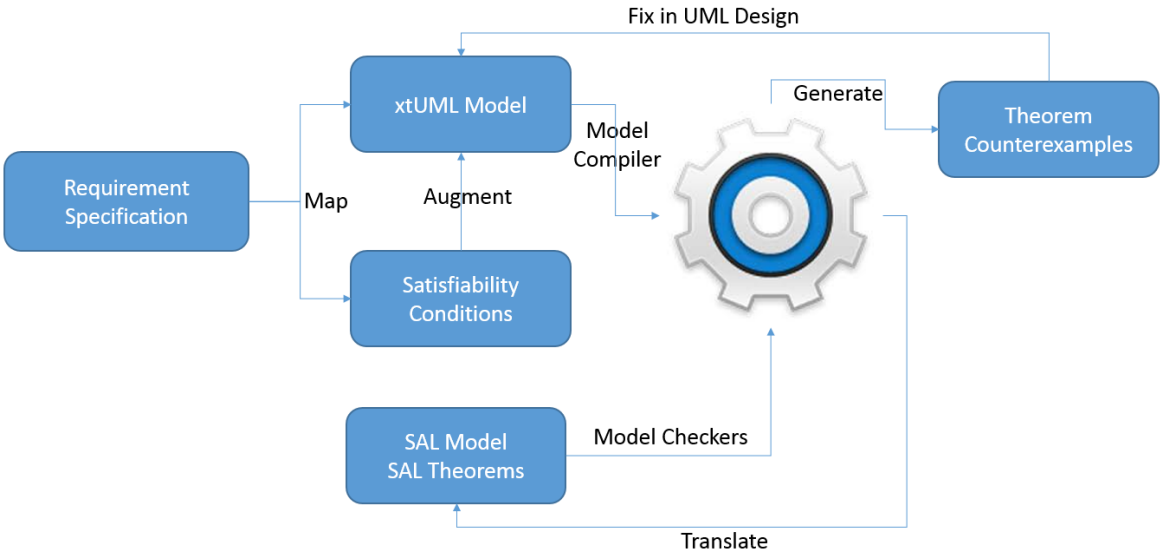

Figure 6 Proposed Framework Workflow

As shown in Figure 6, xtUML (eXecutable Translatable UML) model implementation is initially done based on the software specification in BridgePoint xtUML IDE[96].

The xtUML model contains the de-facto UML standard diagrams and elements including but not limited to component(s), classes(s), state machine(s), transition(s), states, abstract object language to capture behavior within the model, data type(s), operation(s), attribute(s) etc.

The model also encapsulates the requirements captured as satisfiability conditions to trace existing design elements to the original requirements in the software specification. Once the model is complete, a manual build command is triggered from the IDE which automatically triggers the model compiler. BridgePoint enables the creation of a custom model compiler that traverses all UML model elements and generates new model based on extendable implementation.

We have created a custom model compiler that generates SAL model from the xtUML model. The model compiler developed component compiles the xtUML model to generate a formal SAL model and LTL (Linear Temporal Logic) based theorems. Model checkers are manually executed to identify model violations. Examples of checkers include but are not limited to deadlock checker to detect any deadlock in the model. Model checkers are manually triggered on the generated SAL model for each generated theorem. The execution reports any model violation (counter example). Any reported counter example can be analyzed by the designer to trigger xtUML model fix/refinement to address the generated counter example.

Our UML model extensions – satisfiability conditions aim to address the ISO-26262 test case derivation basis as shown in Table 5. (‘++’ indicates that the method is highly recommended for the identified ASIL, ‘+’ indicates that the method is recommended for identified ASIL, ‘o’ means no recommendation)

Table 5 Methods for Deriving Test Cases for Software Unit Testing in ISO-26262

Methods		ASIL			
		A	B	C	D
1a	Analysis of Requirements	++	++	++	++
1b	Generation and analysis of equivalence classes	+	++	++	++
1c	Analysis of boundary values	+	++	++	++
1d	Error Guessing	+	+	+	+

Satisfiability conditions are captured in xtUML to enable generation of theorems to address the above methods. Variable satisfiability conditions (Upper and Lower limit) generate theorems to cover boundary value analysis and equivalence classes. State satisfiability conditions capture conditions to ensure requirement compliance of variables in a given state in the state machine. Transition satisfiability conditions capture conditions to ensure requirement compliance of variables in a given transition in the state machine. Our work / research supports the above methods yet the framework can be extended to cover other methods to verify the design.

4.2 Input Model – xtUML

Requirement Specification document is initially mapped to an xtUML model design implementation. The requirements are mapped into UML packages, components, classes (attributes and operations), and state machines. All defined data types, attributes, functions are defined in the UML model. Once UML model is complete, the model captures architectural design of the specification. OAL(Object Action Language) is now embedded in states, transitions, operations (Instance or class

based), ports, mathematically derived attributes, and functions to capture the specification behavior.

Table 6 shows xtUML diagrams, purpose and usage of each as used within our framework and case study modules.

Table 6 UML Model Diagrams

UML Diagram	Purpose	Usage
Class Diagram	A UML class diagram is not only used to describe the object and information structures in an application, but also show the communication with its users. It provides a wide range of usages; from modeling the static view of an application to describing responsibilities for a system. Composition is a special type of aggregation that denotes a strong ownership.	In a UML class diagram, classes represent an abstraction of entities with common characteristics. Associations represent static relationships between classes. Aggregation is a special type of association in which objects are assembled or configured together to create a more complex object. Generalization is a relationship in which one model element (the child) is based on another model element (the parent). Dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier.
Component Diagram	It allows application designers to verify that a system's required functionality is being implemented by	The UML component diagram doesn't require many notations, thus very easy to draw and requires only two symbols:

UML Diagram	Purpose	Usage
	components, thus ensuring that the final system will be acceptable. Component diagram is a useful communication tool among stakeholders to discuss, analyze or improve a system design.	component and dependency.
State Chart	Statechart diagrams are used to model dynamic nature of a system. They describe all of the possible states of an object as events occur. So the most important purpose of Statechart diagram is to model life time of an object from creation to termination.	A state is a condition during the life of an object during which it satisfies some condition, performs some activity, or waits for some external event. A start state is the state that a new object will be in immediately following its creation. An end state is a state that represents the object going out of existence. A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state, when a specified set of events and conditions are satisfied.

UML Diagram	Purpose	Usage
Package Diagram	Package diagrams are used to organize the elements of a model. They are typically used to depict the high-level organization of a software project. Package diagram can show both structure and dependencies between sub-systems or modules. They can be used to group any construct in the UML such as classes, actors, and use cases.	The package element in UML is represented by a folder icon. Each package represents a namespace. Packages can also be members of other packages, providing for a hierarchic structure in which top-level packages are broken down into sub-packages.

4.3 UML Satisfiability Conditions

We have extended the xtUML model to capture satisfiability conditions on the state, transition, operations and variables in UML model. The Software specification requirements from specification documents are mapped into satisfiability conditions in the UML design. Currently, the conditions are captured as descriptions on the UML element (State, transition, class operation, class attribute etc.). The description captures the conditions that should be satisfied as a function of attributes, states, and/or values given a state, transition or operation.

These conditions map to requirements and serve as the baseline for formal theorem generation via the model compiler. The model compiler maps the condition to a LTL (Linear temporal Logic) rule in SAL Language.

4.3.1 State Level Conditions

A state models a situation during which some invariant condition holds. State is the main entity of a state machine where state changes are driven by events. A state is a condition of being at a certain time. It is also a point in the lifecycle of a model element that satisfies some condition, where some particular action is being performed or where some event is being monitored. States can trigger actions.

Every state in a UML state chart can have optional entry actions, which are executed upon entry to a state, as well as optional exit actions, which are executed upon exit from a state. Entry and exit actions are associated with states, not transitions. Regardless of how a state is entered or exited, all its entry and exit actions will be executed. Because of this characteristic, state charts behave like Moore automata. Because entry actions are executed automatically whenever an associated state is entered, they often determine the conditions of operation or the identity of the state.

Specifications always include state entry conditions, exist conditions and state actions in informal notation or semi-formal notation. Informal text to define state conditions from AUTOSAR specification of WatchDog Manager are shown in Figure 7 and Figure 8.

[WDGM201] ⌈If all values in three sets of results of Supervision (results of Alive Supervision, results of Deadline Supervision, results of Logical Supervision) for the *Supervised Entity* are `correct` and the *Supervised Entity* was in *Local Supervision Status* `WDGM_LOCAL_STATUS_OK`, then the function `WdgM_MainFunction` shall leave the *Supervised Entity* in the *Local Supervision Status* `WDGM_LOCAL_STATUS_OK` (see Transition (1) in Figure 3). ⌋ ()

Figure 7 AUTOSAR Watchdog Manager Informal State Details – 1

[WDGM202] ⌈If the *Supervised Entity* was in *Local Supervision Status* `WDGM_LOCAL_STATUS_OK` AND:

1. (At least one result of Alive Supervision of the *Supervised Entity* is `incorrect` and a Failure Tolerance of zero is configured (see configuration parameter `WdgMFailedAliveSupervisionRefCycleTol` [WDGM327_Conf]) OR
2. If the result of at least one Deadline Supervision of the *Supervised Entity* or the result of at least one Logical supervision of the *Supervised Entity* is `incorrect`),

THEN the function `WdgM_MainFunction` shall change the *Local Supervision Status* to `WDGM_LOCAL_STATUS_EXPIRED` (see Transition (2) in Figure 3).

Figure 8 AUTOSAR Watchdog Manager Informal State Details – 2

Figure 7 and Figure 8 show how informal text in specification document can represent the state entry conditions and exit conditions. In our research, we count on mapping the above informal text into state level satisfiability conditions. For example, a satisfiability condition on state 'WDGM_LOCAL_STATUS_EXPIRED' could be defined as: at least one alive supervision entity (in UML design notation) is incorrect, a zero fault tolerance OR at least one deadline supervision of a supervised entity is incorrect OR at least one logical supervision entity of a supervised entity is incorrect. This will ensure that any design defect that leads to being in 'WDGM_LOCAL_STATUS_EXPIRED' while the informal conditions (that are mapped in UML) are not true as a result of a design bug can be detected in the design verification as opposed to the code level stage.

Table 5 Methods for Deriving Test Cases for Software Unit Testing in ISO-26262, include Analysis of requirements. We consider that state level conditions is based on deriving a test case (theorem) to ensure that requirement 202 in Figure 8 is being adhered to in the design based on analysis of requirements guidelines in ISO-26262.

4.3.2 Transition Level Conditions

A transition is a relationship between a source state and a target state. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type. A Transition is the movement from one state to another state. Transitions between states occur as follows:

1. An element is in a source state
2. An event occurs
3. An action is performed
4. The element enters a target state

[WDGM205] ⌈If the *Supervised Entity* was in *Local Supervision Status* `WDGM_LOCAL_STATUS_FAILED` AND:

1. (If all the results of Alive Supervision of the *Supervised Entity* are `correct` and the *counter for failed supervision reference cycles* equals 1) AND
2. If all the results of Deadline Supervisions of the *Supervised Entity* and all the results of Logical supervision of the *Supervised Entity* are `correct`),

THEN the function `WdgM_MainFunction` shall change the *Local Supervision Status* to `WDGM_LOCAL_STATUS_OK` and decrement the *counter for failed supervision reference cycles* (see Transition (5) in Figure 3). ⌋ ()

Figure 9 AUTOSAR Watchdog Manager Informal Transition Description

Figure 9 shows how informal text in AUTOSAR Watchdog specification document can represent transition conditions. In our research, we count on mapping the above informal text into transition conditions. For example, a satisfiability condition on transition 5 could be defined to match the specification transition conditions. This condition shall ensure that the transition goes to the correct target state and all transition action outcomes are correctly set (e.g. failed supervision reference cycle is decremented) (in UML design notation). This will ensure that any design defect that leads to violations against the requirement in transition can be detected in the design verification as opposed to the code level stage. Table 5 Methods for Deriving Test Cases for Software Unit Testing in ISO-26262 include analysis of requirements. We consider that transition level conditions is based on deriving a test case (theorem) to ensure that requirement 205 in Figure 9 is being adhered to in the design based on analysis of requirements guidelines in ISO-26262.

4.3.3 Variable Level Condition

Table 5 Methods for Deriving Test Cases for Software Unit Testing in ISO-26262 include boundary analysis and equivalence partitioning in deriving test cases. Variable level conditions aim to make sure that the design complies with any

requirements in this category. The variable level condition can capture low bound, high bound or that a certain value can only happen given a set of possible values in other variables.

SWS Item	WDGM329_Conf :		
Name	WdgMExpiredSupervisionCycleTol {WDGM_EXPIRED_SUPERVISION_CYCLE_TOLERANCE}		
Description	This parameter shall be used to define a value that fixes the amount of expired supervision cycles for how long the blocking of watchdog triggering shall be postponed, AFTER THE GLOBAL SUPERVISION STATUS HAS REACHED THE STATE EXPIRED.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 65535		
Default value	--		
ConfigurationClass	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency	scope: ECU		

SWS Item	WDGM307_Conf :		
Name	WdgMModeId		
Description	This parameter fixes the identfier for the mode. This identifier is for instance passed as a parameter to the WdgM_SetMode service.		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	0 .. 255		
Default value	--		
ConfigurationClass	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	--	
	Post-build time	X	VARIANT-POST-BUILD
Scope / Dependency			

SWS Item	WDGM330_Conf :
Name	WdgMSupervisionCycle {WDGM_SUPERVISION_CYCLE}
Description	This parameter defines the schedule period of the main function WdgM_MainFunction. Unit: [s]

Figure 10 Specification Level Boundary Value Requirements

Figure 10 shows how the specification mandates upper/lower range limit for variables in the system. This is captured in our proposed framework via defining variable level conditions to ensure that at no point in the system, the requirement of range for variables is violated. Any such violation shall be detected by the model checker.

4.4 UML to SAL Model Compiler

Once software is modelled in UML and specification requirements are captured as satisfiability conditions, the next step is to launch the model compiler. The model compiler is based on mapping UML notation to SAL notation automatically. This section includes SAL notation subsection, UML notation subsection and finally mapping rules to map SAL to UML subsection.

4.4.1 SAL

SAL (Symbolic Analysis Laboratory) is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. A key part of the SAL framework is an intermediate language for describing transition systems. This language is intended to serve as the target for translators that extract the transition system description for other modeling and programming languages, and as a common source for driving different analysis tools [78].

The SAL intermediate language is a basic transition system language. SAL describes transition systems in terms of initialization and transition commands. The current generation of SAL tools comprises a collection of state of the art LTL model checkers and auxiliary tools based on them.

4.4.1.1 SAL Language

As discussed previously, SAL is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. A key part of the SAL framework is a language for describing transition systems. This language serves as a specification language and as the target for translators that extract the transition system description for popular programming languages. The language also serves as a common source for driving different analysis tools through translators from the SAL language to the input format for the tools, and from the output of these tools back to the SAL language. The basic high-level requirements on the SAL language are :

1. Generality: It supports capturing the transition semantics of a wide variety of source languages.
2. Minimality: The language does not have redundant or extraneous features that add complexity to the analysis. The language captures transition system behavior without any complicated control structures.

3. Semantic Regularity: The semantics of the language is straightforward so that it is easy to verify the correctness of the various translations with respect to linear and branching time semantics. The semantics is definable in a formal logic.
4. Language Modularity: The language is parametric with respect to orthogonal features such as the type/expression sublanguage, the transition sublanguage, and the module sublanguage.
5. Compositionality: The language has a way of defining transition system modules that can be composed in a meaningful way. Properties of systems composed from modules can then be derived from the individual module properties.
• Synchronous composition: In this form of composition, modules react to inputs synchronously or in zero time, as with combinational circuitry in hardware. In order to achieve semantic hygiene, causal loops arising in such synchronous interactions get eliminated. The constraints on the language for the elimination of causal loops is not onerous as to rule out sensible specifications.
• Asynchronous composition: Modules that are driven by independent clocks are modeled by means of interleaving the atomic transitions of the individual modules.

SAL language is divided into type system, expression language, transition language, modules, synchronous and asynchronous composition of modules and the specification of systems. Language syntax details are elaborated in APPENDIX A.

4.4.2 AUTOSAR in UML

Object oriented system design method has been widely adopted, and the Unified Modeling Language (UML) has been recognized as standard modeling tool in object oriented design [95]. UML provides schematic modeling diagrams to describe the structure and behavior of target applications. There are nine modeling diagrams, five of them for system behavior description, and another four of them for system structure description.

Behavior modeling diagrams are use-case diagram, sequence diagram, collaboration diagram, state chart diagram and activity diagram. Structural diagrams are class diagram, object diagram, component diagram and deployment diagram. An additional

package diagram provides a general mechanism to organize system elements into groups. Use case diagram describe the scenario in the usage of system from a specific aspect. Sequence diagram focuses on time ordered messages that passed among related objects in a system to accomplish specific system function requirement. Collaboration diagram is another presentation of system scenario of object interactions that show the objects interconnection through messages. State chart depicts the system states and the state transitions. Vehicles are integrating more and more electronics parts to cope with stringent control and safety regulations, to increase the system performance and driving comfort [95].

It is worth highlighting that ISO26262 highly recommends using semi-formal notation to capture design in high ASIL levels since it forces the designer to address informal notation ambiguities that ends up generating design level bugs and incompliance to requirements.

AUTOSAR as an emerging architectural modeling standard in the automotive domain is increasingly spreading into the broad industrial practice. It is a great chance to establish explicit specifications of software systems' architectures with various benefits such as distributed development and in particular a completely model based development process, reaching even to the final source code. AUTOSAR architecture models are lacking information of interest (behavioral aspects), and AUTOSAR does neither address nor guarantee a transition from architecture into detailed design or implementation.

AUTOSAR architectures are currently augmented with UML to add currently missing expressiveness (interaction behavior) and how a seamless transition from AUTOSAR/UML architectural models to detailed design and succeeding implementation can be achieved [94].

As demonstrated by AUTOSAR itself (in terms of the Basic Software), UML could also be used to document the behavior of components, using state charts, sequence diagrams and other means. As UML defines behavior on top of structure, structural

concepts must however exist in order to achieve this. Most of the required structural AUTOSAR concepts are directly representable in UML (components, ports, connectors, interfaces).

In a model-based setting, the idea is to use models to semi-formally specify the detailed design, which can then in turn be used as a means for documentation, similar to as it was outlined for the Basic Software modules. However, there is a major difference between Basic Software modules and software components with respect to the level of abstraction that is used, which is higher for the latter. As such, in order to use documentation models for the detailed design of software components, these models must suit the respective level of abstraction. The relation between all employed detailed design constructs and those elements that are directly derived from the AUTOSAR architecture has to remain traceable.

UML can help to achieve this. If the architectural information of a software component is already available in UML, then both, structural and behavioral diagrams can be used for the specification of its detailed design. However, this detailed design model will not be able to correspond so closely to code structures as the Basic Software UML model, because for all architectural level elements the abstraction level needs to be preserved in order to achieve above mentioned traceability. That is, while the data elements of a sender/receiver interface will be ultimately mapped to corresponding macros/functions in the application header of the software component, using this representation in the detailed design model would clutter the model and make it hard to read.

Performing such a transition manually would furthermore be an error-prone and tedious task. Instead, as the mapping of these architectural concepts to the source code constructs is well defined, this transition can be left to code generators. Dependent on the completeness of the model, large parts or even the entire implementation may be generated from the detailed design model. In the end, the concrete modeling conventions therefore depend on what is to be generated and what code generator is being used. Behavior diagrams could be used in addition to specify the behavior of

the modeled functions, so that a code generator could also generate function definitions with implementation bodies corresponding to the modeled behavior.

We have selected BridgePoint as UML IDE [96] that is used in our framework. In BridgePoint, the architectural design can evolve into detailed design using the action language which captures behavior. Once the UML model is detailed with behavior and action, the model compiler/code generator steps can be used to generate target models or source code. Next section presents BridgePoint.

4.4.2.1 BridgePoint xtUML

Executable Translatable Unified Modeling Language [96] is a modeling dialect that employs standard UML notation to express executable models following the Shlaer Mellor Method of Object Oriented Analysis and Design [97]. The method is well-defined and documented and carries a substantial base of research, education and industry usage through the last two decades. xtUML community is expanding and is expected to gain a lot of grounds. Eclipse, Papyrus and open source community are among the players of xtUML. It is observed that a fully open source governance and ecosystem around xtUML has dramatically increased the pace of advancements in the tooling and facilitated collaboration among users, suppliers and academics. This is because openness, transparency and elimination of exclusive ownership fosters an environment of security.

The order of modeling encourages as much information as reasonable to be captured in data with the exposure of abstractions at the highest possible level. The method is considered object-oriented due to its emphasis on data modeling. This object concept emphasizes relations between the data abstractions. UML class diagrams provide the notational richness required to capture clear abstractions of conceptual entities with classes, attributes and various forms of associations relating them. UML state machine diagrams formalize the lifecycles of individual UML classes. Concurrent sequential processing is captured in a plurality of relatively simple, communicating instance-based state machines. Finally, activity semantics are modeled in class

operations, state machine states and transitions, and function bodies using an abstract action language that is Turing Complete but platform independent.

Models are partitioned along subject matter boundaries and deployed as compositions using UML component diagrams. xtUML models are interpretively executable following a set of rules. These rules enforce the semantics of the model artifacts and establish a basis to govern time, order and priority. Execution can be performed in simulations run by humans enforcing the rules or by an xtUML interpreter that automates the execution for model testing purposes. A corollary set of semantics governs the transformation of xtUML from one representation into semantically equivalent forms represented in lower-level target deployment languages such as Ada, SPARK, Java, C, C++, MISRA-C, AUTOSAR, SystemC, or VHDL. The process of translating xtUML into other forms is called model compilation and is performed by a model compiler.

A model compiler is a refinement of code generation in its complete and strict mapping of the semantic rules of the language. A model compiler must guarantee adherence to semantics to preserve execution behavior between forms. Model compilers can translate only from a higher level of abstraction to a lower level (or the same level). Model compilers can insert additional platform-specific detail into the transformation output. Shlaer Mellor xtUML model compilers translate PIMs (Platform Independent Models) to PSMs (Platform Specific Models).

The purpose of xtUML is to capture executable models and not just diagrams. This enables testing the application design before coding it which ensures a verified executable specification. The execution is captured using OAL (Object Action Language) which supports:

1. Create/Delete instances
2. Read/write attributes
3. Read parameter values
4. Relate/unrelated instances

5. Invoke operations/set parameter values
6. Send events/set parameter values
7. Find instances
8. Computation
9. Create/read/write local variables
10. Control: iterate, loop, decision(if elif else endif)

OAL is used to define execution in several UML elements, namely:

1. States
2. Transitions
3. Operations (Instance or class based)
4. Ports
5. Mathematically derived attributes
6. Bridge operations
7. Functions

The completeness of the executable model allows the generation of complete target models as opposed to just skeleton that can be used to generate headers. Table 7 shows supported operators.

Table 7 xtUML Operators

Type	Operators
Arithmetic	+, -, *, /. %, unary -
Boolean	AND, OR, unary NOT
Relational	==, !=, <, <=, >, >=
Assignment	Assign x=1
Instance Handles	==, !=, empty, not_empty, cardinality

Figure 11 shows an example of some expression in OAL [96]

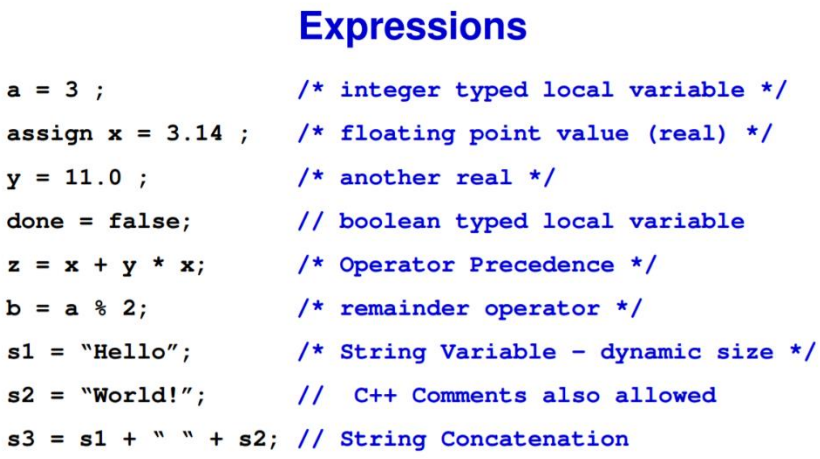

Figure 11 xtUML Expressions

4.4.3 UML to SAL Mapping Rules

Table 8 UML/SAL Mapping Rules list model compiler mapping rules used to generate UML to SAL.

Table 8 UML/SAL Mapping Rules

UML Element	**SAL Mapping**
Component	Translate into the entire context in SAL Ex: WatchdogM: CONTEXT BEGIN END
System	Combines all defined SAL modules (UML state machines) system: MODULE = MOD_1[] MOD_2;

State Machine	The state machine is represented as a module in SAL.
	
	Ex: SM_1: MODULE
	
	BEGIN
	
	<used instances>
	
	INITIALIZATION
	
	TRANSITION
	
	END;

Initial State	INITIALIZATION block within the SAL module will contain the initial state in the state machine

UML Element	**SAL Mapping**
	INITIALIZATION State = ST_STEADY;
Transition	Specified in TRANSITION block. The transition is specified as follows (the first part is a Boolean statement): statemachine_status = SM_current_State AND statemachine_action = SM_EVT_FIRED --> statemachine_status' = SM_Destination_state <Set of actions> (State = ST_DOWNSHIFTING AND CONT.timerStarted = TRUE AND EVT = EVT_SPEEDLESSDOWNTHROTTLE) --> EVT' = IF CONT.timerStarted = TRUE THEN EVT_TIMEELASPEGEARUP ELSE EVT_SPEEDLESSDOWNTHROTTLE ENDIF; CONT'.timerStarted = FALSE;
Event (trigger for a transition)	Define new TYPE in system which contains all possible state machines events, then use as a global or input in the <used instances> block of the module. EVT_WdgM: TYPE = { EVT_WDGM268,

UML Element	SAL Mapping

	EVT_WDGM269,
	EVT_WDGM201,
	EVT_WDGM202,
	EVT_WDGM203,
	EVT_WDGM204,
	EVT_WDGM300,
	EVT_WDGM205,
	EVT_WDGM206,
	EVT_WDGM207,
	EVT_WDGM291,
	EVT_WDGM208,
	EVT_WDGM209,
	EVT_Startup
	};
	
	MOD_WdgM : MODULE =
	BEGIN
	%% Global Section

UML Element	**SAL Mapping**
	GLOBAL EVT: EVT_WdgM

State	Define new TYPE in system which contains all possible states, then use as local or input. Same exact way as EVT
	
	ST_WdgM : TYPE = {
	
	ST_STATUS_OK,
	
	ST_STATUS_DEACTIVATED,
	
	ST_STATUS_FAILED,
	
	};
	
	MOD_WdgM : MODULE =
	
	BEGIN
	
	%% Global Section
	
	GLOBAL WdgM_State: ST_WdgM

State Actions	Specified in a TRANSITION block to self. The transition is specified as follows (the first part is a Boolean statement): statemachine_status = SM_Origin_State AND variableInspect = 255 --> statemachine_status' = SM_Origin_State; startupCounter' = 1;
Class Attributes	Define bounded range for the variable:

UML Element	**SAL Mapping**

Extended to include max limit	vehicleSpeed_idx: INTEGER = 240;
	
	Define bounded type:
	
	vehicleSpeed_type: TYPE = [0..vehicleSpeed_idx];
	
	Define within class attributes structure in SAL
	
	REC_GEARCONTROLLER: TYPE = [#
	
	vehicleSpeed: vehicleSpeed_type,
	
	#];
	
	Define in Module as an instance to be used globally:
	
	GLOBAL CONT: REC_GEARCONTROLLER

Class Operations	Statements mapped to SAL and inlined in a state action when called inside state or transition
Satisfiability conditions (UML Model)	Th<id>: THEOREM <module name> |- G((preconditions) => F(expected outcome)); Ex:

UML Element	SAL Mapping
Satisfiability conditions are mapped to theorems)	WDGM205: THEOREM system |- G(WdgM.WdgmAliveSupervisionStatus = 0 AND WdgM.WdgmDeadlineSupervisionStatus = 0 AND WdgM.WdgmLogicalSupervisionStatus=0 => G(WdgM_State = ST_STATUS_OK)); Th<id>: THEOREM <module name> |- G(Upper bound check) AND (Lower bound check); Safe_WdgM_WDGM327: THEOREM system |- G(FailedAliveSupervisionRefCycleTol <= 255 AND FailedAliveSupervisionRefCycleTol >= 0);

The transformation mapping model compiler is composed of several functions. Each function is responsible for UML to SAL mapping. A list of functions implemented in the model compiler is summarized below:

1. FilePro: responsible for generating SAL file prologue. It takes UML components and generates <ComponentName>.sal file component. It additionally declares the context of the .sal using the components name as the context's name followed by 'BEGIN' keyword which is mandatory in SAL semantics.
2. FileStates: Given a UML class, creates the declaration of the different states, events, input states and input events in an enumerated form in SAL notation. The data type name format is <class name>_<State or InputState>.
3. FileModules: This function creates the SAL module for each state machine. It takes component as a parameter and searches the component for all its classes' state machines. For each class state machine instance, a module is created.

'MODULE=BEGIN' preceded by the class's generated name are written to the SAL file. CreateDecls function is then called with a component and a class name to define all modules relevant declarations. All SAL transitions in this class's state machine are then written to the SAL file. CreateBody function is then called with a state/transition, class, and component arguments to traverse any state/transition preconditions/post conditions to generate the transition/State actual body.

4. CreateDecls: As previously mentioned, this function takes the class and component names. It creates declarations within each class's respective module, namely, declare an instance of the class's states and input states, local to the module as well as class's events and input events global to the module. It will also initialize the instance of the class's status to the initial state in the UML state machine.
5. CreateObjectSAL: Invokes previously described functions in the correct order to generate the SAL representation of the state machine.
6. CreateBody: Parses the preconditions and post conditions checks and generates state transition actions/checks and/or state actions.
7. CreateTheroems: Traverses all UML model satisfiability conditions to generate theorem mapping in SAL notation.

4.5 Model Checking

Model checking is based on SAL model checkers. SAL model checkers are based on several technologies. The below subsection introduces the technologies that SAL model checkers are based on followed by a brief list of SAL model checkers.

4.5.1 Model Checkers Technologies

4.5.1.1 SAT Based Model Checking

Also Called Boolean satisfiability. SAT(Satisfiability) based checking is to determine if there exists an interpretation that satisfies a given Boolean formula. In other words, it establishes if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. If no such assignments exist, the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, it is called unsatisfiable, otherwise satisfiable. For

example, the formula "a AND NOT b" is satisfiable because one can find the values a = TRUE and b = FALSE, which make (a AND NOT b) = TRUE. In contrast, "a AND NOT a" is unsatisfiable. To emphasize the binary nature of this problem, it is frequently referred to as Boolean or propositional satisfiability.

4.5.1.2 SMT (Satisfiability Modulo Theories).

The SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Examples of theories typically used in computer science are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, bit vectors and so on. SMT can be thought of as a form of the constraint satisfaction problem and thus a certain formalized approach to constraint programming.

4.5.1.3 BDD - Binary Decision Diagram

A binary decision diagram (BDD) is a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression.

Bounded model checking algorithms unroll the FSM for a fixed number of steps K and check whether a property violation can occur in K or fewer steps. The process can be repeated with larger and larger values of k until all possible violations have been ruled out

4.5.1.4 Model Solver dependencies/ Techniques

Correctness properties are expressed in SAL by means of LTL (Linear Temporal Logic) or CTL (Computational Tree Logic) formulas. [Appendix A includes SAL examples of correctness properties]. In SAL, verification of the models relies mostly on the infinite-state bounded model checker. The model checker is used as a refutation tool. It searches for counter examples to a given property. It is used as a verification tool and applied to the models using two techniques: proof by induction and proof by abstraction.

Proof by induction (k-induction) assumes that a system S satisfies an invariant P in two steps. The first step shows that all states that are reachable from initial state of S in at most k steps satisfy P. In the inductive step, one shows that for any trajectory of length k+1, if the first k states satisfy P then the last state also satisfies P. This technique is not always scalable to industrial-size use cases.

Proof by Abstraction which amounts to finding a disjunctive variant that implies a safety property under consideration which can be proven using induction at depth 1.
Proof by induction and abstraction is claimed to be a robust technique that is applicable to a wide class of communication protocols. The method of discovering suitable abstractions is claimed to be reusable across protocols.

4.5.2 SAL Model Checkers

SAL comes with a lot of model checkers based on technologies discussed in 4.5.1. sal-wfc is a well-formedness checker or SAL syntax compiler. Sal-smc is a symbolic model checker which is BDD-based for finite state systems [78]. Sal-deadlock-checker is an auxiliary tool, based on the symbolic model checker, for detecting deadlocks in finite state systems [78]. Sal-bmc is a bounded model checker for finite state systems based on SAT(Satisfiability) solving. In addition to refutation (i.e., bug detection and counterexample generation), the SAL bounded model checker can perform verification by k-induction. SAL can use several SAT solvers, but defaults to Yices [78]. Sal-inf-bmc is an infinite bounded model checker for infinite state systems based on SMT solving. In addition to refutation (i.e., bug detection and counterexample generation), the SAL infinite bounded model checker can perform verification by k-induction. SAL can use several SMT solvers, but defaults to Yices [78]. Sal-atg is an automated test generator which uses the symbolic, bounded, and infinite bounded model checkers to perform automated generation of input sequences [78].

Chapter 5. Case Study Modules

We have evaluated our proposed approach using three automotive modules. Two modules are part of the AUTOSAR standard basic software module stack, namely, FlexRay state manager and Watchdog Manager and the third module is an application layer module, namely, automatic transmission controller. We present the background of these case study modules in this chapter and the requirements that has been mapped into the design based on informal specifications of all three modules in our framework. These requirements will form the foundation for evaluating our framework. We will implement these requirements in the UML design, introduce design defects on the UML model and report the SAL model checkers response towards these introduced defects.

Additionally, we used a commercial implementation from an AUTOSAR BSW supplier of Watchdog Manager. The details of this commercial implementation includes challenges faced during design verification of the module, ISO26262 compliance challenges, and defects that were uncovered beyond the design phase that could have been detected in the design phase. We will present this implementation in this chapter and use it for a comparative analysis against our approach in results and discussions chapter. Our plan is to introduce these defects in our watchdog manager implementation design and verify that the model checkers are able to detect the design issues at the design stage.

5.1 AUTOSAR FlexRay State Manager

In the AUTOSAR Layered Software Architecture, the FlexRay State Manager belongs to the ECU Abstraction Layer, or more precisely, to the Communication Hardware Abstraction as depicted in Figure 4. AUTOSAR modules specification are uniform. We managed to verify two modules but our framework can be extended to any AUTOSAR module since all specifications are similar. The FlexRay State Manager shall provide an abstract interface to the AUTOSAR Communication Manager to startup or shutdown the

communication on a FlexRay cluster. The FlexRay State Manager does not directly access the FlexRay hardware (FlexRay Communication Controller and FlexRay Transceiver), but by means of the FlexRay Interface. The FlexRay Interface redirects the request to the appropriate driver module [103]. The FlexRay State Manager shall have one state machine for each FlexRay cluster. Figure 12 shows the FlexRay State Manager state machine as documented in the AUTOSAR FlexRay state manager AUTOSAR specification.

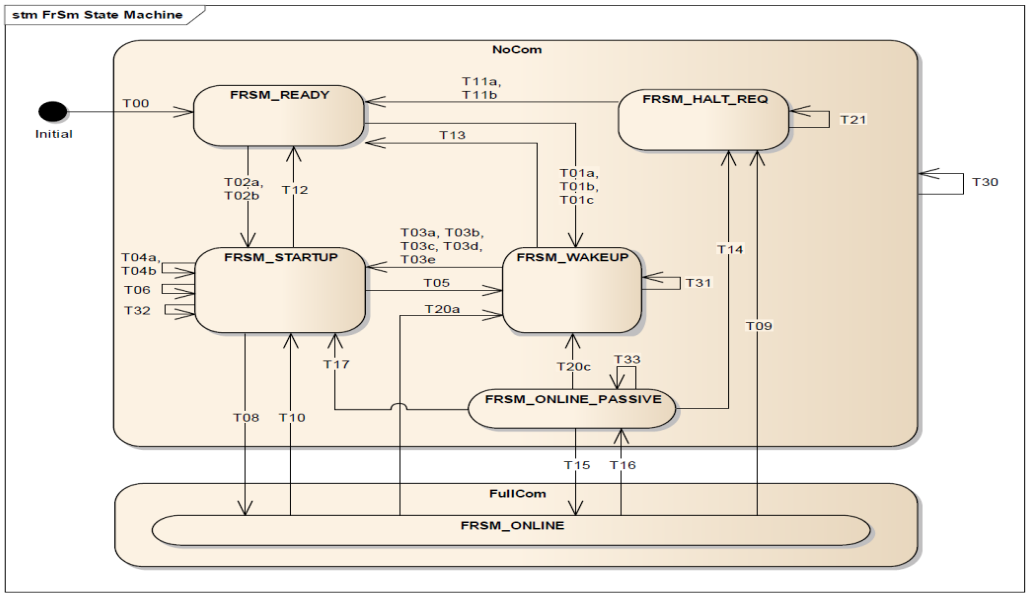

Figure 12 State Machine of FlexRay State Manager

5.1.1 Requirements to be verified

5.1.1.1 FRSM073 and FRSM074

Figure 13 depicts requirements FrSm073 and FrSm074 that govern transitions 2a, 2b, 3a, 3b, 3c, 3d and 3e as documented in the AUTOSAR FlexRay State manager specification. It shows the conditions that drive a transition and the actions to take once the transition is made. In the results and discussions chapter, we will verify that the below requirement hold in the design, introduce bugs and show how our framework can detect the bugs on the design level via the formal model checkers based on the automatically compiled UML design to SAL formal notation. All requirements are retrieved from AUTOSAR software specification document [103].

ID	Tran-sition	Event [Condition]	Actions
FrSm073:	T02a	[reqComMode = FullCom ∧ (¬ FrSMIsWakeupEcu ∨ WUReason = ALL_WU_BY_BUS) ∧ ¬ FrSmDelayStartupWithoutWakeup]	FE_TRCV_NORMAL startupCounter := 1 wakeupType := NoWakeup FE_START FE_ALLOW_COLDSTART start t2 start t3
	T02b	[reqComMode = FullCom ∧ (¬ FrSMIsWakeupEcu ∨ WUReason = ALL_WU_BY_BUS) ∧ FrSmDelayStartupWithoutWakeup]	FE_TRCV_NORMAL startupCounter := 1 wakeupType := NoWakeup FE_START start t1 start t2 start t3
FrSm074:	T03a	[wakeupFinished ∧ FrSmNumWakeupPatterns = *1* ∧ reqComMode = FullCom ∧ wakeupType = SingleChannelWakeup]	FE_START cancel t1 start t1
	T03b	[wakeupFinished ∧ FrSmNumWakeupPatterns > *1* ∧ (wakeupTransmitted ∨ ¬ *t1_IsActive*) ∧ reqComMode = FullCom ∧ wakeupType = SingleChannelWakeup]	FE_START cancel t1 start t2 FE_ALLOW_COLDSTART
	T03c	[wakeupFinished ∧ FrSmNumWakeupPatterns > *1* ∧ ¬ wakeupTransmitted ∧ reqComMode = FullCom ∧ wakeupType = SingleChannelWakeup]	FE_START
	T03d	[wakeupFinished ∧ reqComMode = FullCom ∧ wakeupType = DualChannelWakeup]	FE_START start t2
	T03e	[wakeupFinished ∧ reqComMode = FullCom ∧ wakeupType = DualChannelWakeupForward]	FE_START FE_ALLOW_COLDSTART start t2

Figure 13 Requirements 73 and 74 in FlexRay State Manager Module

The requirements govern the conditions that should be satisfied for a transition/state to be valid. These requirements shall be the basis for the theorems as well to ensure that at no time, the conditions will be satisfied while in a wrong state or invalid transition.

5.1.1.2 FRSM033

Verify range values are compliant to specifications for parameters [103]:

startupCounter: uint8 [0-255]

wakeupCounter:uint8[0-255]

5.2 AUTOSAR WatchDog Manager

Most embedded systems need to be self-reliant. It is not usually possible to wait for someone to reboot them if the software hangs. Some embedded designs, such as space probes, are simply not accessible to human operators. If their software ever hangs, such systems are permanently disabled. In other cases, the speed with which a human operator might reset the system would be too slow to meet the uptime requirements of the product. Watchdog module is used to automatically detect software anomalies and take corrective actions such as a reset the processor.

The Watchdog Manager is a basic software module at the service layer of the standardized basic software architecture of AUTOSAR as shown in Figure 4. It is able to supervise the program execution abstracting from the triggering of hardware watchdog entities. It supervises the execution of a configurable number of *Supervised Entities*. When it detects a violation of the configured temporal and/or logical constraints on program execution, it takes a number of configurable actions to recover from this failure. The watchdog Manager provides three mechanisms [102]:

1. Alive supervision – for supervision of timing of periodic software
2. Deadline monitoring – for aperiodic software
3. Logical monitoring – for supervision of the correctness of the execution sequence.

The Watchdog Manager supervises the execution of software. The logical units of supervision are Supervised Entities. There is no fixed relationship between Supervised Entities and the architectural building blocks in AUTOSAR, i.e., SW-Cs, CDDs, RTE, BSW modules, but typically a Supervised Entity may represent one SW-Cs or a Runnable within an SW-C, a BSW module or CDD depending on the choice of the developer. Important places in a Supervised Entity are defined as Checkpoints. The code of Supervised Entities is interlaced with the calls of Watchdog Manager that report to the Watchdog Manager when they have reached a Checkpoint [102].

Each Supervised Entity has one or more Checkpoints. The Checkpoints and Transitions between the Checkpoints of a Supervised Entity form a Graph. This

Graph is called Internal Graph. Moreover, checkpoints from different Supervised Entities may also be connected by External Transition, forming an External Graph. There can be several External Graphs in each Watchdog Manager mode [102].

A Graph may have one or more initial Checkpoints and one or more final Checkpoints. Any sequence of starting with any initial checkpoint and finishing with any final checkpoint is correct (assuming that the checkpoints belong to the same Graph). After the final Checkpoint, any initial Checkpoint can be reported. Within the Watchdog Manager settings, it is possible to configure the required timing of Checkpoints as well as the allowed External and Internal Graphs [102].

At runtime, Watchdog Manager verifies if the configured Graphs are executed. This is called Logical Supervision. Watchdog Manager verifies also the timing of Checkpoints and Transitions. The mechanism for periodic Checkpoints is called Alive Supervision and for aperiodic Checkpoints it is called Deadline Supervision. The granularity of Checkpoints is not fixed by the Watchdog Manager. Few coarse-grained Checkpoints limit the detection abilities of the Watchdog Manager. For example, if an application SW-C only has one Checkpoint that indicates that a cyclic Runnable has been started, then the Watchdog Manager is only capable of detecting that this Runnable is re-started and check the timing constraints. In contrast, if that SW-C has Checkpoints at each block and branch in the Runnable the Watchdog Manager may also detect failures in the control flow of that SW-C. High granularity of Checkpoints causes a complex and large configuration of the Watchdog Manager [102].

The three supervision mechanisms supervise each supervised entity. A Supervised Entity may have one, two or three mechanisms enabled. Based on the results from each of enabled mechanisms, the status of the Supervised Entity (called Local Status) is computed. When the status of each Supervised Entity is determined, then based on each Local Supervision Status, the status of the whole MCU is determined (called Global Supervision Status). Watchdog has three types of supervision: Alive supervision, deadline supervision and Logical supervision [102].

5.2.1 Alive Supervision

Periodic *Supervised Entities* have constraints on the number of times they are executed within a given time span. By means of Alive Supervision, Watchdog Manager checks periodically if the Checkpoints of a Supervised Entity have been reached within the given limits. This means that Watchdog Manger checks if a *Supervised Entity* is run not too frequently or not too rarely [102].

5.2.2 Deadline Supervision

Aperiodic or episodical *Supervised Entities* have individual constraints on the timing between two *Checkpoints*. By means of Deadline Supervision, Watchdog Manager checks the timing of transitions between two *Checkpoints* of a *Supervised Entity*. This means that Watchodog Manager checks if some steps in a Supervised Entity take a time that is within the configured minimum and maximum [102].

5.2.3 Logical Supervision

Logical supervision is a fundamental technique for checking the correct execution of embedded system software. Please refer to the safety standards (IEC 61508 or ISO26262) when logical supervision is required. Logical supervision focuses on control flow defects, which cause a divergence from the valid (i.e. coded/compiled) program sequence during the error-free execution of the application. An incorrect control flow occurs if one or more program instructions are processed either in the incorrect sequence or are not even processed at all. Control flow errors can lead to data corruption, microcontroller resets, or fail-silence violations. For the control flow graph this implies that every time the Supervised Entity reports a new Checkpoint, it must be verified that there is a Transition configured between the previous Checkpoint and the reported one [102].

5.2.4 Local Supervision State Machine

The local supervision status state machine determines the status of the Supervised Entity. This is done based on the following:

1- Previous value of the Local Supervision Status
2- Current values of alive supervision, deadline supervision and logical supervision.

Figure 14 shows the Watchdog Manager state machine. The states and transitions are detailed in AUTOSAR Watchdog Manager published Software Specification document [79] [102].

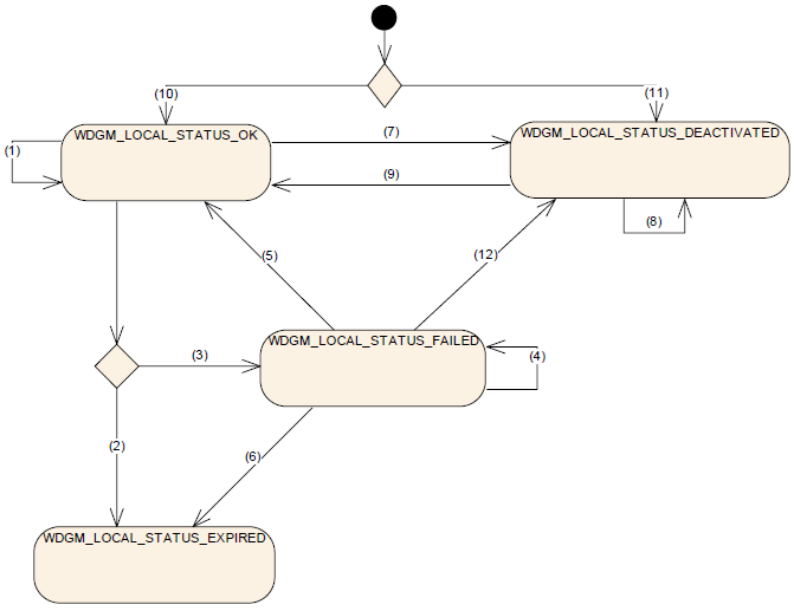

Figure 14 Watchdog Manager Local Supervision Status

5.2.5 Requirements to be verified

In this section, we present selected requirements that we plan to verify using our framework. In the results and discussions chapter, we will verify that the below requirements hold in the design, introduce bugs and show how our framework can detect the bugs on the design levels via the formal model checkers based on the automatically compiled UML design to SAL formal notation. All requirements are retrieved from AUTOSAR software specification document [102].

WDGM202

WDGM202 describes transition 2 in the local state machine as depicted in Figure 15[102]. The requirement specifies the conditions that should be satisfied in order for a transition from OK to expired states to take place in the local state machine. We will use our framework to verify that at no time, the below conditions will be met and the transition will not fire or that we are in a state other than expired while the transition conditions satisfying expired state are true.

[WDGM202] ⌈If the *Supervised Entity* was in *Local Supervision Status* WDGM_LOCAL_STATUS_OK AND:

1. (At least one result of Alive Supervision of the *Supervised Entity* is incorrect and a Failure Tolerance of zero is configured (see configuration parameter WdgMFailedAliveSupervisionRefCycleTol [WDGM327_Conf]) OR
2. If the result of at least one Deadline Supervision of the *Supervised Entity* or the result of at least one Logical supervision of the *Supervised Entity* is incorrect),

THEN the function WdgM_MainFunction shall change the *Local Supervision Status* to WDGM_LOCAL_STATUS_EXPIRED⌋

Figure 15 Requirement 202 - Watchdog Manager Module

WDGM203

WDGM203 describes transition 3 in the local state machine as depicted in Figure 16 [102]. The requirement basically specifies the conditions that should be satisfied in order for a transition from OK to failed states to take place in the local state machine. We will use our framework to verify that at no time, the below conditions will be met and the transition will not fire or that we are in a state other than failed while the transition conditions satisfying failed state are true.

[WDGM203] ⌈If the *Supervised Entity* was in *Local Supervision Status* WDGM_LOCAL_STATUS_OK AND:

1. (If the result of at least one Alive Supervision of the *Supervised Entity* is incorrect and a Failure Tolerance greater than zero is configured (see configuration parameter WdgMFailedAliveSupervisionRefCycleTol [WDGM327_Conf]) AND
2. If all the results of Deadline Supervision of the *Supervised Entity* and all results of Logical supervision of the *Supervised Entity* are correct),

THEN the function WdgM_MainFunction shall change the *Local Supervision Status* to WDGM_LOCAL_STATUS_FAILED and increment the counter for failed supervision reference cycles

Figure 16 Requirement 203 - Watchdog Manager

WDGM204

WDGM204 describes transition 4 in the local state machine as depicted in Figure 17 [102]. The requirement basically specifies the conditions that should be satisfied in order for a stay in FAILED transition in the local state machine. We will use our

framework to verify that at no time, the below conditions will be met the state machine is in a state other than failed.

[WDGM204] ⌈If the *Supervised Entity* was in *Local Supervision Status* WDGM_LOCAL_STATUS_FAILED AND:

1. (If the result of at least one Alive Supervision is incorrect and the *counter for failed supervision reference cycles* does not exceed the configured Failure Tolerance (see parameter WdgMFailedAliveSupervisionRefCycleTol [WDGM327_Conf]) AND
2. If all the results of Deadline Supervisions of the *Supervised Entity* and all the result of Logical Supervision of the *Supervised Entity* are correct),

THEN the function WdgM_MainFunction shall keep the *Local Supervision Status* in WDGM_LOCAL_STATUS_FAILED and increment the counter for failed supervision reference cycles

Figure 17 Requirement 204 - Watchdog Manager

WDGM300

WDGM300 describes transition 4 in the local state machine as depicted in Figure 18 [102]. The requirement basically specifies the conditions that should be satisfied in order for a stay in FAILED transition in the local state machine. We will use our framework to verify that at no time, the below conditions will be met the state machine is in a state other than failed.

[WDGM300] ⌈If the *Supervised Entity* was in *Local Supervision Status* WDGM_LOCAL_STATUS_FAILED AND:

1. (If all the results of Alive Supervision of the *Supervised Entity* are correct and the counter for failed supervision reference cycles is > 1) AND
2. If all the result of Deadline Supervision of the *Supervised Entity* and all the result of Logical supervision of the *Supervised Entity* are correct),

THEN the function WdgM_MainFunction shall keep the *Local Supervision Status* in WDGM_LOCAL_STATUS_FAILED and decrement the *counter for failed supervision reference cycles*

Figure 18 Requirement 300 - Watchdog Manager Module

WDGM205

WDGM205 describes transition 5 in the local state machine as depicted in Figure 19 [102]. The requirement basically specifies the conditions that should be satisfied in

order for a transition from failed to OK states to take place in the local state machine. We will use our framework to verify that at no time, the below conditions will be met and the transition will not fire or that we are in a state other than OK while the transition conditions satisfying OK state are true.

[WDGM205] ⌈If the *Supervised Entity* was in *Local Supervision Status* `WDGM_LOCAL_STATUS_FAILED` AND:

1. (If all the results of Alive Supervision of the *Supervised Entity* are `correct` and the *counter for failed supervision reference cycles* equals 1) AND
2. If all the results of Deadline Supervisions of the *Supervised Entity* and all the results of Logical supervision of the *Supervised Entity* are `correct`),

THEN the function `WdgM_MainFunction` shall change the *Local Supervision Status* to `WDGM_LOCAL_STATUS_OK` and decrement the *counter for failed supervision reference cycles*

Figure 19 Requirement 205 - Watchdog Manager Module

WDGM206

WDGM206 describes transition 6 in the local state machine as depicted in Figure 20 [102]. The requirement basically specifies the conditions that should be satisfied in order for a transition from failed to expired states to take place in the local state machine. We will use our framework to verify that at no time, the below conditions will be met and the transition will not fire or that we are in a state other than expired while the transition conditions satisfying expired state are true.

[WDGM206] ⌈If the *Supervised Entity* was in *Local Supervision Status* `WDGM_LOCAL_STATUS_FAILED` AND:

1. (If at least one result of Alive Supervision is `incorrect` and the *counter for failed supervision reference cycles* exceeds the configured Failure Tolerance (see configuration parameter `WdgMFailedAliveSupervisionRefCycleTol` [WDGM327_Conf]) OR
2. If at least one result of Deadline Supervision of the *Supervised Entity* or at least one the result of Logical supervision of the *Supervised Entity* is `incorrect`),

THEN the function `WdgM_MainFunction` shall change the *Local Supervision Status* to `WDGM_LOCAL_STATUS_EXPIRED`

Figure 20 Requirement 206 - Watchdog Manager Module

WDGM207

WDGM207 describes transition 7 in the local state machine as depicted in Figure 21 [102]. The requirement basically specifies the conditions that should be satisfied in order for a transition from OK to deactivated state to take place in the local state machine. We will use our framework to verify that at no time, the below condition will be met and the transition will not fire or that we are in a state other than deactivated while the setMode function is called with a deactivated state.

[WDGM207] ⌈If the *Supervised Entity* was in *Local Supervision Status* WDGM_LOCAL_STATUS_OK and if a call of WdgM_SetMode switches to a mode which deactivates the *Supervised Entity* (see [WDGM283]), then the Watchdog Manager module shall change the *Local Supervision Status* to WDGM_LOCAL_STATUS_DEACTIVATED

Figure 21 Requirement 207 - Watchdog Manager Module

WDGM291

WDGM291 describes transition 12 in the local state machine as depicted in Figure 22 [102]. The requirement basically specifies the conditions that should be satisfied in order for a transition from Failed to deactivated state to take place in the local state machine. We will use our framework to verify that at no time, the below condition will be met and the transition will not fire or that we are in a state other than deactivated while the transition condition is met. We will also verify that the design does not allow the transition from expired to deactivated as described in the requirement.

[WDGM291] ⌈If the *Supervised Entity* was in *Local Supervision Status* WDGM_LOCAL_STATUS_FAILED and if a call of WdgM_SetMode switches to a mode in which the *Supervised Entity* is Deactivated (see [WDGM283]), then the Watchdog Manager module shall change the *Local Supervision Status* to WDGM_LOCAL_STATUS_DEACTIVATED⌋

Note that the above requirement is only applicable for the WDGM_LOCAL_STATUS_FAILED status, but not for WDGM_LOCAL_STATUS_EXPIRED.

Figure 22 Requirement 291 - Watchdog Manager Module

WDGM208

WDGM208 describes transition 8 in the local state machine as depicted in Figure 23 [102]. The requirement basically specifies the conditions that should be satisfied in order for the state machine to stay in deactivated state in local state machine of the Watchdog Manager Specification[102]. Verification of this requirement will ensure that the state is correct given the conditions and that no supervision functions are performed while in the state.

[WDGM208] ⌈If the *Supervised Entity* was in the *Local Supervision Status* `WDGM_LOCAL_STATUS_DEACTIVATED`, the functions `WdgM_CheckpointReached` and `WdgM_MainFunction` shall not perform any *Supervision Functions* for this *Supervised Entity* and leave the *Local Supervision Status* in the state `WDGM_LOCAL_STATUS_DEACTIVATED`.

Figure 23 Requirement 208 - Watchdog Manager Module

WDGM209

WDGM209 describes transition 9 in the local state machine as depicted in Figure 24 [102]. The requirement basically specifies the conditions that should be satisfied in order for a transition from deactivated to OK state to take place in the local state machine. We will use our framework to verify that at no time, the below condition will be met and the transition will not fire or that we are in a state other than OK while the transition condition is met.

[WDGM209] ⌈If the *Supervised Entity* was in *Local Supervision Status* `WDGM_LOCAL_STATUS_DEACTIVATED` and if a call of `WdgM_SetMode` switches to a mode in which the *Supervised Entity* is active (see [WDGM282]), then the Watchdog Manager module shall change the *Local Supervision Status* to `WDGM_LOCAL_STATUS_OK`.

Figure 24 Requirement 209- Watchdog Manager Module

WDGM327

WDGM327 describes boundary conditions for a configuration parameter within the module as shown in Figure 25 [102]. We will verify that it is not possible at any point in the design for the failed alive supervision reference cycle tolerance to exceed the specification range.

SWS Item	WDGM327_Conf :	
Name	WdgMFailedAliveSupervisionRefCycleTol {WDGM_FAILED_SUPERVISION_REFERENCE_CYCLE_TOLERANCE}	
Description	This parameter shall contain the acceptable amount of reference cycles with incorrect/failed alive supervisions for this Supervised Entity.	
Multiplicity	1	
Type	EcucIntegerParamDef	
Range	0 .. 255	

Figure 25 Requirement 327 - Parameter range - Watchdog Manager

5.3 Automatic Transmission Controller - ATC

A transmission control entity is a device that controls transmission electronically to achieve better fuel economy, reduced engine emissions, greater shift system reliability, improved shift feel and improved shift speed. It uses sensors from the vehicle and data provided by engine control unit to calculate how and when to change gears in the vehicle. Figure 26 shows a state machine of the ATC [104]. The inputs are throttle and vehicle speed and the output is the desired gear number.

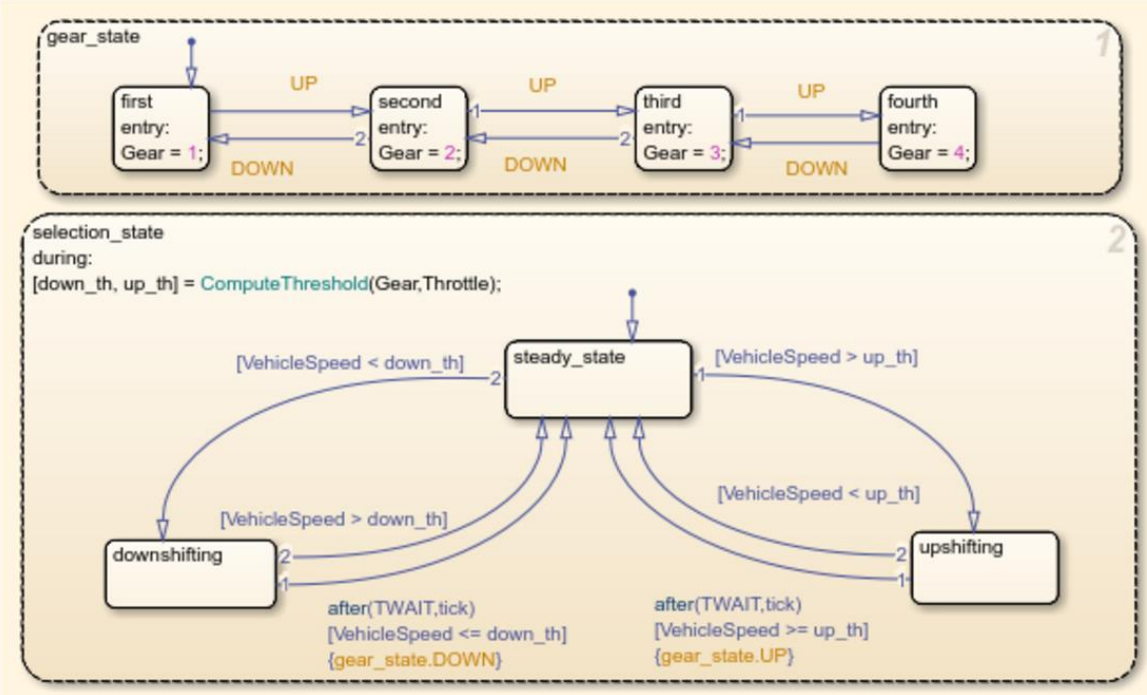

Figure 26 ATC State Machine

The model computes the upshift and downshift speed thresholds as a function of the instantaneous values of gear and throttle. While in steady_state, the model compares

these values to the present vehicle speed to determine if a shift is required. If so, it enters one of the confirm states (upshifting or downshifting). If the vehicle speed no longer satisfies the shift condition, while in the confirm state, the model ignores the shift and it transitions back to steady_state. This prevents extraneous shifts due to noise conditions. If the shift condition remains valid for a duration, the model transitions through the lower junction and, depending on the current gear, it broadcasts one of the shift events. Subsequently, the model again activates steady_state after a transition through one of the central junctions. The shift event, which is broadcast to the gear_selection state, activates a transition to the appropriate new gear [104].

5.3.1 Requirements to be verified

In this application module, we will assume application level requirements or safety assumptions.

If the vehicle is in a second gear vehicle speed range and throttle while in first gear position, the state second should be active in gear_state and steady_state is active in selection state [104].

If the vehicle speed exceeds 21 km/h given the previous condition, transition to up-shift should happen, stabilize in up-shift and gear_state should be in second state[104].

5.4 Industrial Challenges – Commercial Watchdog Manager Implementation

In order to evaluate the existing challenges faced by embedded automotive suppliers, we got data regarding the hardships that currently face automotive embedded suppliers as well as an analysis of the bugs they identify and interpretation of cons in the process/flow that lead up to these bugs. This evaluation needed to be based on an industrial partner that can benefit from our proposed flow and is willing to provide data regarding the current flaws as well as defects that are still identified in the late testing cycle or even after release. Mentor Graphics shared their challenges in developing AUTOSAR Watchdog Manager implementation in compliance to ASIL B ISO 26262 level as well as the defects that were uncovered during testing/production releases.

5.4.1 Verification challenges

BSW supplier reports that one of the major challenges faced during their verification cycle of BSW modules while attempting to be ISO26262 complaint was the inability to follow the design verification guidelines.

Formal and semi-formal verification methods generally apply when working with model-based software but not with embedded C code, or with embedded software designs. So, when working with embedded C software, static analysis, and control flow/data flow analysis were considered as acceptable alternatives by the supplier and an argument was made against performing semi-formal or formal verification against ISO-26262 recommendations as shown in Table 2 Methods for the verification of the software architectural design. This forced the supplier to drop to ASIL B compliance since they were unable to comply with highly recommended requirement of design semi-formal verification in ASILs C and D.

The supplier utilized methods that apply on the source code itself using manual methods. Control flow/data flow analysis were done via manually analyzing the control statements inside source code and creating control flow/data flow graphs for control statements and variables. This was possible since the modules under verification were small, however with larger scale modules, this will not be feasible and has to be automated.

From the supplier perspective, control flow and data flow analysis execution on the source code don't really provide additional value over static analysis and code coverage tools since most of the detectable bugs via control/data flow analysis can be detected by static analysis tools and code coverage tools during unit testing, or by manual code review. In a nut shell, verification is always assumed to be on the code as opposed to the design level due to the lack of an automated verification flow on the design level.

The supplier reports that design reviews were primarily based on several review iterations and re-writes of the design since it lacked needed details and guidance to be

a comprehensive design. Adding control/data flow in the design helped address this design lack of details which primarily impacts the implementation and the identification of design level bugs. The current flow basically pushes design bugs to the implementation cycle and allows the detection of the bugs introduced in both stages only after the code is written and is in a testable state.

The supplier also perceives semi-formal verification as means to automatically derive test cases in accordance with ISO 26262 test case derivation guidelines on the design as shown in Table 5. The recommendations show that in order to achieve at least ASIL B, boundary value analysis and generation and analysis of equivalence classes should be used in test case derivation.

5.4.2 Defects beyond Design Stage

Seventy one defects were raised during ASIL B compliancy endeavor of the WatchDog Manager module. Additionally, some of the reported defects were uncovered after production and during customer module integration endeavors. The table below summarizes the raised defects number and classification.

Table 9 Categorization of identified Defects

Defect Count	Classification
16	Logic Bugs
35	Non-compliance to Specification
20	Traceability

Defects under logic bugs category include but are not limited to, bugs such as array bound issues, incorrect array index, invalid mathematical operator, and last array index not getting initialized properly. Defects under non-compliance to specification includes defects such as WDGM triggers watchdog Interface to be in WDGIF_OFF_MODE while in WDGM_G_STATUS_STOPPED state (non-

compliance to WDGM122) and WdgMExpectedAliveIndications, which is a defined parameter in the specification holding the amount of expected alive indications, range does not match AUTOSAR watchdog manager specification. The remaining defects were related to traceability (Missing text cases, missing relations between requirement and design/code/test elements). 5.4.3 details sampled defects that will be verified in our results and discussions chapter.

71 defects were introduced in the design stage of this project. All of them slipped into the testing stage and were not properly detected in the design stage. Part of our approach evaluation is to decrease the 100% slippage factor between design and implementation stages.

5.4.3 Defects

WdgMExpectedAliveIndications Range

This defect was reported after delivering the software to the customer. In the customer's attempt to define the alive indications to be 65535, which is valid assignment since the specification indicates a range between 0-65535, the software returned an error that only 0-255 is allowed for this parameter. Bug report is 17655. We will introduce the same defect in our design and verify that model checkers will identify the defect at the design stage via model checkers.

SetMode function Defect

This defect was reported in the integration testing stage. The specification indicates that function SetMode should return E_NOT_OK while the state machine is in FAILED or OK states. The requirement indicates that changing mode successfully should trigger the function to return E_OK and failing should trigger the function to return E_NOT_OK. In this defect, the function returned E_OK while state machine was in state OK which is in direct violation to requirement 154. Bug report is 17988. We will introduce the same defect in our design and verify that model checkers will identify the defect at the design stage via model checkers.

Improper Initialization

This defect was reported in system testing stage. The specification indicates that all module variables shall be initialized in WdgM_Init call – WDGM018. The defect elaborates on a non-initialized index of an array in the module. Bug report is 17907. We will introduce the same defect in our design and verify that model checkers will identify the defect at the design stage via model checkers.

Out of Bound array index

This defect was reported during integration testing. An array within the wdgM module was accessed with out of bound index. Bug report 17971. We will introduce the same defect in our design and verify that model checkers will identify the defect at the design stage via model checkers.

Incomplete boundary testing

This boundary testing defect was reported during system testing stage. A bug was raised that not all module parameters were boundary tested. The defect was intended to show lack of testing to ensure compliance to ISO-26262. Bug report 18149. We will introduce the same defect in our design and verify that model checkers will identify the defect at the design stage via model checkers.

Chapter 6. Case Study results and Comparative analysis

6.1 AUTOSAR FlexRay State Manager Results

AUTOSAR FlexRay State Manager SWS served as a baseline for implementation of the design in BridgePoint xtUML. It is a critical module in the AUTOSAR BSW communication stack that is used by all ECUs connected via a FlexRay bus. FlexRay supports high data rates, up to 10Mbits/s and supports both star and party line bus topologies and has two independent data channels for fault tolerance. The first usage was in BMW x5 2006 damping system. It is now used in several cars including Audi, Bentley, BMW, Lamborghini, Mercedes Benz, Rolls-Royce, Land Rover and Volvo. It is mainly used in bandwidth intensive safety critical applications. FlexRay state Manager module is responsible for managing the state machine of a FlexRay cluster impacting all modules running on top of ECUs depending on data being sent/coming over the FlexRay communication channel. Any Failure in the state machine module would affect all applications on ECUs that depend on data utilizing the bus. Similar to FlexRay State manager, there is CAN (bus protocol) state manager, LIN (Bus protocol) state manager and Ethernet (communication protocol) state manager among others. Therefore, verifying this module is crucial in automotive and shows that all state managers could be verified at the design stage in a similar manner.

The xtUML project consisted of a root package called FRSM. The root package contained all data types as documented in the specification and a component named FrSM_Comp. The component consisted of another package, Manager that contained a class definition FrSM. The class contains attributes, functions and state machine design as documented in the FlexRay state manager specification. Figure 27 shows a summary of the xtUML design elements of FlexRay state manager module. Section 6.1.1 details the mapping of the specification into the xtUML design. The figure shows a FlexRaySM xtUML project that has a FRSM package. The package hosts user defined data types (comM_ModType, FrSM_BswM_StateType, Std_ReturnType, wakeup_Type, WUReason_Type, FrSm_ConfigType,

Std_VersionInfoType) and a FrSM_Comp component. The FrSM_Comp contains a package that hosts the FrSM class. THe xtUML class defines all attributes, operations, and state machine as defined in the specification.

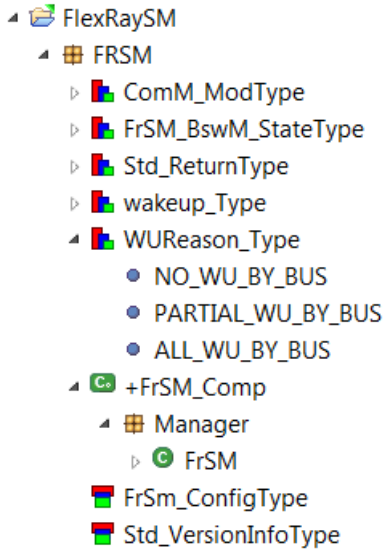

Figure 27 FlexRay xtUML Design

6.1.1 xtUML Design

FlexRay State Manager AUTOSAR specification [82] describes the module in the following order:

1- Defines all states in the state machine (READY, WAKEUP, STARTUP, HALT_REQ, ONLINE, ONLINE_PASSIVE) as shown in Figure 28.

2- Defines all variables used in the state machine and their type. Some examples are shown in Figure 29 where specification indicates that a variable reqComMode needs to be defined of type ComM_ModType, startupCounter of type integer, wakeupType of type Enumeration and others.

3- Condition variables that are evaluated at runtime that control transitions in the state machine as shown in Figure 30.

4- A table that describes the transition, conditions for the transition to take place and actions that should be executed once the transition happens as shown in Figure 13.

5- Function definitions and actions to be taken inside functions. An example is shown in Figure 31 where specification indicates that a function

FrSM_Init needs to be defined that returns void and accepts a configuration parameter of type pointer to FrSm_ConfigType. The intent of this function is to initialize the state manager and set the configuration parameters in the channel configuration.

FrSm032:
The state machine shall comprise the following states:

FrSm Cluster State	*Mapped FlexRay CC state*
FRSM_READY	POC:ready or (transitional) POC:default config or (transitional) POC:config or (transitional) POC:halt
FRSM_WAKEUP	POC:wake-up
FRSM_STARTUP	POC:start-up
FRSM_HALT_REQ	POC:normal active or POC:normal passive
FRSM_ONLINE	POC:normal active
FRSM_ONLINE_PASSIVE	POC:normal passive

Figure 28 FlexRay States

FrSm033:
In addition to its state, the state machine shall comprise the following variables.

FrSm Variable	*Type*	*Description*
reqComMode	ComM_ModeType	The communication mode that has been requested by the ComM. The communication modes are abbreviated in this document as follows: NoCom: COMM_NO_COMMUNICATION SilentCom:COMM_SILENT_COMMUNICATION FullCom: COMM_FULL_COMMUNICATION According to the definition of ComM_ModeType these modes are ordered as follows: NoCom < SilentCom < FullCom
startupCounter	Integer	The number of startup attempts that have been performed
wakeupType	Enum	The following values are supported: • SingleChannelWakeup • DualChannelWakeup • DualChannelWakeupForward • NoWakeup
wakeupTransmitted	boolean	True if vPOC!WakeupStatus = FR_WAKE-UP_TRANSMITTED for at least one attempt to transmit a wakeup pattern, false otherwise
busTrafficDetected	boolean	True if vPOC!WakeupStatus = FR_WAKE-UP_RECEIVED_HEADER or FR_WAKE-UP_RECEIVED_WUP for at least one attempt to transmit a wakeup pattern, false otherwise
wakeupCounter	Integer	The number of attempts that have been performed for transmitting a wakeup pattern.

Figure 29 FlexRay Variables

FrSm Condition	Type	Description
WUReason	Enum	If FrSMCheckWakeupReason is false, WUReason evaluates to NO_WU_BY_BUS. Otherwise if FrSMCheckWakeupReason is true, determine the wakeup reason by • calling FrIf_GetTransceiverWUReason for each transceiver of the FlexRay cluster and check for FRTRCV_WU_BY_BUS and and evaluate WUReason to • NO_WU_BY_BUS in case no wakeup has been detected. • PARTIAL_WU_BY_BUS in case the ECU is connected to both FlexRay channels of the cluster and wakeup has been detected for exactly one channel • ALL_WU_BY_BUS in case wakeup has been detected for all of the FlexRay channels of the cluster to which the ECU is connected. Note: The wakeup of a single channel of dual channel FlexRay cluster is not supported in this version of the FlexRay State Manager.

Figure 30 FlexRay Conditions

FrSm013:

Service name:	FrSm_Init

Syntax:	`void FrSm_Init(`
	`  const FrSm_ConfigType* FrSm_ConfigPtr`
	`)`

Service ID[hex]:	0x01
Sync/Async:	Synchronous
Reentrancy:	Non Reentrant
Parameters (in):	FrSm_ConfigPtr — Pointer to a selected configuration structure
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Description:	Initializes the FlexRay State Manager.

FrSm014: The FrSm_Init function shall
- initialize the state machines for all FlexRay clusters and set them into the state FRSM_READY;
- internally store the configuration data address to enable subsequent API calls to access the configuration data;

Figure 31 FlexRay Functions

The module specification was mapped into xtUML design. All defined types were mapped into user defined types in xtUML. FrSm033 requirement indicates how several variables should be defined. An example is reqComMode variable of type ComM_ModType enum that should be defined with the following enum values:

NoCom, SilentCom, FullCom. Figure 32 shows the mapping of this user defined type in xtUML.

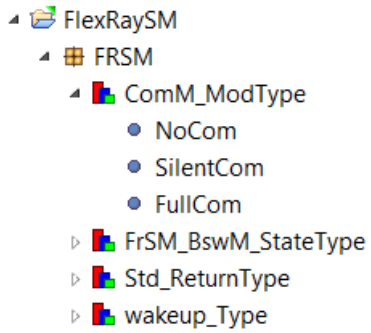

Figure 32 ComM_ModeType User Defined Type

Additionally, variables documented in the specification were mapped into class attributes of the FrSM class definition in xtUML as shown in Figure 33. xtUML has been extended to record the default value and max value for each variable so that it can be used to generate boundary conditions theorems in SAL notation.

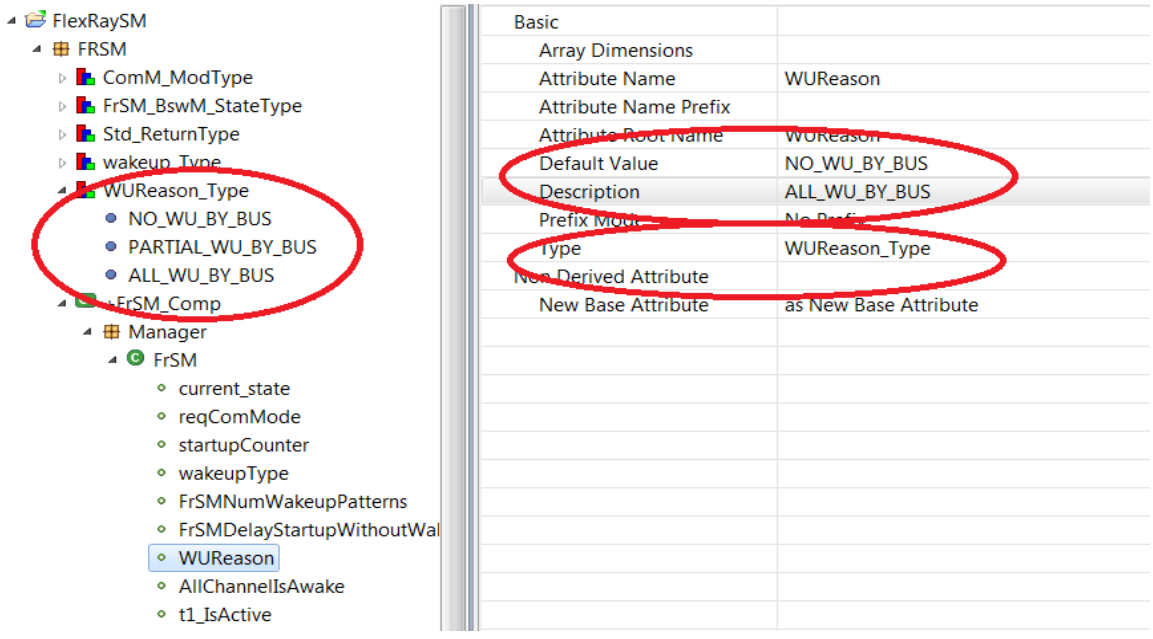

Figure 33 Variable Definition

FlexRay Manager States, state machine and transitions were also implemented in xtUML in accordance to FlexRay State Manager specification document. Figure 34 shows the xtUML implementation of the state machine. Each FlexRay State machine state checks for the conditions and triggers actions as documented in the AUTOSAR FlexRay State Manager specification.

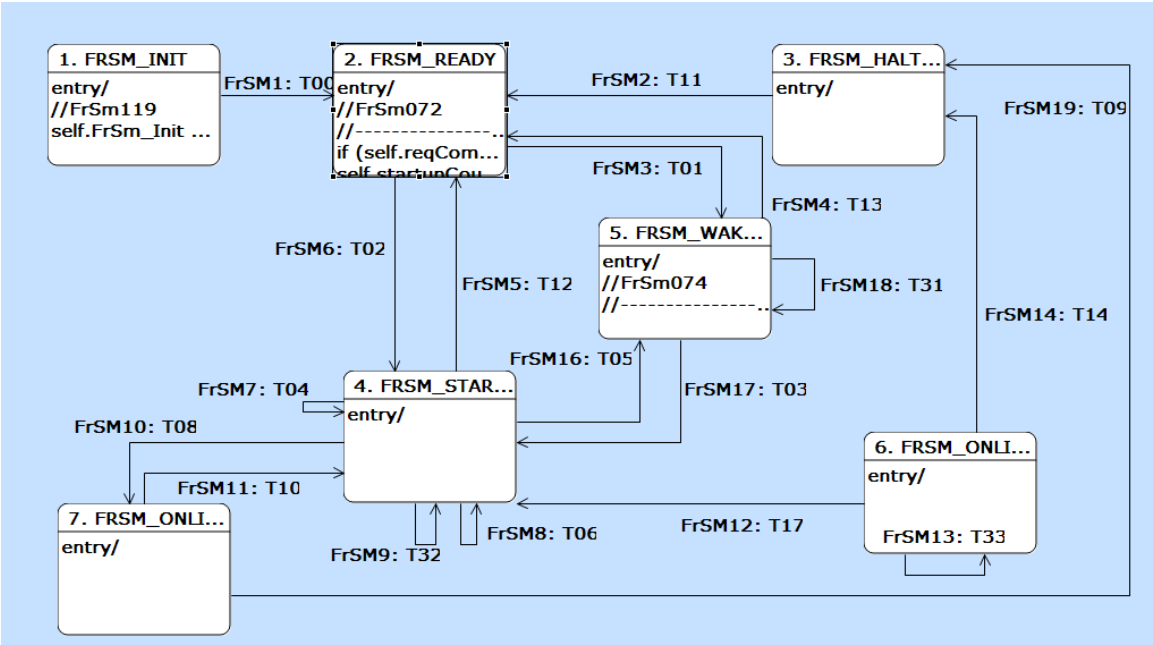

Figure 34 FlexRay xtUML State Machine

The xtUML state machine as shown in Figure 34 implements the states INIT, READY, HALT_REQ, WAKEUP, STARTUP, ONLINE and ONLINE_Passive as documented in the AUTOSAR FlexRay State manager specification. All specification transitions are implemented in xtUML to match the specification. Each state/transition actions in the specification are also mapped to xtUML action language to manipulate the variables in accordance to the specification as shown in Figure 13.

State hosts action language that checks conditions and takes actions in accordance to the specification. Figure 35 shows the OAL (Object Action Language) in the FRSM_READY state that checks variables and triggers transitions T01a, T01b, T01c and required actions once the conditions are satisfied. In xtUML, the state change is triggered via events that trigger the specified transition.

FrSM_Comp | FrSM::FRSM_READY

```
//FrSm072
//----------------------------------------
if (self.reqComMode == ComM_ModType::FullCom) AND
   (self.WUReason == WUReason_Type::NO_WU_BY_BUS) AND
   (self.FrSMIsWakeupEcu == true) AND
   (self.FrSMIsDualChannelNode == false)
self.startupCounter =1;
self.wakeupType = wakeup_Type::SingleChannelWakeup;
self.wakeupTransmitted = false;
self.t1 = true;
self.t3=true;
generate FrSM3 to self;
end if;

if (self.reqComMode == ComM_ModType::FullCom) AND
   (self.WUReason == WUReason_Type::NO_WU_BY_BUS) AND
   (self.FrSMIsWakeupEcu == true) AND
   (self.FrSMIsDualChannelNode == true)
self.startupCounter =1;
self.wakeupType = wakeup_Type::DualChannelWakeup;
self.wakeupTransmitted = false;
self.t1 = true;
self.t3=true;
generate FrSM3 to self;
end if;

if (self.reqComMode == ComM_ModType::FullCom) AND
   (self.WUReason == WUReason_Type::PARTIAL_WU_BY_BUS) AND
   (self.FrSMIsWakeupEcu == true)
self.startupCounter =1;
self.wakeupType = wakeup_Type::DualChannelWakeupForward;
self.wakeupTransmitted = false;
self.t3=true;
generate FrSM3 to self;
end if;
```

Figure 35 xtUML Implementation of FrSm072

Functions are mapped to operations in the FrSM class. Each operation manipulates the state variables and takes the actions specified in the specification. Figure 36 shows an example of FrSM_Init implementation in xtUML as documented in the specification where the configuration parameters are stored and accessible via other functions and a transition is made to FRSM_READY once the initialization has taken place.

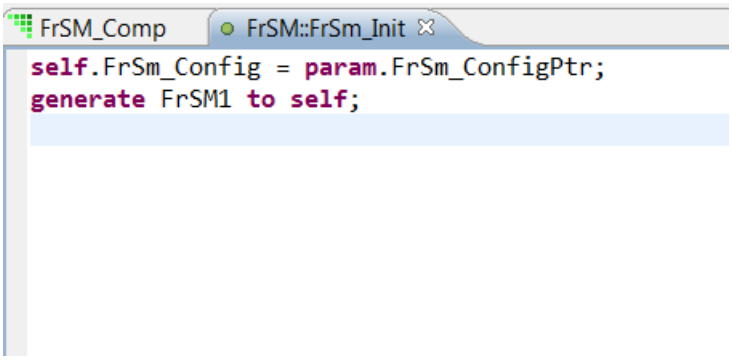

Figure 36 FrSM Initialization in xtUML

Once the design is complete in xtUML, a build is triggered in C/C++ perspective to launch the model compiler and generate SAL model counterpart. Figure 37 Shows generated output after a successful build. Figure 38 shows the console output during SAL generation/build.

At this point in the flow, xtUML model of the design under test has been automatically compiled into a formal SAL model and theorems. Sample SAL output is documented in APPENDIX B.

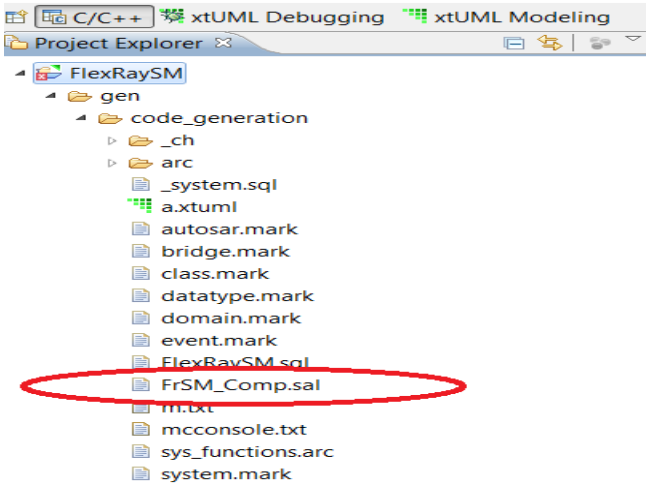

Figure 37 Generated SAL Model

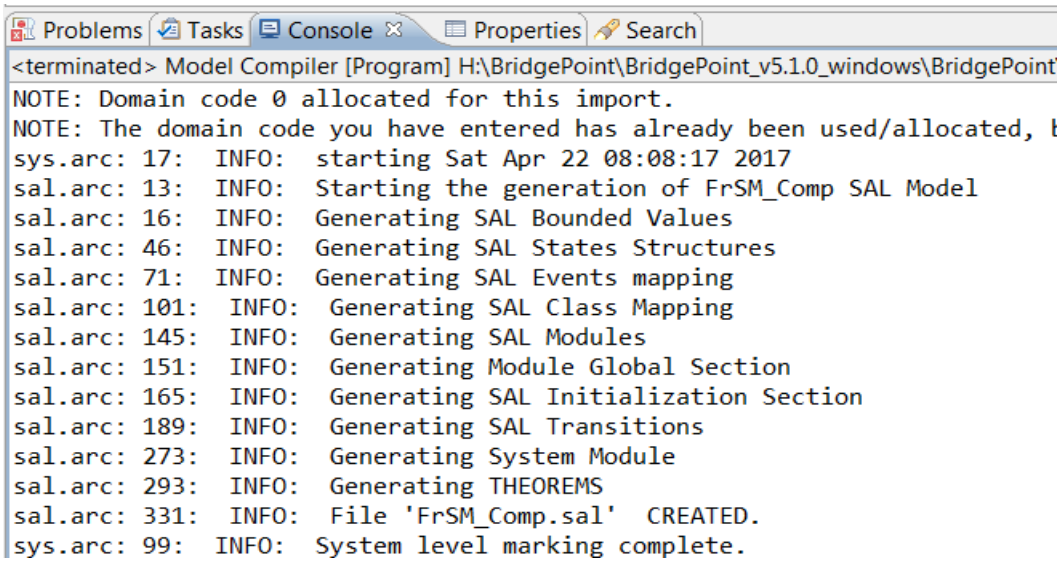

Figure 38 SAL Generation

6.1.2 Model Checking Results

This section details the results of running the model checkers on the generated SAL model. The intention is to show sample design defects that can be uncovered via the model checkers in the early design stage. FlexRay SAL model shall be checked via a SAL compiler to validate syntax, SAL deadlock checker to validate that the state machine has no deadlock state, and finally a BDD checker to verify theorems (Boundary check and requirement compliance).

6.1.2.1 SAL Model Compilation

The SAL compiler is triggered on the generated SAL model to verify that the generated SAL syntax is correct and that the formal model is complete. This step will fail if there are non-bounded variables, non-initialized variables or syntax defects. We have left an uninitialized variable in the xtUML model to check that the SAL model compiler will report any un-initialized variable. In the FlexRay State manager xtUML model, we left the configuration class member as un-initialized as shown in Figure 39

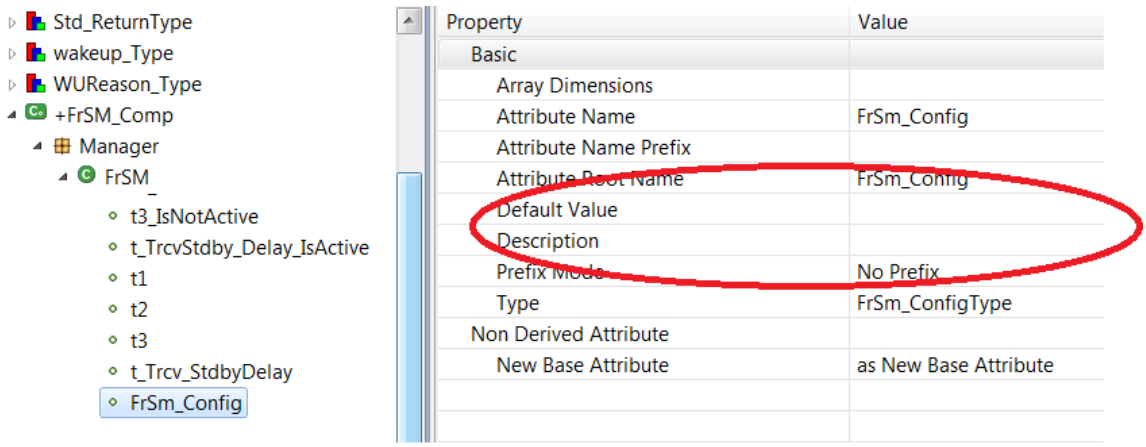

Figure 39 Un-initialized Data Member

The output below shows the result of the compilation which basically can be summarized as mismatch between the defined FrSM_Comp type and the initiation instance of REC_FrSM as elements in the structure are not initialized. In the case below, the variable FrSM_Config was not part of the initialization in xtUML and thus the SAL compiler generated the error below.

```
$ sal-wfc FrSM_Comp.sal --verbose=3
importing context "FrSM_Comp"...
parsing SAL file "FrSM_Comp.sal"...
creating abstract syntax tree for context "FrSM_Comp"...
  ast generation time: 0.0 secs
type checking context "FrSM_Comp"...
Error:    [Context:    FrSM_Comp,    line(173),    column(2)]:
Incompatible types in assignment.
The following types are incompatible:
FrSM_Comp!REC_FrSM

[# AllChannelIsAwake: bool,
   FrSMCheckWakeupReason: bool,
   FrSMDelayStartupWithoutWakeup: bool,
   FrSMIsColdstartEcu: bool,
   FrSMIsDualChannelNode: bool,
   FrSMIsWakeupEcu: bool,
   FrSMNumWakeupPatterns: nat,
   FrSMStartupRepetitions: nat,
   FrSMStartupRepetitionsWithWakeup: nat,
   WUReason: FrSM_Comp!WUReason_type,
```

```
   busTrafficDetected: bool,
   reqComMode: FrSM_Comp!ComM_ModType,
   startupCounter: nat,
   t1: bool,
   t1_IsActive: bool,
   t2: bool,
   t3: bool,
   t3_IsNotActive: bool,
   t_TrcvStdby_Delay_IsActive: bool,
   t_Trcv_StdbyDelay: nat,
   wakeupCounter: nat,
   wakeupTransmitted:  bool,wakeupType:  FrSM_Comp!wakeup_Type
#]
```

Once the issue was corrected in xtUML, the SAL compiler compiled the file successfully and gave the below generated output.

```
$ sal-wfc FrSM_Comp.sal --verbose=3
importing context "FrSM_Comp"...
parsing SAL file "FrSM_Comp.sal"...
creating abstract syntax tree for context "FrSM_Comp"...
  ast generation time: 0.0 secs
type checking context "FrSM_Comp"...
  type-checker time: 0.0 secs
Ok.
total execution time: 0.0 secs
```

6.1.2.2 SAL Deadlock Checker

The SAL deadlock checker is triggered on the generated SAL model to verify that the state machine has no deadlock state. The deadlock checker shall be executed before the SAL model checker as the theorems cannot be verified if a tree can only be built with a deadlock state. Our first run of the deadlock checker against the FlexRay State Manager SAL model revealed a set of variable assignments that lead up to a deadlock state in our state machine. The Deadlock checker output below shows the set of assignments that lead up to being stuck in FRSM_INIT State in the FlexRay State manager state machine. In summary, the model compiler reports a set of variable

assignments that lead to being in FRSM_INIT state and deadlocking there given the reported set of variable assignments.

```
$ sal-deadlock-checker FrSM_Comp MOD_FrSM --verbose=3
detecting deadlock states...
  computing set of reachable states...
  iteration: 1
  frontier lower bound: 90 nodes, upper bound: 90 nodes
  using frontier with 90 nodes
  total bdd node count: 876
  iteration: 2
  frontier lower bound: 87 nodes, upper bound: 93 nodes
  using frontier with 87 nodes
  total bdd node count: 978
  number of visited states: 19.0
  time to compute set of reachable states: 0.0 secs
  deadlock state detection time: 0.0 secs
Total number of deadlock states: 18.0
Deadlock states:
State 1
--- System Variables (assignments) ---
FrSM.AllChannelIsAwake = false
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = false
FrSM.FrSMIsWakeupEcu = false
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
```

```
FrSM.busTrafficDetected = false
FrSM.reqComMode = NoCom
FrSM.startupCounter = 2
FrSM.t1 = false
FrSM.t1_IsActive = false
FrSM.t2 = false
FrSM.t3 = false
FrSM.t3_IsNotActive = false
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = NoWakeup
EVT = EVT_T06
FrSM_State = ST_FRSM_INIT
----------------------------
total execution time: 0.281 secs
```

6.1.2.3 SAL Model Checker

In this run, the experiments will aim to verify that any violation to the specification boundary conditions are detected and any incompliance to specification in the state machine is detected as well. Initially, we introduced a defect in the design where startupCounter is incremented infinitely in the FRSM_READY state machine. According to the specification (Requirement FrSm033 as shown in 5.1.1.2), the value should not exceed 255. We launched the model checker against the SAL model and the automatically generated SAL LTL (Linear Temporal Logic) theorem below to validate the requirement.

```
THEOREM system |- G(FrSM.startupCounter <= 255 AND FrSM.startupCounter >= 0 );
```

The above theorem map textually to,

Globally, it is always true that startupCounter
is less or equal to 255 and greater than or equal to 0.

We also embedded an invalid statement in FRSM_READY state that increments the startup counter. The model checker captured the violation and indicated all the

variable assignments/state paths that lead up to the violation. A snapshot of the violation is shown below:

```
$ sal-smc FrSM_Comp Safe_FrSM_033
Counterexample:
=======================
Path
=======================
Step 0:
--- System Variables (assignments) ---
FrSM.AllChannelIsAwake = false
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = false
FrSM.FrSMIsWakeupEcu = false
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
```

```
FrSM.busTrafficDetected = false
FrSM.reqComMode = NoCom
FrSM.startupCounter = 0
FrSM.t1 = false
FrSM.t1_IsActive = false
FrSM.t2 = false
FrSM.t3 = false
FrSM.t3_IsNotActive = false
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = NoWakeup
EVT = EVT_TOO
FrSM_State = ST_FRSM_INIT
------------------------
Transition Information:
(module instance at [Context: FrSM_Comp, line(416), column(8)]
  (module instance at [Context: FrSM_Comp, line(409),
column(17)]
    transition at [Context: FrSM_Comp, line(396),
column(10)]))
------------------------
Step 1:
--- System Variables (assignments) ---
FrSM.AllChannelIsAwake = false
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = true
FrSM.FrSMIsWakeupEcu = true
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
FrSM.busTrafficDetected = false
FrSM.reqComMode = FullCom
FrSM.startupCounter = 0
```

```
FrSM.t1 = false
FrSM.t1_IsActive = false
FrSM.t2 = false
FrSM.t3 = false
FrSM.t3_IsNotActive = false
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = NoWakeup
EVT = EVT_T01
FrSM_State = ST_FRSM_READY
------------------------
Transition Information:
(module instance at [Context: FrSM_Comp, line(416), column(8)]
  (module instance at [Context: FrSM_Comp, line(409),
column(17)]
    transition at [Context: FrSM_Comp, line(334),
column(10)]))
------------------------
Step 2:
--- System Variables (assignments) ---
FrSM.AllChannelIsAwake = false
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = true
FrSM.FrSMIsWakeupEcu = true
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
FrSM.busTrafficDetected = false
FrSM.reqComMode = FullCom
FrSM.startupCounter = 1
FrSM.t1 = true
FrSM.t1_IsActive = false
FrSM.t2 = false
FrSM.t3 = true
```

```
FrSM.t3_IsNotActive = false
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = DualChannelWakeup
EVT = EVT_T01
FrSM_State = ST_FRSM_READY
------------------------
Transition Information:(module instance at [Context:
FrSM_Comp, line(416), column(8)]
  (module instance at [Context: FrSM_Comp, line(409),
column(17)]
    transition at [Context: FrSM_Comp, line(334),
column(10)]))
------------------------
Step 3:
--- System Variables (assignments) ---
FrSM.AllChannelIsAwake = false
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = true
FrSM.FrSMIsWakeupEcu = true
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
FrSM.busTrafficDetected = false
FrSM.reqComMode = FullCom
FrSM.startupCounter = 2
FrSM.t1 = true
FrSM.t1_IsActive = false
FrSM.t2 = false
FrSM.t3 = true
FrSM.t3_IsNotActive = false
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
```

```
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = DualChannelWakeup
EVT = EVT_T01
FrSM_State = ST_FRSM_READY
------------------------
.
.
Transition Information:
(module instance at [Context: FrSM_Comp, line(416), column(8)]
  (module instance at [Context: FrSM_Comp, line(409),
column(17)]
    transition at [Context: FrSM_Comp, line(334),
column(10)]))
------------------------
Step 257:
--- System Variables (assignments) ---
FrSM.AllChannelIsAwake = false
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = true
FrSM.FrSMIsWakeupEcu = true
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
FrSM.busTrafficDetected = false
FrSM.reqComMode = FullCom
FrSM.startupCounter = 256
FrSM.t1 = true
FrSM.t1_IsActive = false
FrSM.t2 = false
FrSM.t3 = true
FrSM.t3_IsNotActive = false
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = DualChannelWakeup
```

```
EVT = EVT_T01
FrSM_State = ST_FRSM_READY
```

Once the defect was removed from xtUML and Model compiler was launched to regenerate the fixed SAL model, the model checker reported that the theorem is proven as shown below:

```
$ sal-smc FrSM_Comp Safe_FrSM_033 --verbose=1
importing context "FrSM_Comp"...
parsing SAL file "FrSM_Comp.sal"...
creating abstract syntax tree for context "FrSM_Comp"...
type checking context "FrSM_Comp"...
number of system variables: 86, number of auxiliary variables:
5
converting flat module to BDD representation (initial states,
and transition relation)...
proving invariant or producing counterexample using BDDs...
  using forward search
proved.
total execution time: 0.328 secs
```

The second experiment was to validate that conditions that should take place for the state machine to be in FRSM_READY state are correct in the design. The requirement is captured on the xtUML model in the state as per the specification as shown in Figure 40 FrSM Requirement 073 in xtUML.

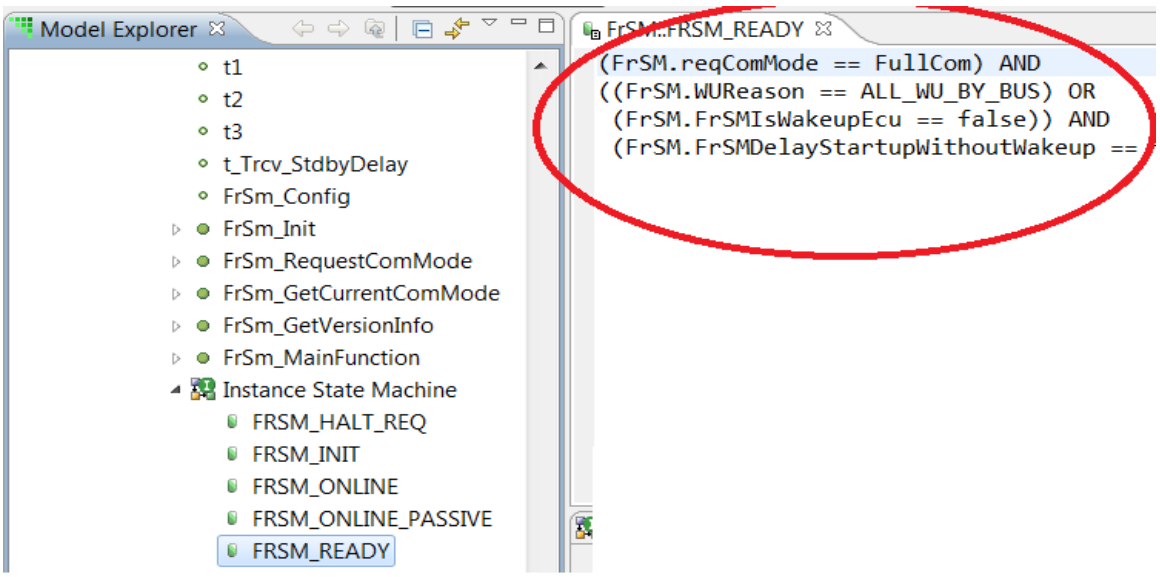

Figure 40 FrSM Requirement 073 in xtUML

The model compiler generated an LTL theorem in the SAL model that maps to the captured state level requirement expressed in xtUML as shown below:

THEOREM system |- G(FrSM.reqComMode = FullCom AND ((FrSM.WUReason = ALL_WU_BY_BUS) OR (FrSM.FrSMIsWakeupEcu = FALSE)) AND (FrSM.FrSMDelayStartupWithoutWakeup = FALSE) => G(FrSM_State = ST_FRSM_READY));

The model compiler generated an LTL theorem in the SAL model that maps to the captured state level requirement expressed in xtUML as shown below:

Globally, it is always true that when communication mode = 'FullCom' AND (WUReason= 'ALL_WU_BY_BUS' OR is Wakeup ECU flag is false) AND Delay startup with wakeup flag is false then globally, FlexRay State should be FRSM_READY.

The BDD (Binary Decision Diagram) based model checker initially proved the above theorem as shown below:

```
$ sal-smc FrSM_Comp Safe_FrSM_073 --verbose=1
importing context "FrSM_Comp"...
parsing SAL file "FrSM_Comp.sal"...
creating abstract syntax tree for context "FrSM_Comp"...
type checking context "FrSM_Comp"...
number of system variables: 88, number of auxiliary variables:
5
converting flat module to BDD representation
proving invariant or producing counterexample using BDDs...
proved.
total execution time: 0.437 secs
```

We introduced a defect in the xtUML model where the initialization state sets the same conditions and transitions to FRSM_WAKEUP state in violation to the specification, which mandates that the FlexRay Manager state machine should be in READY State given these conditions/assignments. We ran the model checker that reported successfully the counter example/violation shown below which clearly shows a violation against the above theorem as the conditions lead up to being in both the Wakeup followed by the startup states:

```
$ sal-smc FrSM_Comp Safe_FrSM_073
Counterexample:
=======================
Path
=======================
Step 0:
--- System Variables (assignments) ---
```

```
ba-pc!1 = 2
FrSM.AllChannelIsAwake = false
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = false
FrSM.FrSMIsWakeupEcu = false
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
FrSM.busTrafficDetected = false
FrSM.reqComMode = NoCom
FrSM.startupCounter = 0
FrSM.t1 = false
FrSM.t1_IsActive = false
FrSM.t2 = false
FrSM.t3 = false
FrSM.t3_IsNotActive = false
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = NoWakeup
EVT = EVT_TOO
FrSM_State = ST_FRSM_INIT
------------------------
Transition Information:
(module instance at [Context: FrSM_Comp, line(408), column(8)]
  (module instance at [Context: FrSM_Comp, line(403),
column(17)]
    transition at [Context: FrSM_Comp, line(388),
column(10)]))
------------------------
Step 1:
--- System Variables (assignments) ---
ba-pc!1 = 2
FrSM.AllChannelIsAwake = false
```

```
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = false
FrSM.FrSMIsWakeupEcu = false
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
FrSM.busTrafficDetected = false
FrSM.reqComMode = FullCom
FrSM.startupCounter = 0
FrSM.t1 = false
FrSM.t1_IsActive = false
```

```
FrSM.t2 = false
FrSM.t3 = false
FrSM.t3_IsNotActive = false
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = NoWakeup
EVT = EVT_T03
FrSM_State = ST_FRSM_WAKEUP
Transition Information:
(module instance at [Context: FrSM_Comp, line(408), column(8)]
  (module instance at [Context: FrSM_Comp, line(403),
column(17)]
    transition at [Context: FrSM_Comp, line(234),
column(10)]))
------------------------
Step 2:
--- System Variables (assignments) ---
ba-pc!1 = 1
FrSM.AllChannelIsAwake = false
FrSM.FrSMCheckWakeupReason = false
FrSM.FrSMDelayStartupWithoutWakeup = false
FrSM.FrSMIsColdstartEcu = false
FrSM.FrSMIsDualChannelNode = false
FrSM.FrSMIsWakeupEcu = false
FrSM.FrSMNumWakeupPatterns = 0
FrSM.FrSMStartupRepetitions = 0
FrSM.FrSMStartupRepetitionsWithWakeup = 0
FrSM.WUReason = NO_WU_BY_BUS
FrSM.busTrafficDetected = false
FrSM.reqComMode = FullCom
FrSM.startupCounter = 0
FrSM.t1 = false
FrSM.t1_IsActive = false
FrSM.t2 = false
FrSM.t3 = false
FrSM.t3_IsNotActive = false
```

```
FrSM.t_TrcvStdby_Delay_IsActive = false
FrSM.t_Trcv_StdbyDelay = 0
FrSM.wakeupCounter = 0
FrSM.wakeupTransmitted = false
FrSM.wakeupType = NoWakeup
EVT = EVT_T03
FrSM_State = ST_FRSM_STARTUP
total execution time: 0.405 secs
```

6.2 Automatic Transmission Controller

The Automatic Transmission controller is responsible for automatically changing gears in the vehicle. It is inevitable that engaging the right gear at right speed is a defined and reliable behavior as any failure could lead to damage in the transmission, which could introduce undefined behavior of the car while speeding. Several automotive recalls were done historically due to faulty automatic transmission. It has been reported that Honda recalled 2.5 million 2005-2010 4 cylinder Accord to update the software that controls the automatic transmission as a sudden shift could lead to damaged shaft bearing. The update was intended to handle sudden gear change transitions to reduce possibility of damage. General Motors also recalled their 2013 Cadillac due to a software defect in the ATC module that introduce a 3-4 second lag in acceleration.

Automatic Transmission Controller specification [104] served as a baseline for implementation of the design in BridgePoint xtUML. The xtUML project consisted of a root package called ATC. The root package contained a component named ATC. The component consisted of another package, Shift Gear that contained two classes, gearController and gearPosition. The classes contain attributes and state machine design as documented in the ATC specification. Figure 41 shows a summary of the xtUML design elements of ATC module. 6.2.1 details the mapping of the specification into the xtUML design.

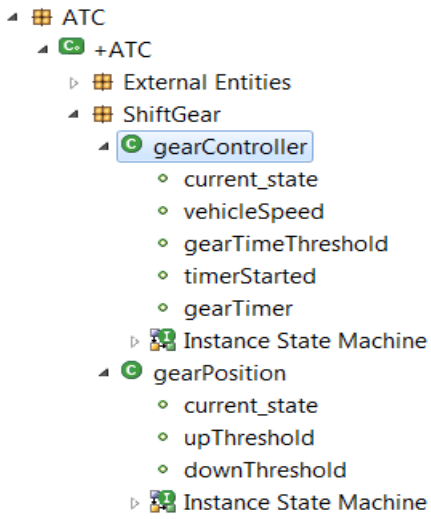

Figure 41 ATC xtUML Design

6.2.1 xtUML Design

The input to the shift logic block as stated in the specification [104] is a vehicle speed and the output is the desired gear number. There are two state machines to keep track of the gear state and the state of the gear selection process.

In gear process selection state machine, while in steady_state, the model compares up threshold and down threshold values to the present vehicle speed to determine if a shift is required. If so, it enters one of the confirm states (upshifting or downshifting), which records the time of entry.

If the vehicle speed no longer satisfies the shift condition, while in the confirm state, the model ignores the shift and it transitions back to steady_state. This prevents extraneous shifts due to noise conditions. If the shift condition remains valid for a duration, the model transitions through the lower junction and, depending on the current gear, it broadcasts one of the shift events. Subsequently, the model again activates steady_state after a transition through one of the central junctions. The shift event, which is broadcast to the gear_selection state, activates a transition to the appropriate new gear [104].

The ATC module specification was mapped into xtUML design. Variables documented in the ATC specification were mapped into class attributes of gearController and GearPosition classes' definition in xtUML. xtUML has been extended to record the default value and max value for each variable so that it can be used to generate boundary conditions theorems in SAL notation.

ATC state machine, transitions and states were also implemented in xtUML in accordance to ATC specification. Figure 42 and Figure 43 show the xtUML implementation of the ATC state machines. Each state checks for the conditions (vehicle speed against current gear position) and triggers actions (switch to proper gear) as documented in the specification.

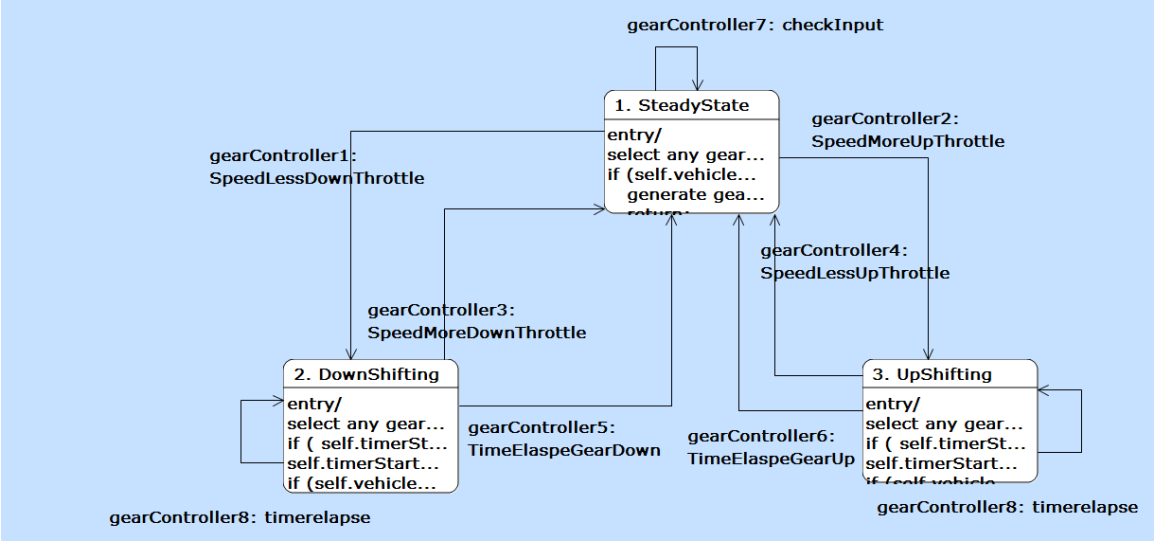

Figure 42 Gear Controller State Machine in xtUML

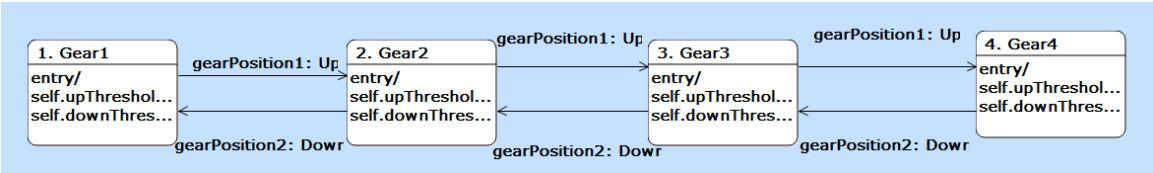

Figure 43 Gear Position State Machine in xtUML

Every ATC state machine state hosts action language that checks conditions and takes actions in accordance to the specification [104]. Figure 44 shows the OAL (Object Action Language) in the Steady State that checks vehicle speed against the threshold to take an action to either transition to down shifting, up shifting or stay at steady

state. In xtUML, the state change is triggered via events that trigger the specified transition.

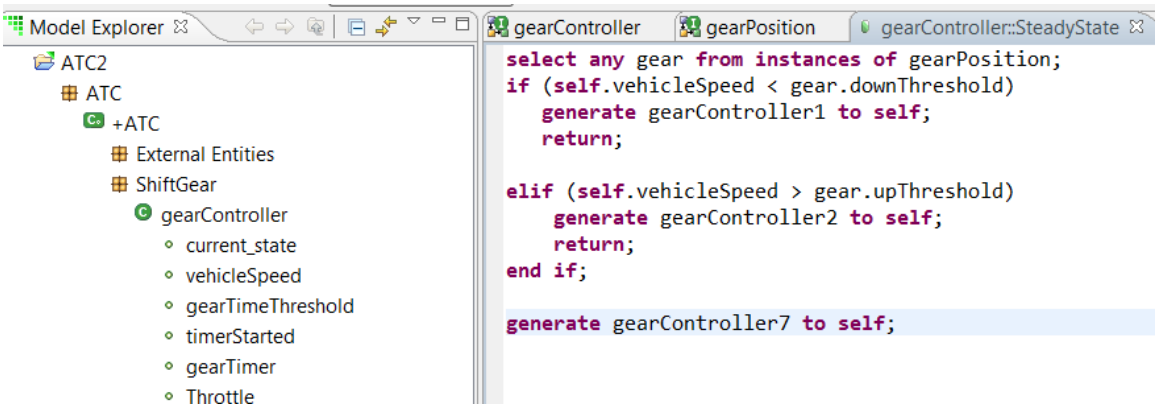

Figure 44 Steady State Action in xtUML

Once the ATC design is complete in xtUML, a build is triggered in C/C++ perspective to launch the model compiler and generate ATC SAL model counterpart. Figure 45 shows generated output after a successful build. Figure 46 shows the console output during SAL generation/build of the SAL ATC model.

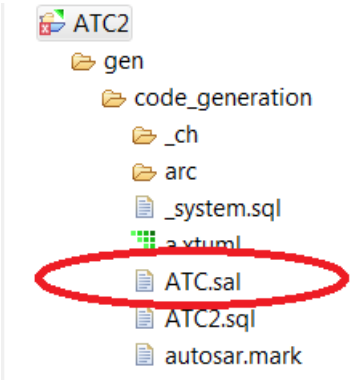

Figure 45 ATC SAL Model Generation

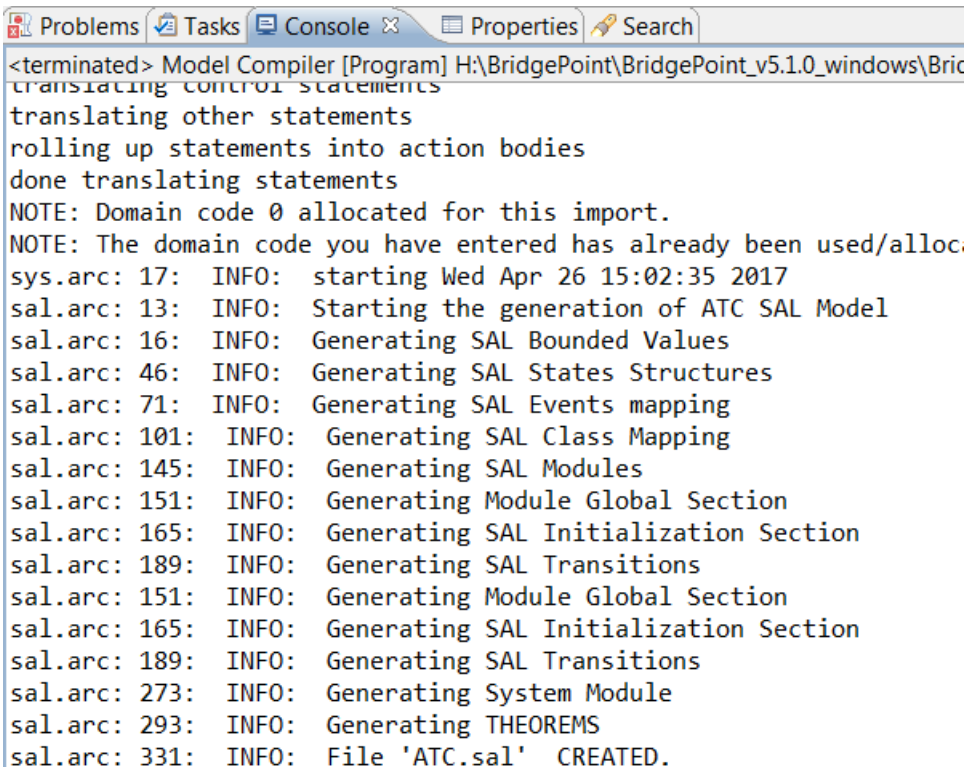

Figure 46 ATC Generation Console Output

At this point in the flow, xtUML model of the design under test has been automatically compiled into a formal SAL model and theorems. ATC SAL model output is documented in APPENDIX B.

6.2.2 Model Checking Results

This section details the results of running the model checkers on the generated ATC SAL model. The intention is to show sample design defects that can be uncovered via the model checkers in the early design stage. ATC model shall be checked via a SAL compiler to validate syntax, SAL deadlock checker to validate that the ATC state machine has no deadlock state, and finally a BDD (Binary Decision Diagram) based checker to verify generated theorems (Boundary check and requirement compliance).

6.2.2.1 SAL Model Compilation

The SAL compiler is triggered on the generated ATC SAL model to verify that the generated SAL syntax is correct and that the ATC formal model is complete. This step will fail if there are non-bounded variables, non-initialized variables or syntax

defects as shown in previous section. The output below shows the result of successful compilation on the generated ATC file.

```
$ sal-wfc ATC.sal --verbose=1
importing context "ATC"...
parsing SAL file "ATC.sal"...
creating abstract syntax tree for context "ATC"...
type checking context "ATC"...
Ok.
total execution time: 0.0 secs
```

6.2.2.2 SAL Deadlock Checker

The SAL deadlock checker is triggered on the generated SAL model to verify that the ATC state machine has no deadlock states. The deadlock checker shall be executed before the BDD model checker as the theorems cannot be verified if a BDD tree can only be built with a deadlock state. Our first run of the deadlock checker against the ATC SAL model revealed a set of variable assignments that lead up to a deadlock state in our state machine. The Deadlock checker output below shows the set of assignments that lead up to being stuck in ST_UPSHIFTING in the gear controller state machine given a specific assignments as shown below.

```
$ sal-deadlock-checker ATC.sal system --verbose==1
Total number of deadlock states: 1.0
Deadlock states:
State 1
--- System Variables (assignments) ---
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = true
CONT.vehicleSpeed = 240
EVT = EVT_UP
PState = ST_POSITION4
Position.downThreshold = 71
Position.upThreshold = 100
State = ST_UPSHIFTING
-----------------------------
```

6.2.2.3 SAL Model Checker

In this run, the experiments will aim to verify that any violation to the specification boundary conditions are detected and any incompliance against ATC specification in the ATC state machine is detected as well. Initially, we introduced a defect in the design where it is possible to be in Position1 in the gear position state machine while vehicle speed is 22 while setting up an xtUML requirement that gear position can only be in Position 1 if vehicle speed is between 0 and 21. We launched the model checker against the SAL model and the automatically generated SAL LTL (Linear Temporal Logic) theorem below to validate the requirement.

```
Req1_Th1: THEOREM system |- AG(PState = ST_POSITION1 =>
AF(CONT.vehicleSpeed > 0 AND  CONT.vehicleSpeed <= 21));
```

The above theorem map textually to,

```
Globally, it is always true that Gear Position is in Position 1
For all paths when vehicle speed is greater than 0 and less or equal to 21
```

We introduced a bug whereby it is possible to be in Position1 while vehicle speed is 22 in xtUML ATC model. The model checker captured the violation and indicated all the variable assignments/state paths that lead up to the violation. A snapshot of the violation is shown below:

```
$ sal-smc ATC Req1_Th1 --verbose=1
importing context "ATC"...
parsing SAL file "ATC.sal"...
creating abstract syntax tree for context "ATC"...
type checking context "ATC"...
number of system variables: 44, number of auxiliary variables:
8
converting flat module to BDD representation (initial states,
and transition relation)...
proving or producing counterexample using BDDs...
Counterexample:
=======================
Path
```

```
========================
Step 0:
--- System Variables (assignments) ---
ba-pc!1 = 1
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = false
CONT.vehicleSpeed = 20
EVT = EVT_CHECKINPUT
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_STEADY
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    else transition at [Context: ATC, line(138), column(9)]))
------------------------
Step 1:
--- System Variables (assignments) ---
ba-pc!1 = 1
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = false
CONT.vehicleSpeed = 21
EVT = EVT_CHECKINPUT
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_STEADY
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(116), column(7)]))
------------------------
```

```
Step 2:
--- System Variables (assignments) ---
ba-pc!1 = 1
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = false
CONT.vehicleSpeed = 21
EVT = EVT_SPEEDMOREUPTHROTTLE
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_STEADY
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(119), column(7)]))
------------------------
Step 3:
--- System Variables (assignments) ---
ba-pc!1 = 1
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = false
CONT.vehicleSpeed = 21
EVT = EVT_SPEEDMOREUPTHROTTLE
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_UPSHIFTING
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(122), column(7)]))
------------------------
Step 4:
--- System Variables (assignments) ---
ba-pc!1 = 1
```

```
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = true
CONT.vehicleSpeed = 21
EVT = EVT_SPEEDMOREUPTHROTTLE
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_UPSHIFTING
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(125), column(7)]))
------------------------
Step 5:
--- System Variables (assignments) ---
ba-pc!1 = 1
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = false
CONT.vehicleSpeed = 21
EVT = EVT_TIMEELASPEGEARUP
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_UPSHIFTING
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(129), column(7)]))
------------------------
Step 6:
--- System Variables (assignments) ---
ba-pc!1 = 1
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
```

```
CONT.timerStarted = false
CONT.vehicleSpeed = 21
EVT = EVT_UP
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_UPSHIFTING
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(122), column(7)]))
Step 7:
--- System Variables (assignments) ---
ba-pc!1 = 1
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = true
CONT.vehicleSpeed = 21
EVT = EVT_UP
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_UPSHIFTING
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    else transition at [Context: ATC, line(138), column(9)]))
------------------------
Step 8:
--- System Variables (assignments) ---
ba-pc!1 = 1
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = true
CONT.vehicleSpeed = 22
```

```
EVT = EVT_CHECKINPUT
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_UPSHIFTING
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(132), column(7)]))
------------------------
Step 9:
--- System Variables (assignments) ---
ba-pc!1 = 0
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = true
CONT.vehicleSpeed = 22
EVT = EVT_CHECKINPUT
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_STEADY
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(116), column(7)]))
------------------------
Step 10:
--- System Variables (assignments) ---
ba-pc!1 = 0
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = true
CONT.vehicleSpeed = 22
EVT = EVT_SPEEDMOREUPTHROTTLE
PState = ST_POSITION1
Position.downThreshold = 0
```

```
Position.upThreshold = 20
State = ST_STEADY
========================
Begin of Cycle
========================
Step 10:
--- System Variables (assignments) ---
ba-pc!1 = 0
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = true
CONT.vehicleSpeed = 22
EVT = EVT_SPEEDMOREUPTHROTTLE
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_STEADY
------------------------
Transition Information:
(module instance at [Context: ATC, line(222), column(20)]
  (module instance at [Context: ATC, line(207), column(19)]
    transition at [Context: ATC, line(116), column(7)]))
------------------------
Step 11:
--- System Variables (assignments) ---
ba-pc!1 = 0
CONT.gearTimeTHreshold = 10
CONT.gearTimer = 20
CONT.timerStarted = true
CONT.vehicleSpeed = 22
EVT = EVT_SPEEDMOREUPTHROTTLE
PState = ST_POSITION1
Position.downThreshold = 0
Position.upThreshold = 20
State = ST_STEADY
total execution time: 1.482 secs
```

Once the defect was removed from xtUML and Model compiler was launched to regenerate the fixed SAL model, the model checker reported that the theorem is proven as shown below:

```
$ sal-smc ATC Req1_Th1 --verbose=1
importing context "ATC"...
parsing SAL file "ATC.sal"...
creating abstract syntax tree for context "ATC"...
type checking context "ATC"...
number of system variables: 44, number of auxiliary variables:
8
converting flat module to BDD representation (initial states,
and transition relation)...
proving or producing counterexample using BDDs...
proved.
total execution time: 0.124 secs
```

6.3 WatchDog State Manager Results

AUTOSAR WatchDog Manager SWS served as a baseline for implementation of the design in BridgePoint xtUML. It is a critical module in the AUTOSAR BSW Services stack that provide services for monitoring the timing and the correctness of execution of an entity in the application or basic software of AUTOSAR stack. It avoids crash of the system via detecting anomalies during supervision and taking configurable actions when the anomalies happen. Therefore, verifying this module is crucial in automotive and shows that all managers in the services layer could be verified at the design stage in a similar manner.

The xtUML project consisted of a root package called WdgM. The root package containes a WdgM component. The root component has a Manager package which hosts all data types as documented in the specification and a class named WdgM. The class contains attributes, functions and state machine designs as documented in the Watchdog manager specification. Figure 47 shows a summary of the xtUML design elements of Watchdog manager module. Section 6.3.1 details the mapping of the specification into the xtUML design.

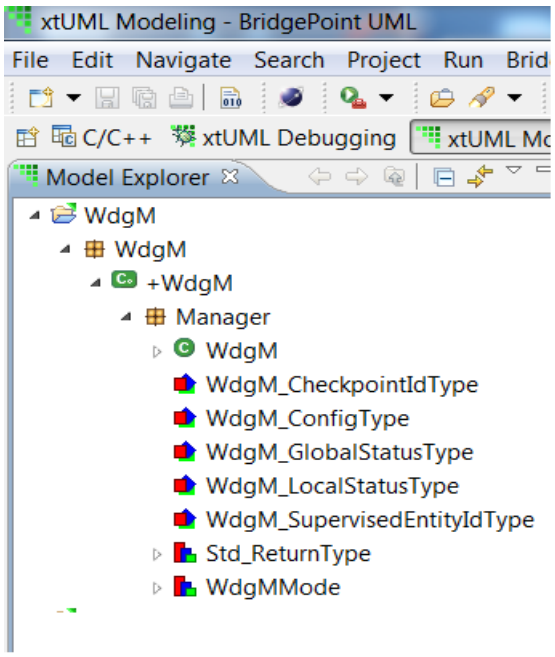

Figure 47 WatchDog Manager xtUML Design

6.3.1 xtUML Design

WatchDog Manager AUTOSAR specification [102] describes the module in the following order:

1- Defines all states in the Watchdog Local state machine (DEACTIVATED, OK, FAILED, EXPIRED) as shown in Figure 14.

2- Defines all transition preconditions and actions in the state machine. An example is shown in Figure 15 where specification mandates that a transition from OK to Expired shall be triggered if at least one supervised entity alive flag is incorrect and a fault tolerance of zero is configured OR at least one deadline /logical supervision value of supervised entity is incorrect.

3- DataTypes are specified. An example is shown in Figure 48 where SupervisedEntityIdType is a uint16 or uint8 in the specification.

4- Function definitions and actions to be taken inside functions. An example is shown in Figure 49 where specification indicates that a function WdgM_Init needs to be defined that returns void and accepts a configuration parameter of type pointer to WdgM_ConfigType. The intent

of this function is to initialize the manager and set the configuration parameters.

Name:	`WdgM_SupervisedEntityIdType`	
Type:	`uint8, uint16`	
Range:	`0-<Number of Supervised Entities>`	The range of valid IDs depends on the number of configured Supervised Entities and on the chosen platform type.
Description:	This type identifies an individual Supervised Entity for the Watchdog Manager.	

Figure 48 WdgM_SupervisedEntityId Type Definition

[WDGM151] ⌈

Service name:	WdgM_Init	

Syntax:	`void WdgM_Init(`
	`  const WdgM_ConfigType* ConfigPtr`
	`)`	

Service ID[hex]:	0x00	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	ConfigPtr	Pointer to post-build configuration data
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	Initializes the Watchdog Manager.	

Figure 49 WdgM_Init Function

The module specification was mapped into xtUML design. All defined types were mapped into user defined types in xtUML. An example is WdgMMode user defined type enum that should be defined with the following enum values: SUPERVISION_OK, SUPERVISION_FAILED, SUPERVISION_EXPIRED, SUPERVISION_STOPPED, SUPERVISION_DEACTIVATED. Figure 50 shows the mapping of this user defined type in xtUML.

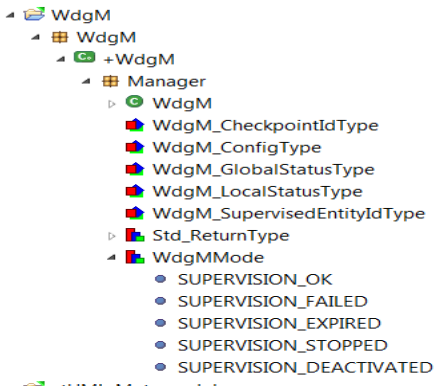

Figure 50 WdgMMode User Defined Type

Additionally, variables documented in the specification were mapped into class attributes of the WdgM class definition in xtUML as shown in Figure 51. xtUML has been extended to record the default value and max value for each variable so that it can be used to generate boundary conditions theorems in SAL notation.

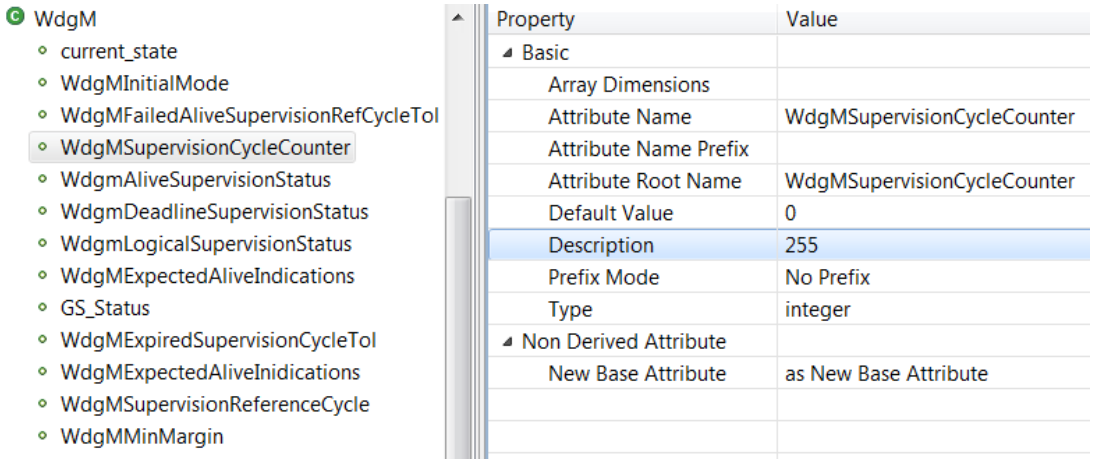

Figure 51 WdgMSupervisionCycleCounter Variable Definition

Watchdog Manager States, state machine and transitions were also implemented in xtUML in accordance to Watchdog local state machine specification document. Figure 52 shows the xtUML implementation of the state machine.

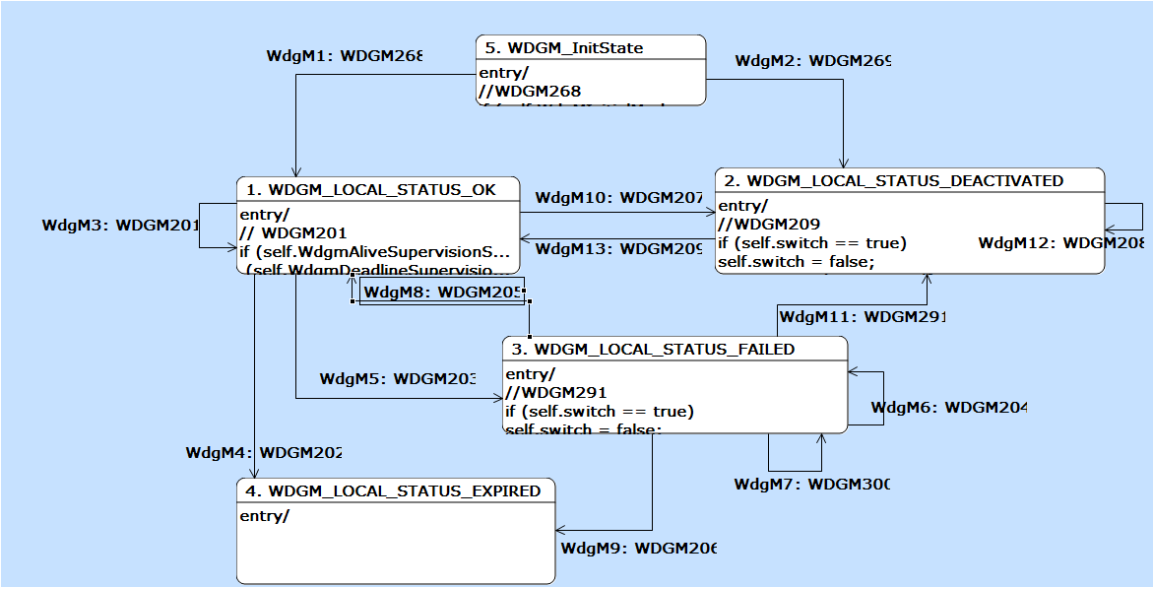

Figure 52 Watchdog Local xtUML State Machine

The xtUML state machine as shown in Figure 34 implements the states INIT, OK, DEACTIVATED, FAILED, and EXPIRED as documented in the AUTOSAR Watchdog manager specification. All specification transitions are implemented in xtUML to match the specification. Each state/transition actions in the specification are also mapped to xtUML action language to manipulate the variables in accordance to the specification as shown in Figure 53.

State hosts action language that checks conditions and takes actions in accordance to the specification. Figure 53 shows the OAL (Object Action Language) in the WdgM Local Status Ok state that checks variables and triggers transitions in accordance to requirements WdgM 201(Stay in State OK given variable set values), WdgM 202(transition to State Expired given variable set values) and WdgM 203(transition to State Failed given variable set values) and required actions once the conditions are satisfied. In xtUML, the state change is triggered via events that trigger the specified transition.

WdgM WdgM::WDGM_LOCAL_STATUS_OK

```
// WDGM201
if (self.WdgmAliveSupervisionStatus == 0) AND
 (self.WdgmDeadlineSupervisionStatus == 0) AND
 (self.WdgmLogicalSupervisionStatus == 0)
generate WdgM3 to self;
return;
end if;


//WDGM203
if (self.WdgmAliveSupervisionStatus != 0)AND
   (self.WdgMFailedAliveSupervisionRefCycleTol > 0) AND
   (self.WdgmDeadlineSupervisionStatus == 0) AND
   (self.WdgmLogicalSupervisionStatus == 0)

self.WdgMSupervisionCycleCounter = self.WdgMSupervisionCycleCounter +1;
generate WdgM5 to self;
return;
//WDGM202
elif (self.WdgMFailedAliveSupervisionRefCycleTol == 0) AND
((self.WdgmAliveSupervisionStatus != 0) OR
 (self.WdgmDeadlineSupervisionStatus != 0) OR
 (self.WdgmLogicalSupervisionStatus != 0))
generate WdgM4 to self;
return;
end if;
```

Figure 53 xtUML Implementation of WdgM201, WdgM203 and WdgM202

Functions are mapped to operations in the WdgM class. Each operation manipulates the state variables and takes the actions specified in the specification. Figure 54 shows an example of WdgM_SetMode implementation in xtUML as documented in the specification where SetMode is accepted when in deactivated state and it is requested to be in OK state and rejected if it is in FAILED state and a request is made to deactivate it.

```
//WDGM209
if ((self.current_state == WDGM_LOCAL_STATUS_DEACTIVATED) AND (Mode = WDGMMode::SUPERVISION_OK)
 self.switch = true;
 return Std_ReturnType::E_OK;
end if;


//WDGM291
if (self.current_state == WDGM_LOCAL_STATUS_FAILED) AND (Mode = WDGMMode::SUPERVISION_DEACTIVATED)
 self.switch = true;
 return Std_ReturnType::E_OK;
end if;

return Std_ReturnType::E_NOT_OK;
```

Figure 54 WdgM setMode function in xtUML

Once the design is complete in xtUML, a build is triggered in C/C++ perspective to launch the model compiler and generate SAL model counterpart. Figure 55 Shows generated output after a successful build. Figure 55 shows the console output during SAL generation/build.

At this point in the flow, xtUML model of the design under test has been automatically compiled into a formal SAL model and theorems. Sample SAL output is documented in APPENDIX B.

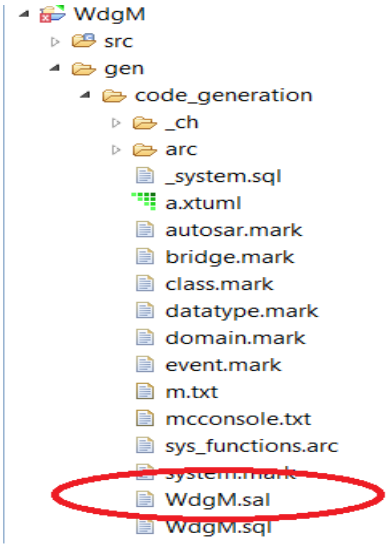

Figure 55 Generated WdgM SAL Model

```
Upgrading translation workspace:  code_generation
Enabling detection of empty handles for component(s) *.
Action statement tracing enabled for component WdgM.
Enabling state transition tracing for component(s) *.
Analyzing model and making optimizations....
Self event queue needed.
6 attributes read
2 attributes written
translating control statements
translating other statements
rolling up statements into action bodies
done translating statements
NOTE: Domain code 0 allocated for this import.
NOTE: The domain code you have entered has already been used/allc
sys.arc: 17:  INFO:  starting Thu May 11 10:39:12 2017
sal.arc: 13:  INFO:  Starting the generation of WdgM SAL Model
sal.arc: 16:  INFO:  Generating SAL Bounded Values
sal.arc: 46:  INFO:  Generating SAL States Structures
sal.arc: 71:  INFO:  Generating SAL Events mapping
sal.arc: 101:  INFO:  Generating SAL Class Mapping
sal.arc: 145:  INFO:  Generating SAL Modules
sal.arc: 151:  INFO:  Generating Module Global Section
sal.arc: 165:  INFO:  Generating SAL Initialization Section
sal.arc: 189:  INFO:  Generating SAL Transitions
sal.arc: 273:  INFO:  Generating System Module
sal.arc: 293:  INFO:  Generating THEOREMS
sal.arc: 331:  INFO:  File 'WdgM.sal'  CREATED.
```

Figure 56 WdgM SAL Generation

6.3.2 Model Checking Results

This section details the results of running the model checkers on the generated SAL model. The intention is to show sample design errors that can be uncovered via the model checkers in the early design stage. Watchdog SAL model shall be checked via a SAL compiler to validate syntax, SAL deadlock checker to validate that the state machine has no deadlock state, and finally a BDD (Binary Decision Diagram) based checker to verify theorems (Boundary check and requirement compliance).

6.3.2.1 SAL Model Compilation

The SAL compiler is triggered on the generated SAL model to verify that the generated SAL syntax is correct and that the formal model is complete. This step will fail if there are non-bounded variables, non-initialized variables or syntax defects. We have left an uninitialized variable in the xtUML model to check that the SAL model compiler will report any un-initialized variable. In the Watchdog manager xtUML model, we left the WdgMFailedAliveSupervisionRefCycleTol class member as un-initialized as shown in Figure 57

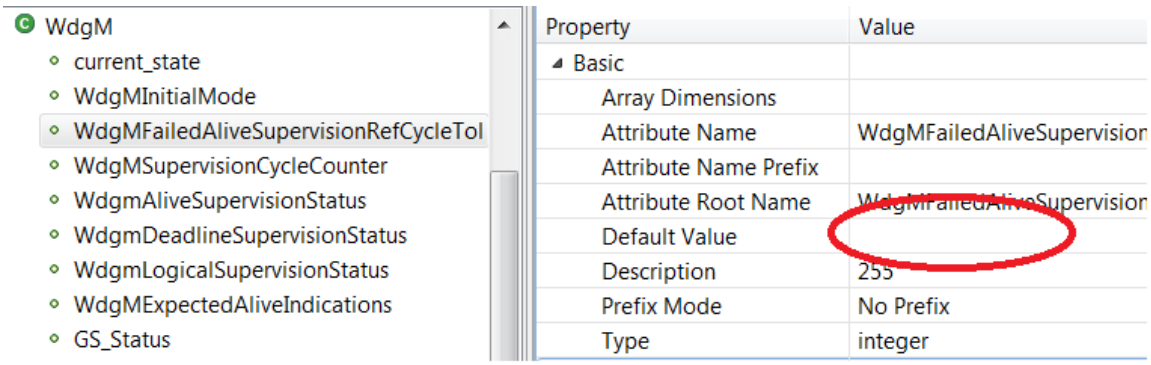

Figure 57 Un-initialized Data Member

The output below shows the result of the compilation which basically can be summarized as mismatch between the defined WdgM type and the initiation instance of REC_WdgM as WdgMFailedAliveSupervisionRefCycleTol in the structure is not initialized which triggered the SAL compiler to generate the error below.

```
$ sal-wfc WdgM.sal --verbose=2
importing context "WdgM"...
parsing SAL file "WdgM.sal"...
creating abstract syntax tree for context "WdgM"...
  ast generation time: 0.0 secs
type checking context "WdgM"...
Error: [Context: WdgM, line(80), column(2)]: Incompatible
types in assignment.
The following types are incompatible:
WdgM!REC_WdgM

[# WdgMExpectedAliveIndications: nat,
   WdgMInitialMode: nat,
   WdgMSupervisionCycleCounter: nat,
   WdgmAliveSupervisionStatus: nat,
   WdgmDeadlineSupervisionStatus: nat,
   WdgmLogicalSupervisionStatus: nat #]
```

Once the issue was corrected in xtUML, the SAL compiler compiled the file successfully and gave the below generated output.

```
$ sal-wfc WdgM.sal --verbose=2
importing context "WdgM"...
parsing SAL file "WdgM.sal"...
creating abstract syntax tree for context "WdgM"...
```

```
  ast generation time: 0.0 secs
type checking context "WdgM"...
  type-checker time: 0.0 secs
Ok.
total execution time: 0.0 secs
```

6.3.2.2 SAL Deadlock Checker

The SAL deadlock checker is triggered on the generated SAL model to verify that the state machine has no deadlock state. The deadlock checker shall be executed before the SAL model checker as the theorems cannot be verified if a tree can only be built with a deadlock state. Our first run of the deadlock checker against the Watchdog local State SAL model revealed a set of variable assignments that lead up to a deadlock state in our state machine. The Deadlock checker output below shows the set of assignments that lead up to being stuck in WdgM_EXPIRED State in the WdgM local State machine. In summary, the model compiler reports a set of variable assignments that lead to being in WDGM_LOCAL_STATUS_EXPIRED state and deadlocking there given the reported set of variable assignments.

```
$ sal-deadlock-checker WdgM MOD_WdgM --verbose=3
importing context "WdgM"...
parsing SAL file "WdgM.sal"...
creating abstract syntax tree for context "WdgM"...
  ast generation time: 0.0 secs
type checking context "WdgM"...
  type-checker time: 0.0 secs
flattening module at [Context: scratch, line(1), column(1)]
converting flat module to BDD representation (initial states,
and transition relation)...
  creating BDD variables...
  computing static variable ordering (minimizing support)...
    collecting state variables dependencies...
  static order time: 0.015 secs
  number of BDD variables: 182
  creating definition section BDDs...
  creating valid state predicate BDDs...
  creating BDD: set of initial states...
  creating BDD: transition relation...
  rearranging clusters...
```

```
  reordering BDD variables...
  transition relation - size: 805 (nodes), number of clusters:
1
  flat-module -> BDD conversion time: 0.187 secs
detecting deadlock states...
  computing set of reachable states...
  iteration: 1
  frontier lower bound: 92 nodes, upper bound: 92 nodes
  using frontier with 92 nodes
  total bdd node count: 1270
  iteration: 2
  frontier lower bound: 92 nodes, upper bound: 95 nodes
  using frontier with 92 nodes
  total bdd node count: 1465
  iteration: 3
  frontier lower bound: 92 nodes, upper bound: 99 nodes
  using frontier with 92 nodes
  total bdd node count: 1620
  iteration: 4
  frontier lower bound: 92 nodes, upper bound: 99 nodes
  using frontier with 92 nodes
  total bdd node count: 1776
  iteration: 5
  frontier lower bound: 92 nodes, upper bound: 99 nodes
  using frontier with 92 nodes
  total bdd node count: 1984
  iteration: 6
  frontier lower bound: 92 nodes, upper bound: 104 nodes
  using frontier with 92 nodes
  total bdd node count: 2194
  number of visited states: 6.0
  time to compute set of reachable states: 0.0 secs
  deadlock state detection time: 0.0 secs
```

```
Total number of deadlock states: 1.0
Deadlock states:
State 1
--- System Variables (assignments) ---
WdgM.WdgMExpectedAliveIndications = 1
WdgM.WdgMFailedAliveSupervisionRefCycleTol = 0
WdgM.WdgMInitialMode = 0
WdgM.WdgMSupervisionCycleCounter = 0
WdgM.WdgmAliveSupervisionStatus = 0
WdgM.WdgmDeadlineSupervisionStatus = 0
WdgM.WdgmLogicalSupervisionStatus = 0
EVT = EVT_WDGM202
WdgM_State = ST_WDGM_LOCAL_STATUS_EXPIRED
-----------------------------
total execution time: 0.187 secs
```

6.3.2.3 SAL Model Checker

In this run, experiments will aim to verify that any violation to the specification boundary conditions are detected and any incompliance to specification in the state machine is detected as well. We will aim to reproduce a design defect that matches the defect identified post production in deployed ASIL B compliant WdgM. We introduced a defect in the design where WdgMFailedAliveSupervisionRefCycleTol is assigned a value outside the specified range as per the specification. According to the specification (Requirement WdgM327 as shown in Figure 25, WdgM327), the value should not exceed 255. We launched the model checker against the SAL model and the automatically generated SAL LTL (Linear Temporal Logic) theorem below to validate the requirement.

```
Safe_WdgM_WDGM327: THEOREM system |-
G(WdgM.WdgMFailedAliveSupervisionRefCycleTol <= 255 AND
WdgM.WdgMFailedAliveSupervisionRefCycleTol >= 0);
```

The above theorem map textually to,

> Globally, it is always true that WdgMFailedAliveSupervisionRefCycleTol is less or equal to 255 and greater than or equal to 0.

We updated the logic to initialize the variable to a value outside the specified range. The model checker captured the violation and indicated all the variable assignments/state paths that lead up to the violation. A snapshot of the violation is shown below:

```
$ sal-smc WdgM Safe_WdgM_WDGM327 --verbose=1
importing context "WdgM"...
parsing SAL file "WdgM.sal"...
creating abstract syntax tree for context "WdgM"...
type checking context "WdgM"...
flattening modules in the assertion located at [Context:
scratch, line(1), column(1)]
simplifying abstract syntax tree...
expanding function applications...
eliminating common subexpressions in an assertion...
eliminating common subexpressions in a flat module...
converting flat module to boolean flat module...
converting property to boolean property...
number of system variables: 92, number of auxiliary variables:
5
converting flat module to BDD representation (initial states,
and transition relation)...
proving invariant or producing counterexample using BDDs...
  using forward search
Counterexample:
========================
Path
========================
Step 0:
--- System Variables (assignments) ---
WdgM.WdgMExpectedAliveIndications = 1
WdgM.WdgMFailedAliveSupervisionRefCycleTol = 256
WdgM.WdgMInitialMode = 0
WdgM.WdgMSupervisionCycleCounter = 0
WdgM.WdgmAliveSupervisionStatus = 0
WdgM.WdgmDeadlineSupervisionStatus = 0
WdgM.WdgmLogicalSupervisionStatus = 0
EVT = EVT_Startup
```

```
WdgM_State = ST_WDGM_InitState
total execution time: 0.203 secs
```

Once the defect was removed from xtUML and Model compiler was launched to regenerate the fixed SAL model, the model checker reported that the theorem is proven as shown below:

```
$ sal-smc WdgM Safe_WdgM_WDGM327 --verbose=1
importing context "WdgM"...
parsing SAL file "WdgM.sal"...
creating abstract syntax tree for context "WdgM"...
type checking context "WdgM"...
flattening modules in the assertion located at [Context:
scratch, line(1), column(1)]
simplifying abstract syntax tree...
expanding function applications...
eliminating common subexpressions in an assertion...
eliminating common subexpressions in a flat module...
converting flat module to boolean flat module...
converting property to boolean property...
number of system variables: 91, number of auxiliary variables:
5
converting flat module to BDD representation (initial states,
and transition relation)...
proving invariant or producing counterexample using BDDs...
  using forward search
proved.
total execution time: 0.218 secs
```

The second experiment set was to validate compliance to the state machine as documented in the Watchdog Manager local state machine. Our experiments aimed to verify that the design is compliant to requirements 202, 203, 204, 300, 205, 206, 207, 291, 208 and 209 as discussed in section 5.2.5. Our generated SAL theorems aim to identify any matching conditions that lead to an incorrect state in the state machine thus in violation to the local Watchdog Manager AUTOSAR specification. Figure 58 shows an example of WDG requirement 201 as captured in xtUML as satisfiability conditions for the state to be active on the state level. Given the set of conditions, the

state machine needs to be at LOCAL_STATUS_OK state as per the Watchdog manager Specification document.

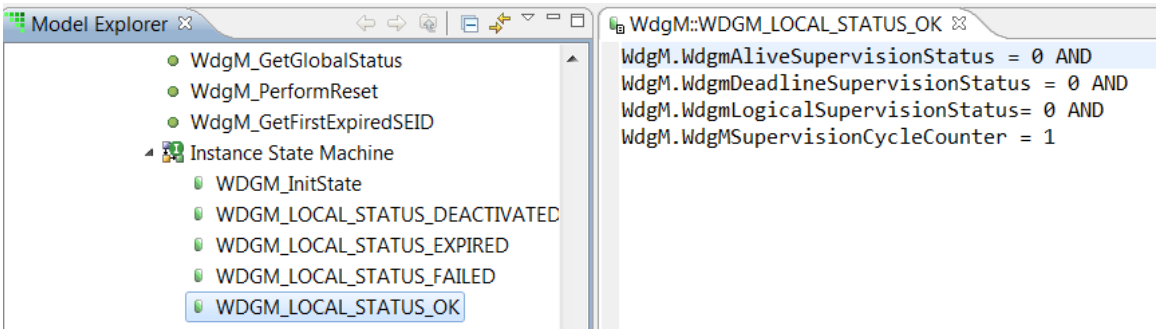

Figure 58 WdgM Requirement 202 in xtUML

The model compiler generated an LTL theorem in the SAL model that maps to the captured state level requirement expressed in xtUML as shown below:

```
Safe_WdgM_WDGM201: THEOREM system |- G(WdgM.WdgmAliveSupervisionStatus = 0
AND    WdgM.WdgmDeadlineSupervisionStatus = 0 AND
WdgM.WdgmLogicalSupervisionStatus=0 AND WdgM.WdgMSupervisionCycleCounter=1
=>        G(WdgM_State = ST_WDGM_LOCAL_STATUS_OK));
```

The LTL theorem map textually to the below description:

Globally, it is always true that when Alive supervision status = No error AND Deadline supervision status = No Error AND Logical Supervision Status = No Error and supervision cycle counter = 1 then globally, Watchdog local State should be WDGM_LOCAL_STATUS_OK.

The BDD (Binary Decision Diagram) based model checker initially proved the above theorem as shown below:

```
$ sal-smc WdgM Safe_WdgM_WDGM205 --verbose=2
importing context "WdgM"...
parsing SAL file "WdgM.sal"...
creating abstract syntax tree for context "WdgM"...
  ast generation time: 0.0 secs
type checking context "WdgM"...
  type-checker time: 0.0 secs
```

```
flattening modules in the assertion located at [Context:
scratch, line(1), column(1)]
calculating implicit assignments of base module at [Context:
WdgM, line(70), column(0)]...
  assertion flattening time: 0.0 secs
simplifying abstract syntax tree...
  simplification time: 0.0 secs
  LTL -> VWAA (very weak alternating automata)...
  VWAA -> GBA (generalized buchi automata)...
  simplifying GBA...
  GBA -> BA (buchi automata)...
  simplifying BA...
  number of states in the BA: 3
  eliminating common subexpressions...
  monitor generation time: 0.0 secs
converting flat module to boolean flat module...
  flat module -> boolean flat module conversion time: 0.015
secs
converting property to boolean property...
  property -> boolean property conversion time: 0.0 secs
number of system variables: 93, number of auxiliary variables:
5
converting flat module to BDD representation (initial states,
and transition relation)...
  creating BDD variables...
  computing static variable ordering (minimizing support)...
  static order time: 0.016 secs
  number of BDD variables: 196
  creating definition section BDDs...
  creating valid state predicate BDDs...
  creating BDD: set of initial states...
  creating BDD: transition relation...
  rearranging clusters...
  reordering BDD variables...
  compressing BDD clusters...
  rearranging clusters...
  flat-module -> BDD conversion time: 0.375 secs
proving invariant or producing counterexample using BDDs...
  using forward search
```

```
  iteration: 2
  iteration: 2
  iteration: 3
iteration: 4
  verification time: 0.0 secs
proved.
total execution time: 0.39 secs
```

We introduced a defect in the xtUML model where an additional setup initialization state was introduced which sets the same conditions in violation to the specification, moreover, the state machine no longer transition to STATUS_OK state correctly in violation to requirement WdgM201 which explains that the set of variables value map to the state machine being in LOCAL_STATE_OK. We ran the model checker that reported successfully the counter example/violation shown below which clearly shows a violation against the above theorem as the conditions lead up to being in Init_State instead of STATUS_OK state:

```
$ sal-smc WdgM Safe_WdgM_WDGM205_2 --verbose=2
importing context "WdgM"...
parsing SAL file "WdgM.sal"...
creating abstract syntax tree for context "WdgM"...
  ast generation time: 0.0 secs
type checking context "WdgM"...
  type-checker time: 0.0 secs
flattening modules in the assertion located at [Context:
scratch, line(1), column(1)]
calculating implicit assignments of base module at [Context:
WdgM, line(70), column(0)]...
  assertion flattening time: 0.0 secs
simplifying abstract syntax tree...
  simplification time: 0.0 secs
converting flat module to boolean flat module...
  flat module -> boolean flat module conversion time: 0.016
secs
converting property to boolean property...
  property -> boolean property conversion time: 0.015 secs
```

```
number of system variables: 91, number of auxiliary variables:
5
converting flat module to BDD representation (initial states,
and transition relation)...
  creating BDD variables...
  computing static variable ordering (minimizing support)...
  static order time: 0.0 secs
  number of BDD variables: 192
  creating definition section BDDs...
  creating valid state predicate BDDs...
  creating BDD: set of initial states...
  creating BDD: transition relation...
  rearranging clusters...
  reordering BDD variables...
  compressing BDD clusters...
  rearranging clusters...
  flat-module -> BDD conversion time: 0.203 secs
proving invariant or producing counterexample using BDDs...
  using forward search
```

```
Counterexample:
=======================
Path
=======================
Step 0:
--- System Variables (assignments) ---
WdgM.WdgMExpectedAliveIndications = 1
WdgM.WdgMFailedAliveSupervisionRefCycleTol = 1
WdgM.WdgMInitialMode = 0
WdgM.WdgMSupervisionCycleCounter = 0
WdgM.WdgmAliveSupervisionStatus = 0
WdgM.WdgmDeadlineSupervisionStatus = 0
WdgM.WdgmLogicalSupervisionStatus = 0
EVT = EVT_Startup
WdgM_State = ST_WDGM_InitState
total execution time: 0.234 secs
```

Similarly, we have generated LTL theorems for requirements 202, 203, 204, 300, 205, 206, 207, 291, 208 and 209 as shown below based on xtUML satisfiability conditions.

Safe_WdgM_WDGM202: THEOREM system |- G((
NOT(WdgM.WdgmAliveSupervisionStatus = 0) AND
NOT(WdgM.WdgMFailedAliveSupervisionRefCycleTol = 0)) OR (NOT
(WdgM.WdgmDeadlineSupervisionStatus = 0) OR
NOT(WdgM.WdgmLogicalSupervisionStatus = 0)) => G(WdgM_State =
ST_WDGM_LOCAL_STATUS_EXPIRED));
Safe_WdgM_WDGM203: THEOREM system |- G(NOT
(WdgM.WdgmAliveSupervisionStatus = 0) AND
NOT(WdgM.WdgMFailedAliveSupervisionRefCycleTol = 0) AND
WdgM.WdgmDeadlineSupervisionStatus = 0 AND WdgM.WdgmLogicalSupervisionStatus
= 0) => G(WdgM_State = ST_WDGM_LOCAL_STATUS_FAILED);

Safe_WdgM_WDGM204: THEOREM system |- G(NOT
(WdgM.WdgmAliveSupervisionStatus = 0) AND
WdgM.WdgMFailedAliveSupervisionRefCycleTol <
WdgM.WdgMSupervisionCycleCounter AND WdgM.WdgmDeadlineSupervisionStatus = 0
AND WdgM.WdgmLogicalSupervisionStatus = 0) => G(WdgM_State =
ST_WDGM_LOCAL_STATUS_FAILED);

Safe_WdgM_WDGM300: THEOREM system |- G(
WdgM.WdgmAliveSupervisionStatus = 0 AND
WdgM.WdgMSupervisionCycleCounter > 1 AND
WdgM.WdgmDeadlineSupervisionStatus = 0 AND
WdgM.WdgmLogicalSupervisionStatus = 0) => G(WdgM_State =
ST_WDGM_LOCAL_STATUS_FAILED);

Safe_WdgM_WDGM205: THEOREM system |- G(WdgM.WdgmAliveSupervisionStatus=0 AND WdgM . WdgmDeadlineSupervisionStatus=0 AND WdgM.WdgmLogicalSupervisionStatus=0 AND WdgM.WdgMSupervisionCycleCounter=1 => G(WdgM_State = ST_WDGM_LOCAL_STATUS_OK));
Safe_WdgM_WDGM206: THEOREM system |- G((NOT (WdgM.WdgmAliveSupervisionStatus = 0) AND WdgM.WdgMFailedAliveSupervisionRefCycleTol < WdgM.WdgMSupervisionCycleCounter) OR (NOT(WdgM.WdgmDeadlineSupervisionStatus = 0) OR NOT(WdgM.WdgmLogicalSupervisionStatus = 0))) => G(WdgM_State = ST_WDGM_LOCAL_STATUS_EXPIRED);

Safe_WdgM_WDGM207: THEOREM system |- G(Function= SET_MODE AND WdgM_State = ST_WDGM_LOCAL_STATUS_OK) => G(State = ST_WDGM_LOCAL_STATUS_DEACTIVATED AND Response=E_OK);

Safe_WdgM_WDGM291_1: THEOREM system |- G(Function= SET_MODE AND

WdgM_State = ST_WDGM_LOCAL_STATUS_FAILED) =>

G(State = ST_WDGM_LOCAL_STATUS_DEACTIVATED AND Response=E_OK);

Safe_WdgM_WDGM291_2: THEOREM system |- G(Function= SET_MODE AND

WdgM_State = ST_WDGM_LOCAL_STATUS_EXPIRED) =>

G(State = ST_WDGM_LOCAL_STATUS_EXPIRED AND Response=E_NOT_OK);

Safe_WdgM_WDGM209: THEOREM system |- G(Function= SET_MODE AND

WdgM_State = ST_WDGM_LOCAL_STATUS_DEACTIVATED) =>

```
G(State = ST_WDGM_LOCAL_STATUS_OK AND Response=E_OK);
```

We have successfully proved all the above theorems. We have also induced defects in the design and attempted to verify the theorems and were successfully able to get via the model checker the counter example that shows the violation in xtUML.

6.4 Mentor Graphics' WatchDog Manager Results

In order to evaluate our approach, we needed to conduct a comparative analysis between our proposal and an existing module that was developed in compliance to ISO 26262 via a BSW supplier. In this section, we present an analysis of defects and challenges faced while complying with ISO 26262 guidelines in a BSW watchdog Manager Module development.

Development team faced several challenges in their endeavor to comply several AUTOSAR BSW implementations with ISO 26262 ASIL B level[1]. Initially, the team focused on ASIL B compliancy since they were unable to conduct highly recommended guidelines in ASILs C and D, namely, semi-formal and formal verification of the design. Our aim is to present the challenges faced during trying to comply Watchdog Manager Implementation with ASIL B and try to overcome these challenges to pave the way for module suppliers to comply with verification design guidelines as recommended in ASILs C and D.

The first challenge faced was the ability to apply required verification methods. ISO 26262 formal and semi-formal verification guidelines were not feasible when working with embedded C software. Static analysis, and control/data flow analysis were considered as acceptable alternatives. Arguments for not performing semi-formal or formal verifications were made. The alternative approaches apply on the source code itself and are mainly manually driven. Control flow/data flow analysis were done by

[1] Feedback based on Mentor Graphics safety team ASIL B compliance challenges for AUTOSAR WatchDog Manager module

manually analyzing the control statements inside source code and creating control flow/data flow graphs for control statements and variables. This was feasible as the modules that were required to be ASIL B compliant were small which rendered this manual effort feasible. It is expected that this is not going to be possible with larger modules. From a safety team perspective aiming to reach ASIL C compliancy for developed modules, it is inevitable that the software be verified in early phases using semi-formal and formal verification as recommended by ISO 26262 design verification guidelines.

The second challenge faced was establishing traceability between requirements, design, and code and test elements in an automated fashion. Manual trace and label of the requirements were employed. Supporting traceability in an automated fashion between requirements, design, code and test elements would decrease the number of review/update iterations of sequence diagrams, control and data flow diagrams and traceability to software requirements.

The third challenge was related to the test case derivation. Table 5 shows the recommended methods for deriving test cases for software unit testing according to ISO 26262-6 Table 11. Automation methods that help in identifying decision points/variables to automatically generate test cases would definitely save time and ensure compliance to ISO 26262 guidelines.

In conclusion, all safety team verification and test case derivation methods were based on manual inspection, as recommended for ASIL B compliant modules. This will not be feasible for ASIL C compliant modules as semi-formal verification is highly recommended for levels greater than B.

Seventy-one defects were raised during ASIL B compliancy endeavor of the WatchDog Manager module. Additionally, some of the reported defects were uncovered after production and during customer module integration endeavors although they were introduced in the design stage. The table below summarizes the

raised defects number and classification. A defect leakage/slippage of 100% is visible in the current approach from design to testing stage.

Table 10 Watchdog Manager Defects Classification

Defect Count	Classification
16	Logic Bugs
35	Non-compliance to Specification
20	Traceability

Defects under logic bugs category include but are not limited to, bugs such as array bound issues, incorrect array index, invalid mathematical operator, and last array index not being initialized properly. Defects under non-compliance to specification includes defects such as WDGM triggers watchdog Interface to be in WDGIF_OFF_MODE while in WDGM_G_STATUS_STOPPED state (non-compliance to WDGM122) and WdgMExpectedAliveIndications, which is a defined parameter in the specification holding the amount of expected alive indications, range does not match AUTOSAR watchdog manager specification. The remaining defects were related to traceability (Missing text cases, missing relations between requirement and design/code/test elements).

6.5 Evaluation of the approach

Our aim in this research was to be able to verify design in an automated and reliable fashion to empower ASIL compliance to design verification guidelines as discussed in Table 2. Our framework enables formal verification of a semi-formal xtUML design. The design does not just capture architecture but also behavior via the action language inside states, transitions and functions. The framework addresses the complexities that discouraged the industry form moving to formal notation. The designer does not need to write formal notations or complex mathematical theorems, as this is done automatically via a model compiler that maps xtUML design and constraints into a formal model and set of specification compliance theorems[105,106,107,108].

The framework allows early detection of specification incompliances, and boundary analysis defects that was shown to be a major contributor to defects in software

systems. ISO 26262 design verification guidelines highly recommends using semi-formal verification for critical software in automotive as shown in Table 2 and highly recommends using analysis of requirements, equivalence classes, and boundary values to derive test cases to verify the design as discussed in Table 5. We have shown how our framework can be used to capture requirements in xtUML model that maps to specification requirements, equivalence classes and boundary values theorem to uphold while formally verifying the design.

We have extended the xtUML model to capture satisfiability conditions as follows:

- Variable satisfiability conditions (Upper and Lower limit): Generate theorems to cover boundary value analysis and equivalence classes
- State satisfiability conditions: capture conditions to ensure requirement compliance of variables in a given state in the state machine
- Transition satisfiability conditions: capture conditions to ensure requirement compliance of variables in a given transition in the state machine

We showed how our framework was able to detect all introduced defects, whether they are logic, or specification compliance defects in the design level via running SAL model checkers against our LTL generated theorems to detect model violation against the satisfiability conditions that is captured in our model. All requirement non-compliance defects that were previously detected on the testing/production level were detected via our framework on the design level. Our approach allows non-compliance to be detected on the design as opposed to the coding level in an automated way. The reported counterexamples are using the same UML notations and states allowing the UML designer to understand the faulty sequence and resolve the issue in UML domain. Counterexamples are additionally reported against the specification requirement Id ensuring traceability and ease of resolution via the application designer.

We have verified the framework on three industrial software modules, namely, FlexRay State Manager AUTOSAR module, Watchdog Manager AUTOSAR module, and Automatic transmission controller. We have shown how our framework detects software incompliances and boundary violations to variables as well as logic bugs (

out of bound counter increment) on the use-case modules. We have introduced defects in Watchdog Manager Module that has been found after release of a software to customers. Our framework was able to detect all defects on the design as opposed to post production level.

In comparison to the industrial module developed without applying our framework, the defects slippage/leakage percentage from design to testing has decreased from 100% to 28% (51 out of the 71 defects were identified in the design stage as opposed to the testing stage).

Chapter 7. Conclusion, contributions and Future work

Our intent in this research was to address software verification in the early stage of the software lifecycle, namely, the design stage. The research was motivated by the steep growth of critical software functions in embedded systems, the fact that 50% of defects are introduced by the design stage, cost of finding a defect during testing is much higher than finding it during design, late defects are mostly due to specification incompliance defects, and the birth of AUTOSAR Automotive standard and ISO-26262.

We have reported that current V&V techniques utilized still heavily depend on testing and little effort focuses on pushing the verification to the design stage. Our research aimed to address the motivations while addressing the current shortcoming that have discouraged the industry from using formal methods in the design stage of automotive software development. Formal methods have not been widely adopted due to complexity of notations, lack of support and lack of support tools. Automotive suppliers are also looking for non-disruptive techniques that integrate with their used models and design environments so that they do not have to re-invent the wheel for their software development lifecycle.

With the above said, our aim was to propose a framework that addresses the above motivations and challenges and fulfills the main objective of being able to identify defects in the design stage via extensive thorough methods as opposed to a method that is based on some selection criteria. We proposed a framework where any UML finite state machine based model could be transformed to a formal transition model augmented with complex data types in SAL notation. The mapping is based on a 1-1 UML to SAL mapping. We showed how we were able to formally verify several semi-formal models via augmenting the UML model with satisfiability conditions. We have constructed theorems that represent specification requirements in UML. A model compiler was developed to map the UML model into SAL formal model and LTL based theorems automatically shielding the application designer from having to

create a formal model. A SAL solver was utilized so that formal verification can be accomplished on the SAL model. We demonstrated how the solver through the asserted counterexamples detected specification non-compliances. We have shown how our framework can detect requirement non-compliances as well as support tests of boundary conditions in an automated fashion. The application model engineer could fix the model violations at a very early stage based on the formal verification of the semi-formal model using the proposed approach.

We have tested the framework on three industrial modules, namely, AUTOSAR FlexRay manager, Automatic Transmission Controller, and Watchdog Manager. We mapped the specification requirements into UML design elements and satisfiability conditions. We showed how the model (with behavior) was transformed from specification to UML design to a SAL formal model. We have also shown how the requirements were mapped into UML satisfiability conditions, which were mapped into formal assertions (theorems). We executed several formal checkers on the formal model with assertions to show that specification incompliances (state, transition, and boundary conditions), static properties, logic defects and deadlock detection can happen early in the design stage. We verified 51 requirements and introduced several defects in the design model to show how the framework is able to detect the incompliance. The validity of the mapping / SAL generation was established through the correct assertions that show violations of satisfiability conditions in the UML design. All introduced defects were properly detected and reported via the model checker. In comparison to the industrial module developed without applying our framework, the defects slippage/leakage percentage from design to testing has decreased from 100% to 28% (51 out of the 71 defects were identified in the design stage as opposed to the testing stage)

The proposed approach makes emerging ISO 26262 standard ASIL C and D test case derivation and software unit design and implementation verification guidelines, namely, semi-formal and formal verification guidelines possible in an automated fashion. Our work also addresses one of AUTOSAR's major drives, which is early

design defects detection. Problems such as the complexity of the formal notations, theorem construction and checkers execution and analysis have been shielded via the use-case yet the benefits of using formal verification on a semi-formal notation are retained.

Our research focused on UML state machine diagram. The work can be extended to cover other behavioral UML diagrams (sequence or use-cases). We have also not tested the framework against complex state machines to ensure reasonable execution time via the formal model checkers. We currently support one level requirement in UML model that is mapped into SAL model theorem. The framework can be extended to support multi-level requirement that can be generated into several theorems. We also need to qualify the model compiler in accordance with ISO 26262 tool qualification guidelines to ensure the safety of the tool and the generated SAL intermediate language in preparation for industrial utilization of the tool

The research could be extended in the future to automatically generate software implementation based on the verified design. Once the design is verified, the same framework could be extended to generate AUTOSAR compliant C code. This will ensure that defects uncovered and fixed in the design are not later re-introduced in the implementation stage. We also propose work to integrate our flow to AUTOSAR XML metamodel language that defines a system. The aim will be to interact with existing AUTOSAR xml tools to automatically map the AUTOSAR model architecture into starting UML structure. This proposes a system level flow where the automotive OEM can start with a system level model in AUTOSAR XML, which automatically can be compiled into UML modules with requirements. This output would then become the starting point to our flow and will ensure that formally verified design can be used to generate implementation.

Chapter 8. References

1. Feder, Barnaby J. "A Heart Device Is Found Vulnerable to Hacker Attacks". The New York Times, 2008-09-28.
2. US-Canada Power System Outage Task Forcem “Final Report on August 14th, 2003 Blackout in the United States and Canada. Causes and Recommendations.”, April 2004. https://reports.energy.gov/BlackoutFinal-Web.pdf.
3. Nancy Leveson, Clark S Turner, “An Investigation of the Therac-25 Accidents.” IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41.
4. http://spectrum.ieee.org/riskfactor/green-tech/advanced-cars/gm-recalls-50500-2011-cadillac-srxs-over-airbagrelated-software-glitch
5. Bahig, G.M., El-Kadi, A.,” Ensuring software safety in safety critical domains”, Internet Technology And Secured Transactions, 2012 International Conference for Internet Technology and Secured Transactions, Dec. 2012.
6. Gwangmin Park ; Daehyun Ku ; Seonghun Lee ; Woong-Jae Won, Test methods of the AUTOSAR application software components, ICCAS-SICE, 2009.
7. Mews, M. ; Svacina, J. ; Weissleder, S. ,From AUTOSAR Models to Co-simulation for MiL-Testing in the Automotive Domain, Software Testing, Verification and Validation (ICST), 2012 IEEE.
8. ANSI/IEEE. Standard Glossary of Software Engineering Terminology. New York: IEEE, 1983.
9. http://www.coverity.com/library/pdf/ControllingSoftwareComplexity.pdf
10. V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, J. Cazin, G. Durrieu, “ Formal Verification of Critical Aerospace Software.”, AerospaceLab Journal, Issue 4, May 2012.

11. Lutz RR. Software engineering for safety: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering.Vol 1. ACM; 2000:213–226. Available at: http://portal.acm.org/citation.cfm?id=336512.336556
12. Edward A. Lee, ” Key challenges in Embedded Software”, System Design Frontier, Volume 2, Number 1, January 2005
13. http://www.iso.org/iso/catalogue_detail?csnumber=43464
14. Pfleeger, S.L., Fenton, N. , Page, S. “ Evaluating software engineering standards.”, Computer, Volume 27, Issue 9, Pages 71-79, 2002.
15. http://www.coverity.com/library/pdf/Coverity_White_Paper-SAT-Next_Generation_Static_Analysis.pdf
16. Beyer D, Henzinger T. Shape refinement through explicit heap analysis. Approaches to Software. 2010:263-277.
17. Beyer D, Henzinger T, Théoduloz G. Lazy shape analysis. In: Computer Aided Verification. Springer; 2006:532–546.
18. Beyer D, Henzinger TA, Jhala R, Majumdar R. Checking memory safety with Blast. Fundamental Approaches to Software Engineering. 2005:2–18.
19. Beyer D, Henzinger T a, Majumdar R, Rybalchenko A. Path invariants. ACM SIGPLAN Notices. 2007;42(6):300.
20. Beyer D, Henzinger T a, Theoduloz G. Program Analysis with Dynamic Precision Adjustment. 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. 2008:29-38.
21. Jones N, “Introduction to MISRA C”, EE Times, 2002
22. Zheng, Jiang ; Williams, Laurie ; Nagappan, Nachiappan ; Snipes, Will ; Hudepohl, John ; Vouk, Mladen “A Study of Static Analysis for Fault Detection in Software”, Technical Report, NC State University, 2005
23. Lu Luo, “Software Testing Techniques: Technology Maturation and Research Strategy. “, Technical Report.
24. M. Prasanna, S.N. Sivanandam, R. Venkastesan, R. Sundarrajan, “ A Survey On Automatic Test Case Generation, “, acadjournal, Volume 15, 2005.
25. Clay E. Williams, November 1999, "Software testing and the UML", International Symposium on Software Reliability Engineering (ISSRE’99), Boca, Raton.

26. Pfleeger SL, Hatton L. Do Formal Methods Improve Code Quality? leshatton.org.1-21. http://www.leshatton.org/Documents/IEEEComputer1-97.pdf
27. Woodcock J, Larsen P. G, Bicarregui J, Fitzgerald J, "Formal Methods: Practice and Experience", ACM Computing Surveys, 2009
28. Bowen JP. The Ethics of Safety-Critical Systems. Communications of the ACM. 2000;43(4):91-9.
29. Basir N, Denney E, Fischer B. Constructing a safety case for automatically generated code from formal program verification information. Computer Safety, Reliability, and Security. 2008:249–262. Available at: http://www.springerlink.com/index/d21658255100p251.pdf.
30. Basir N, Denney E, Fischer B. Deriving Safety Cases for the Formal Safety Certification of Automatically Generated Code. Electronic Notes in Theoretical Computer Science. 2009;238(4):19-26.
31. Basir N, Denney E, Fischer B. Constructing a safety case for automatically generated code from formal program verification information. Computer Safety, Reliability, and Security. 2008:249–262.
32. Denney E. Correctness of source-level safety policies. FME 2003: Formal Methods. 2003.
33. Denney E, Fischer B. Extending Source Code Generators for Evidence-Based Software Certification. Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006). 2006:138-145.
34. Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra, " Combining Formal Methods and MDE Techniques for Model-Driven System Design and Analysis.", International Journal on Advances in Software, 2010 vol. 3 nr. 1 & 2.
35. M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim, "Integrating a formal method into a software engineering process with UML and Java," Form. Asp. Comput., vol. 20, no. 2, pp. 161–204, 2008.
36. T. Zhang, F. Jouault, J. Bézivin, and J. Zhao, "A MDE Based Approach for Bridging Formal Models," in TASE '08. IEEE Computer Society,

2008, pp. 113–116.

37. Thomas Koltz, Eva Fordoran, Bernd Straube, and Jurgen Haufe, “Formal Verification of UML-modeled Machine Controls.”, IEEE 2009
38. Wei-gang Ma, Xin-hong Hei, “An Approach for Design and Formal Verification of Safety-Critical Software.”, 2010 International Conference on Computer Application and System Modeling (ICCASM 2010)
39. Vitus S.W. Lam and Julian Padget, “Symbolic Model Checking of UML Statchart Diagrams with an Integrated Approach.”, Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS’04)
40. Shao Jie Zhang, Yang Liu, “An Automatic Approach to Model Checking UML State Machines.”, 2010 Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement Companion.
41. Lixia JI, Jianhong, Zhuowei Shan, “Research on Model Checking Technology of UML.”, 2012 International Conference on Computer Science and Service System, IEEE.
42. “OMG. The Unified Modeling Language (UML), v2.1.2,” http:// www.uml.org, 2007.
43. “Eclipse Modeling Framework (EMF),” http://www.eclipse.org/emf/.
44. D. Gasevic, R. Lämmel, and E. V. Wyk, Eds., Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008.
45. “The Maude System,” http://maude.cs.uiuc.edu/.
46. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, “ATL: a QVT-like transformation language,” in Proc. OOPSLA’06. ACM, 2006, pp. 719–720.
47. A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo, “The design of a language for model transformations,” Software and System Modeling, vol. 5, no. 3, pp. 261–288, 2006.
48. J. Fischer, M. Piefel, and M. Scheidgen, “A Metamodel for SDL-2000 in the Context of Metamodelling ULF,” in Proc. SAM’04, 2004, pp. 208–223.

49. M. Alanen and I. Porres, “A Relation Between Context-Free Grammars and Meta Object Facility Metamodels,” Turku Centre for Computer Science, Tech. Rep., 2003.
50. M. Wimmer and G. Kramler, “Bridging grammarware and modelware,” in Proc. of the 4th Workshop in Software Model Engineering (WiSME’05), Montego Bay, Jamaica, 2005.
51. T. Gjøsæter, I. F. Isfeldt, and A. Prinz, “Sudoku - a language description case study,” in Proc. SLE’08, 2008, pp. 305–321.
52. “Abstract State Machines tools,” http://www.eecs.umich.edu/gasm/tools.html.
53. Y. Gurevich and B. Rossman and W. Schulte, “Semantic Essence of AsmL,” Microsoft Research Technical Report MSR-TR-2004-27, March 2004 .
54. A. Slissenko and P. Vasilyev, “Simulation of timed abstract state machines with predicate logic model-checking,” J. UCS, vol. 14, no. 12, pp. 1984–2006, 2008.
55. F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, “Derivation and refinement of textual syntax for models,” in ECMDAFA, 2009.
56. F. Jouault, J. Bézivin, and I. Kurtev, “TCS: a DSL for the specification of textual concrete syntaxes in model engineering.” in Proceedings of the fifth international conference on Generative programming and Component Engineering (GPCE’06), 2006
57. S. Efftinge, “oAW xText - A framework for textual DSLs,” in Workshop on Modeling Symposium at Eclipse Summit, 2006.
58. P.-A. Muller, F. Fondement, F. Fleurey, M. Hassenforder, R. Schneckenburger, S. Gérard, and J.-M. Jézéquel, “Model-driven analysis and synthesis of textual concrete syntax,” Software and System Modeling, vol. 7, no. 4, pp. 423–441, 2008.
59. “OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08- 01,” http://www.uml.org/.
60. D. Hearnden, K. Raymond, and J. Steel, “Anti-Yacc: MOF-to-text,” in Proc. of EDOC, 2002, pp. 200–211.

61. . A. Idani, J.-L. Boulanger, and L. P. 0002, “A generic process and its tool support towards combining uml and b for safety critical systems,” in Proc. CAINE, 2007, pp. 185–192.
62. Y. Sun, Z. Demirezen, F. Jouault, R. Tairas, and J. Gray, “A model engineering approach to tool interoperability,” in SLE, 2008, pp. 178–187.
63. P.-A. Muller, F. Fleurey, and J.-M. Jezequel, “Weaving Executability into
64. Object-Oriented Meta-Languages,” in Proc. MODELS, 2005.
65. M. Soden and H. Eichler, “Towards a model execution framework for Eclipse,” in Proc. of the 1st Workshop on Behavior Modeling in Model-Driven Architecture. ACM, 2009
66. J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo, “Analyzing rulebased behavioral semantics of visual modeling languages with maude,” in SLE, ser. Lecture Notes in Computer Science, D. Gasevic, R. Lämmel, and E. V. Wyk, Eds., vol. 5452. Springer, 2008, pp. 54–73.
67. K. Chen, J. Sztipanovits, and S. Neema, “Toward a semantic anchoring infrastructure for domain-specific modeling languages,” in EMSOFT, 2005, pp. 35–43.
68. D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio, “Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs,” LINA, Tech. Rep. 06.02, 2006.
69. M. Anlauff, “XASM - An Extensible, Component-Based ASM Language,” in Proc. of Abstract State Machines, 2000, pp. 69–90.
70. D. A. Sadilek and G. Wachsmuth, “Using grammarware languages to define operational semantics of modelled languages,” in TOOLS (47), 2009, pp. 348–356.
71. A. Gargantini, E. Riccobene, and P. Scandurra, “A semantic framework for metamodel-based languages,” Journal of Automated Software Engineering, vol. 16, no. 3-4, pp. 415–454, 2009.
72. A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra, “Exploiting the ASM method for Validation & Verification of Embedded Systems,” in Proc. of ABZ’08, LNCS 5238. Springer, 2008, pp. 71–84.

73. E. Riccobene and P. Scandurra, “An executable semantics of the SystemC UML profile,” in ABZ 2010, ser. LNCS, M. F. et al., Ed., vol. 5977, 2010, pp. 75–90.
74. E. Riccobene, P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, and L. Mantellini, “SystemC/C-based model-driven design for embedded systems,” ACM Trans. Embedded Comput. Syst., vol. 8, no. 4, 2009.
75. J. Armstrong, “Industrial integration of graphical and formal specifications,” J. of Systems and Software, vol. 40, no. 3, pp. 211–225, 1998.
76. Denney E, Fischer B, Schumann J, Richardson J. Automatic Certification of Kalman Filters for Reliable Code Generation. *2005 IEEE Aerospace Conference*. 2005;(1207):1-1.
77. Fey-Safety I. Model-Based Design for Safety-Related Applications. *Society*. 2008. Available at: http://papers.sae.org/2008-21-0033.
78. http://sal.csl.sri.com/introduction.shtml.
79. http://www.autosar.org/
80. http://www.autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf
81. http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf
82. http://www.autosar.org/download/R3.1/AUTOSAR_SWS_FlexRay_StateManager.pdf
83. Gwangmin Park ; Daehyun Ku ; Seonghun Lee ; Woong-Jae Won, Test methods of the AUTOSAR application software components, ICCAS-SICE, 2009.
84. Mews, M. ; Svacina, J. ; Weissleder, S. ,From AUTOSAR Models to Co-simulation for MiL-Testing in the Automotive Domain, Software Testing, Verification and Validation (ICST), 2012 IEEE.
85. H. Altinger; Y. Dajsuren; S. Siegl; J. Vinju; F. Wotawa. “On Error-class Distribution in Automotive Model-Based Software,” International Conference on Software Analysis, Evolution and Reengineering, 2016 IEEE
86. https://automotivetechis.wordpress.com/autosar-concepts/

87. Alberto Sangiovanni-Vincentelli, 2003, “Electronic-System Design in the Automobile Industry.” IEEE Micro, vol. 23, no. 3, pp. 8-18.
88. Beizer B. Software Testing Techniques. Van Nostrand Reinhold, 2nd edition, 1990.
89. C. Cheng, A. Dumitrescu, P. Schroeder, 2003, “Generating Small Combinatorial Test Suites to Cover Input-Output Relationships.” Proceedings of 3rd Quality Software International Conference (QSIC '03) pp. 76-82.
90. Anders Hessel and Paul Pettersson, “A Global Algorithm for Model-Based Test Suite Generation.” Third Workshop on Model-Based Testing, Braga, Portugal, Satellite workshop of ETAPS 2007.
91. P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann, 1998, “Qualitatssicherung Software-basierter Technischer Systeme--Problembereiche und Lösungs-ansätze.” Informatik Spektrum, 21(5):249--258.
92. Software Engineering Institute; Carnegie Mellon University; Handbook CMU/SEI-96-HB-002; page 56-58
93. Roy Awedikian. Quality of the design of test cases for automotive software: design platform and testing process. Business administration. Ecole Centrale Paris, 2009. English.
94. A. NyBen, P. Konemann. “Model-based Automotive Software Development using Autosar, UML, and Domain-Specific Languages,” Embedded World Conference 2013, Nurnberg, Germany.
95. Win-Bin See, "UML-based modeling approach for automotive system development," 2005 IEEE International Conference on Industrial Technology, Hong Kong, 2005, pp. 448-452.
doi: 10.1109/ICIT.2005.1600680
96. https://xtuml.org/
97. Stephen Mellor, Sally Shlaer - Modeling the World in Data, ISBN 978-0136290230. Stephen Mellor, Sally Shlaer - Object Lifecycles, ISBN 978-0136299400.

98. H. Siyuan and Z. Hong, "Towards Transformation from UML to Event-B," 2015 IEEE International Conference on Software Quality, Reliability and Security - Companion, Vancouver, BC, 2015, pp. 188-189.

99. M. Sharbaf, B. Zamani and B. T. Ladani, "Towards automatic generation of formal specifications for UML consistency verification," 2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI), Tehran, 2015, pp. 860-865.

100. S. Neysian and S. M. Babamir, "Automatic verification of uml state chart by bogor model checking tool: Automatic formal verification of network and distributed systems," 2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI), Tehran, 2015, pp. 797-802.

101. A. B. Hocking, J. Knight, M. A. Aiello and S. Shiraishi, "Arguing Software Compliance with ISO 26262," Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on, Naples, 2014, pp. 226-231.doi: 10.1109/ISSREW.2014.88.

102. http://www.autosar.org/standards/classic-platform/release-40/software-architecture/communication-stack/

103. http://www.autosar.org/standards/classic-platform/release-32/software-architecture/communication-stack/

104. https://www.mathworks.com/help/simulink/examples/modeling-an-automatic-transmission controller.html?requestedDomain=www.mathworks.com

105. G. Bahig, A. El-Kadi and A. Salem, "Formal verification of AUTOSAR FlexRay state manager," *2014 9th IEEE International Design and Test Symposium (IDT)*, Algiers, 2014, pp. 193-198. doi: 10.1109/IDT.2014.7038612

106. G. Bahig and A. El-Kadi, "Formal Verification Framework for Automotive UML Designs," *2016 2nd ACM Africa and Middle East Conference on Software Engineering (AMECSE)*, Egypt, 2016, pp. 21-27. doi: 10.1145/2944165.2944169

107. G. Bahig, A. El-Kadi, A. El-Hamedy and A. Salem, "Safety Analysis of AUTOSAR WatchDog Manager: A Case Study," *2016 1st IEEE Automotive Reliability and Test Workshop (ART)*, International Test Conference, Fort-Worth, Texas, 2016

108. G. Bahig and A. El-Kadi, "Formal Verification of Automotive Design in Compliance With ISO 26262 Design Verification Guidelines," in *IEEE Access*, vol. 5, no. , pp. 4505-4516, 2017.
doi: 10.1109/ACCESS.2017.2683508

APPENDIX A

SAL Language

Types

The SAL language supports the built-in basic types for booleans, natural numbers, integers, and reals. New basic types may be introduced using uninterpreted type declarations. Types may be used in type constructions to create subtype, sub range, array, function, tuple, and record types. Function, tuple, and record types may be dependent. In addition to uninterpreted type declarations, that introduce a name without a defining form, type declarations may be used to introduce names for existing types, as well as scalars and data types. The grammar for types is given by

```
TypeDef        :=  Type
                 | ScalarType
                 | DataType
Type           :=  BasicType
                 | Name
                 | Subrange
                 | SubType
                 | ArrayType
                 | TupleType
                 | FunctionType
                 | RecordType
                 | StateType
BasicType      :=  BOOLEAN | REAL | INTEGER | NZINTEGER | NATURAL | NZREAL
Name           :=  Identifier
QualifiedName  :=  Identifier[ {ActualParameters} ]!Identifier
Subrange       :=  [ Bound .. Bound ]
SubType        :=  { Identifier : Type | Expression }
Bound          :=  Unbounded | Expression
Unbounded      :=  -
ArrayType      :=  ARRAY IndexType OF Type
IndexType      :=  INTEGER | Subrange | ScalarTypeName
ScalarTypeName :=  Name
TupleType      :=  [ VarType , {VarType}+, ]
FunctionType   :=  [ VarType -> Type ]
VarType        :=  [ Identifier : ] Type
RecordType     :=  [# {Identifier : Type}+, #]
StateType      :=  Module . STATE
ScalarType     :=  {{Identifier}+,}
DataType       :=  DATATYPE Constructors END
Constructors   :=  {Identifier[ ( Accessors) ]}+,
Accessors      :=  {Identifier : Type}+,
```

Figure 59 Types in SAL Grammar

Expressions

Expressions in the SAL language have to be type-correct with respect to the types in the type language. The expressions consist of constants, variables, applications with Boolean, arithmetic, and bit-vector operations, and array, function, tuple, and record selection and updates. Conditional (if-then-else) expressions are also part of the expression language as shown in Figure 60 and Figure 61.

```
Expression  :=  NameExpr
            |  QualifiedNameExpr
            |  NextVariable
            |  Numeral
            |  Application
            |  InfixApplication
            |  ArraySelection
            |  RecordSelection
            |  TupleSelection
            |  UpdateExpression
            |  LambdaAbstraction
            |  QuantifiedExpression
            |  LetExpression
            |  SetExpression
            |  ArrayLiteral
            |  RecordLiteral
            |  TupleLiteral
            |  Conditional
            |  ( Expression )
            |  StatePred
```

Figure 60 SAL Expressions

```
NameExpr              :=  Name
QualifiedNameExpr     :=  QualifiedName
NextVariable          :=  Identifier '
Application           :=  Function Argument
Function              :=  Expression
Argument              :=  ({Expression}+,)
InfixApplication      :=  Expression Identifier Expression
ArraySelection        :=  Expression[Expression]
RecordSelection       :=  Expression.Identifier
TupleSelection        :=  Expression.Numeral
UpdateExpression      :=  Expression WITH Update
Update                :=  UpdatePosition := Expression
UpdatePosition        :=  {Argument | [Expression] | .Identifier | .Numeral}+
LambdaAbstraction     :=  LAMBDA (VarDecls) : Expression
VarDecls              :=  {VarDecl}+;
VarDecl               :=  {Identifier}+, : Type
QuantifiedExpression  :=  Quantifier (VarDecls) : Expression
Quantifier            :=  FORALL | EXISTS
LetExpression         :=  LET LetDeclarations IN Expression
LetDeclarations       :=  {Identifier : Type = Expression}+;
SetExpression         :=  SetListExpression | SetPredExpression
SetPredExpression     :=  { Identifier : Type | Expression }
SetListExpression     :=  | { {Expression}+, }
ArrayLiteral          :=  [[IndexVarDecl] Expression]
IndexVarDecl          :=  Identifier : IndexType
RecordLiteral         :=  (# {RecordEntry}+, #)
RecordEntry           :=  Identifier := Expression
TupleLiteral          :=  Argument
Conditional           :=  IF Expression ThenRest
ThenRest              :=  THEN Expression
                          [ ElsIf ]
                          ELSE Expression ENDIF
ElsIf                 :=  ELSIF Expression ThenRest
StatePred             :=  Module . (INIT | TRANS)
```

Figure 61 SAL Expressions – Detailed

Transition Language

A transition system module consists of a state type, an invariant definition on this state type, an initialization condition on this state type, and a binary transition relation of a specific form on the state type. The state type is defined by four pairwise disjoint sets of input, output, global, and local variables. The input and global variables are the observed variables of a module and the output, global, and local variables are the controlled variables of the module. The transition rules are constraints on the current and next states of the transition. The current variables are written as X whereas the next state variables are written as X'

Definitions are the basic constructs used to build up the invariants, initializations, and transitions of a module. Definitions are used to specify the trajectory of variables in a computation by providing constraints on the controlled variables in a transition system. For variables ranging over aggregate data structures like records or arrays, it is possible to define each component separately. For example,

x’ = x + 1 simply increments the state variable x, where x’ is the newstate of the variable,

y’[i] = 3 sets the new state of the array y to be 3 at index i, and to remain unchanged on all other indices,

z.foo.1[0] = y constrains state variable z, which is a record whose foo component is a tuple, whose first component in turn is an array of the same type as y.

The left-hand side of a definition is given by the nonterminal Lhs. For an RhsExpression, the Lhs is simply assigned the corresponding value.

$$
\begin{array}{rcl}
\mathit{Lhs} & := & \mathit{Identifier}\,[\,'\,]\,\{\mathit{Access}\}^{*} \\
\mathit{Access} & := & \mathit{ArrayAccess} \mid \mathit{RecordAccess} \mid \mathit{TupleAccess} \\
\mathit{ArrayAccess} & := & [\ \mathit{Expression}\ ] \\
\mathit{RecordAccess} & := & .\ \mathit{Identifier} \\
\mathit{TupleAccess} & := & .\ \mathit{Numeral}
\end{array}
$$

Simple definitions are of the form

$$
\begin{array}{rcl}
\mathit{SimpleDefinition} & := & \mathit{Lhs}\ \mathit{RhsDefinition} \\
\mathit{RhsDefinition} & := & \mathit{RhsExpression} \mid \mathit{RhsSelection} \\
\mathit{RhsExpression} & := & =\ \mathit{Expression} \\
\mathit{RhsSelection} & := & \texttt{IN}\ \mathit{Expression}
\end{array}
$$

Figure 62 Rhs/Lhs Definitions

Module Language

A module is a self-contained specification of a transition system in SAL. Modules can be independently analyzed for properties and composed synchronously or asynchronously. Here is a fairly simple module declaration. Figure 63 shows an example of a SAL module

```
m : MODULE =
 BEGIN
  INPUT temp: INTEGER
  LOCAL high: BOOLEAN, ctr: NATURAL
  OUTPUT danger: BOOLEAN
  DEFINITION high = i > 100
  INITIALIZATION ctr = 0; danger = FALSE
  TRANSITION [  ctr > 3 --> danger' = danger OR high
             [] ctr <= 3 AND high --> ctr' = ctr + 1
             [] ELSE --> ctr' = 0
              ]
 END
```

Figure 63 SAL Module

m is a BaseModule, that is intended to monitor the temperature and indicate a problem if the temperature stays high for too long. It declares the input variable temp, local variables high and ctr, and output variable danger. Initially danger is FALSE and ctr is 0, and when this module is activated it sets danger to TRUE if temp exceeds 100 more than 3 times in a row. Once base modules are declared, they may be composed synchronously or asynchronously to yield new modules. Figure 64 and Figure 65 show the grammar of module expressions.

A BaseModule identifies the pairwise distinct sets of input, output, global, and local variables. This characterizes the state of the module. Base modules also may consist of several sections. The grammar allows variables and sections to be given in any order, and there may, for example, be 3 distinct TRANSITION sections. In every case, it is the same as if there was a prescribed order, with each class of variable and section being the union of the individual declarations.

Definitions appearing in the DEFINITION section(s) are treated as invariants for the system. When composed with other modules, the definitions remain true even during the transitions of the other modules. For this reason, proof obligations may be generated for a composition where definition sections are involved. This section is usually used to define controlled variables whose values ultimately depend on the

inputs, for example, a Boolean variable that becomes true when the temperature goes above a specified value.

The INITIALIZATION section(s) constrain the possible initial values for the local, global, and output declarations. Input variables may not be initialized. The INITIALIZATION section(s) determine a state predicate that holds of the initial state of the base module.

The TRANSITION section(s) constrain the possible next states for the local, global, and output declarations. As this is generally defined relative to the previous state of the module, the transition section(s) determine a state relation. Input variables may not appear on the Lhs of any assignments.

Modules can be combined by either synchronous or asynchronous composition. Let module Mi consists of input variables Ii , output variables Oi , global variables Gi , and local variables Li . The module M1||M2 and M1[]M2 respectively represent the synchronous and asynchronous composition of M1 and M2.

It is good pragmatics to name a module. This name can be used to index the local variables so that they need not be renamed during composition. Also, the properties of the module can be indexed on the name for quick look-up. Parametric modules allow the use of logical (state-independent) and type parameterization in the definition of modules.

Module := *BaseModule*
| *ModuleInstance*
| *SynchronousComposition*
| *AsynchronousComposition*
| *MultiSynchronous*
| *MultiAsynchronous*
| *Hiding*
| *NewOutput*
| *Renaming*
| *WithModule*
| *ObserveModule*
| (*Module*)

BaseModule := `BEGIN` *BaseDeclarations* `END`
BaseDeclarations := {*BaseDeclaration*}*
BaseDeclaration := *InputDecl*
| *OutputDecl*
| *GlobalDecl*
| *LocalDecl*
| *DefDecl*
| *InitDecl*
| *TransDecl*
InputDecl := `INPUT` *VarDecls*
OutputDecl := `OUTPUT` *VarDecls*
GlobalDecl := `GLOBAL` *VarDecls*
LocalDecl := `LOCAL` *VarDecls*
DefDecl := `DEFINITION` *Definitions*
InitDecl := `INITIALIZATION` {*DefinitionOrCommand*}$^{+}_{;}$ [;]
TransDecl := `TRANSITION` {*DefinitionOrCommand*}$^{+}_{;}$ [;]

DefinitionOrCommand := *Definition*
| [*SomeCommands*]
Definitions := {*Definition*}$^{+}_{;}$
Definition := *SimpleDefinition* | *ForallDefinition*
ForallDefinition := (`FORALL` (*VarDecls*): *Definitions*)
SimpleDefinition := *Lhs RhsDefinition*
Lhs := *Identifier* ['] {*Access*}*
Access := *ArrayAccess* | *RecordAccess* | *TupleAccess*
ArrayAccess := [*Expression*]
RecordAccess := . *Identifier*
TupleAccess := . *Numeral*
RhsDefinition := *RhsExpression* | *RhsSelection*
RhsExpression := = *Expression*
RhsSelection := `IN` *Expression*
SomeCommands := {*SomeCommand*}$^{+}_{[]}$ [[] *ElseCommand*]
SomeCommand := *NamedCommand* | *MultiCommand*
NamedCommand := [*Identifier* :] *GuardedCommand*
GuardedCommand := *Guard* --> *Assignments*
Guard := *Expression*
Assignments := {*SimpleDefinition*}$^{*}_{;}$ [;]
MultiCommand := ([] (*VarDecls*): *SomeCommand*)
ElseCommand := [*Identifier* :] `ELSE` --> *Assignments*

Figure 64 Module Grammar

```
ModuleInstance      := {ModuleName | QualifiedModuleName} Name[ [{Expression}+,] ]
ModuleName          := Name
QualifiedModuleName := QualifiedName
SynchronousComposition  := Module || Module
AsynchronousComposition := Module [] Module
MultiSynchronous    := (|| (Identifier : IndexType): Module)
MultiAsynchronous   := ([] (Identifier : IndexType): Module)
Hiding              := LOCAL {Identifier}+, IN Module
NewOutput           := OUTPUT {Identifier}+, IN Module
Renaming            := RENAME Renames IN Module
Renames             := {Lhs TO Lhs}+,
WithModule          := WITH NewVarDecls Module
NewVarDecls         := {InputDecl | OutputDecl | GlobalDecl}+;
ObserveModule       := OBSERVE Module WITH Module
```

Figure 65 Module Grammar 2

SAL Contexts

The SAL context language provides the framework for declaring types, constants, modules, and module properties. Figure 66 show the syntax for contexts containing declarations for constants, types, modules, assertions, and other (imported) contexts. SAL contexts are read from left to right, top to bottom, and an entity must be declared before it is referenced.

```
Context             := Identifier [ {Parameters} ] : CONTEXT = ContextBody
Parameters          := [ TypeDecls ] ; {VarDecls}*;
TypeDecls           := {Identifier}+, : TYPE
ContextBody         := BEGIN Declarations END
Declarations        := {Declaration ;}+
Declaration         := ConstantDeclaration
                     | TypeDeclaration
                     | AssertionDeclaration
                     | ContextDeclaration
                     | ModuleDeclaration
ConstantDeclaration := Identifier [ (VarDecls) ] : Type [ = Expression ]
TypeDeclaration     := Identifier : TYPE [ = TypeDef ]
AssertionDeclaration := Identifier : AssertionForm = AssertionExpression
AssertionForm       := OBLIGATION | CLAIM | LEMMA | THEOREM
ContextDeclaration  := Identifier : CONTEXT = Identifier{ActualParameters}
ActualParameters    := {Type}*, ; {Expression}*,
```

Figure 66 SAL Context

APPENDIX B

SAL Generated Model Snippets

SAL Context

FrSM_Comp: CONTEXT =

BEGIN

END

Bounded Variables

%% Max Limits

FrSMDelayStartupWithoutWakeup_idx: INTEGER = 255;

FrSMNumWakeupPatterns_idx: INTEGER = 255;

FrSMStartupRepetitions_idx: INTEGER = 255;

FrSMStartupRepetitionsWithWakeup_idx: INTEGER = 255;

FrSMIsDualChannelNode_idx: INTEGER = 2;

FrSMIsColdstartEcu_idx: INTEGER = 2;

FrSMCheckWakeupReason_idx: INTEGER = 2;

FrSMIsWakeupEcu_idx: INTEGER = 2;

wakeupCounter_idx: INTEGER = 255;

t_Trcv_StdbyDelay_idx: INTEGER = 65536;

t3_idx: INTEGER = 2;

t2_idx: INTEGER = 2;

t1_idx: INTEGER = 2;

t_TrcvStdby_Delay_IsActive_idx: INTEGER = 2;

t3_IsNotActive_idx: INTEGER = 2;

t1_IsActive_idx: INTEGER = 2;

AllChannelIsAwake_idx: INTEGER = 2;

WUReason_idx: INTEGER = 3;

busTrafficDetected_idx: INTEGER = 2;

wakeupTransmitted_idx: INTEGER = 3;

startupCounter_idx: INTEGER = 255;

reqComMode_idx: INTEGER = 3;

Bounded Types

t_Trcv_StdbyDelay_type: TYPE = [0..t_Trcv_StdbyDelay_idx];

t_TrcvStdby_Delay_IsActive_type: TYPE = [0..t_TrcvStdby_Delay_IsActive_idx];

AllChannelIsAwake_type: TYPE = [0..AllChannelIsAwake_idx];

FrSMDelayStartupWithoutWakeup_type: TYPE = [0..FrSMDelayStartupWithoutWakeup_idx];

FrSMNumWakeupPatterns_type: TYPE = [0..FrSMNumWakeupPatterns_idx];

FrSMStartupRepetitions_type: TYPE = [0..FrSMStartupRepetitions_idx];

FrSMStartupRepetitionsWithWakeup_type: TYPE = [0..FrSMStartupRepetitionsWithWakeup_idx];

FrSMIsDualChannelNode_type: TYPE = [0..FrSMIsDualChannelNode_idx];

FrSMIsColdstartEcu_type: TYPE = [0..FrSMIsColdstartEcu_idx];

FrSMCheckWakeupReason_type: TYPE = [0..FrSMCheckWakeupReason_idx];

FrSMIsWakeupEcu_type: TYPE = [0..FrSMIsWakeupEcu_idx];

wakeupCounter_type: TYPE = [0..wakeupCounter_idx];

startupCounter_type: TYPE = [0..startupCounter_idx];

Defined States

%% States:

ST_FrSM : TYPE = {

ST_FRSM_ONLINE,

ST_FRSM_ONLINE_PASSIVE,

ST_FRSM_WAKEUP,

ST_FRSM_STARTUP,

ST_FRSM_HALT_REQ,

ST_FRSM_READY,

ST_FRSM_INIT

```
                    };
```

Defined Types

```
        %% Types:

        wakeup_Type : TYPE = {
                            SingleChannelWakeup,
                            DualChannelWakeup,
                            DualChannelWakeupForward,
                            DualChannelEchoWakeup,
                            NoWakeup
        };

        WUReason_type : TYPE = {
                            NO_WU_BY_BUS,
                            PARTIAL_WU_BY_BUS,
                            ALL_WU_BY_BUS
        };

        ComM_ModType : TYPE = {
                            NoCom,
                            SilentCom,
                            FullCom
        };

        REC_FrSM_Config : TYPE = [#
                wakeupType : wakeup_Type,
                FrSMDelayStartupWithoutWakeup : BOOLEAN,
                FrSMNumWakeupPatterns : FrSMNumWakeupPatterns_type,
                FrSMStartupRepetitions : FrSMStartupRepetitions_type,
```

```
                FrSMStartupRepetitionsWithWakeup                    :
FrSMStartupRepetitionsWithWakeup_type,
                FrSMDurationT1 :FrSMDurationT1_type,
                FrSMDurationT2 :FrSMDurationT2_type,
                FrSMDurationT3 :FrSMDurationT3_type
        #];

        REC_FrSM : TYPE = [#
                FrSm_Config : REC_FrSM_Config,
                t_Trcv_StdbyDelay : t_Trcv_StdbyDelay_type,
                t3 : BOOLEAN,
                t2 : BOOLEAN,
                t1 : BOOLEAN,
                t_TrcvStdby_Delay_IsActive : BOOLEAN,
                t3_IsNotActive : BOOLEAN,
                t1_IsActive : BOOLEAN,
                AllChannelIsAwake : BOOLEAN,
                WUReason : WUReason_type,
                FrSMDelayStartupWithoutWakeup : BOOLEAN,
                FrSMNumWakeupPatterns : FrSMNumWakeupPatterns_type,
                FrSMStartupRepetitions : FrSMStartupRepetitions_type,
                FrSMStartupRepetitionsWithWakeup                    :
FrSMStartupRepetitionsWithWakeup_type,
                FrSMIsDualChannelNode : BOOLEAN,
                FrSMIsColdstartEcu : BOOLEAN,
                FrSMCheckWakeupReason : BOOLEAN,
                FrSMIsWakeupEcu : BOOLEAN,
                wakeupCounter : wakeupCounter_type,
                busTrafficDetected : BOOLEAN,
```

```
            wakeupTransmitted : BOOLEAN,
            wakeupType : wakeup_Type,
            startupCounter : startupCounter_type,
            reqComMode : ComM_ModType
        #];
```

Events

```
        %% Events
        EVT_FrSM_Comp: TYPE = {
                                EVT_T09,
                                EVT_T31,
                                EVT_T03,
                                EVT_T05,
                                EVT_T14,
                                EVT_T33,
                                EVT_T17,
                                EVT_T10,
                                EVT_T08,
                                EVT_T32,
                                EVT_T06,
                                EVT_T04,
                                EVT_T02,
                                EVT_T12,
                                EVT_T13,
                                EVT_T01,
                                EVT_T11,
                                EVT_T00
        };
```

Module Definitions

```
        MOD_FrSM : MODULE =
```

BEGIN

%% Global Section

GLOBAL FrSM: REC_FrSM

GLOBAL EVT: EVT_FrSM_Comp

GLOBAL FrSM_State: ST_FrSM

INITIALIZATION

FrSM_State = ST_FRSM_INIT;

FrSM = (# t_Trcv_StdbyDelay := 0, t3 := FALSE, t2 := FALSE, t1 := FALSE, t_TrcvStdby_Delay_IsActive := FALSE, t3_IsNotActive := FALSE, t1_IsActive := FALSE, AllChannelIsAwake := FALSE, WUReason := NO_WU_BY_BUS, FrSMDelayStartupWithoutWakeup := FALSE, FrSMNumWakeupPatterns := 0, FrSMStartupRepetitions := 0, FrSMStartupRepetitionsWithWakeup := 0, FrSMIsDualChannelNode := FALSE, FrSMIsColdstartEcu := FALSE, FrSMCheckWakeupReason := FALSE, FrSMIsWakeupEcu := FALSE, wakeupCounter := 0, busTrafficDetected := FALSE, wakeupTransmitted := FALSE, wakeupType := NoWakeup , startupCounter := 2, reqComMode := NoCom #);

TRANSITION

[

%% stateName = ST_FRSM_ONLINE

%% event = EVT_T09

(FrSM_State = ST_FRSM_ONLINE) AND (EVT = EVT_T09) -->

FrSM_State' = ST_FRSM_HALT_REQ;

```
%% event = EVT_T10
[]
( FrSM_State = ST_FRSM_ONLINE ) AND (EVT = EVT_T10) -->
FrSM_State' = ST_FRSM_STARTUP;
%% stateName = ST_FRSM_ONLINE_PASSIVE
%% event = EVT_T14
[]
( FrSM_State = ST_FRSM_ONLINE_PASSIVE ) AND (EVT = EVT_T14) -->
FrSM_State' = ST_FRSM_HALT_REQ;
%% event = EVT_T33
[]
( FrSM_State = ST_FRSM_ONLINE_PASSIVE ) AND (EVT = EVT_T33) -->
FrSM_State' = ST_FRSM_ONLINE_PASSIVE;
%% event = EVT_T17
[]
( FrSM_State = ST_FRSM_ONLINE_PASSIVE ) AND (EVT = EVT_T17) -->
FrSM_State' = ST_FRSM_STARTUP;
%% stateName = ST_FRSM_WAKEUP
%% event = EVT_T31
[]
( FrSM_State = ST_FRSM_WAKEUP ) AND (EVT = EVT_T31) -->
FrSM_State' = ST_FRSM_WAKEUP;
%% event = EVT_T03
 []
( FrSM_State = ST_FRSM_WAKEUP ) AND (EVT = EVT_T03) -->
FrSM_State' = ST_FRSM_STARTUP;
```

```
%% event = EVT_T13
[]
( FrSM_State = ST_FRSM_WAKEUP ) AND (EVT = EVT_T13) -->
FrSM_State' = ST_FRSM_READY;
%% stateName = ST_FRSM_STARTUP
%% event = EVT_T05
[]
( FrSM_State = ST_FRSM_STARTUP ) AND (EVT = EVT_T05) -->
FrSM_State' = ST_FRSM_WAKEUP;
%% event = EVT_T08
[]
( FrSM_State = ST_FRSM_STARTUP ) AND (EVT = EVT_T08) -->
FrSM_State' = ST_FRSM_ONLINE;
%% event = EVT_T32
[]
( FrSM_State = ST_FRSM_STARTUP ) AND (EVT = EVT_T32) -->
FrSM_State' = ST_FRSM_STARTUP;
%% event = EVT_T06
[]
( FrSM_State = ST_FRSM_STARTUP ) AND (EVT = EVT_T06) -->
FrSM_State' = ST_FRSM_STARTUP;
%% event = EVT_T04
[]
( FrSM_State = ST_FRSM_STARTUP ) AND (EVT = EVT_T04) -->
FrSM_State' = ST_FRSM_STARTUP;
%% event = EVT_T12
 []
( FrSM_State = ST_FRSM_STARTUP ) AND (EVT = EVT_T12) -->
FrSM_State' = ST_FRSM_READY;
```

%% stateName = ST_FRSM_HALT_REQ

%% event = EVT_T11

[]

(FrSM_State = ST_FRSM_HALT_REQ) AND (EVT = EVT_T11) -->

FrSM_State' = ST_FRSM_READY;

%% stateName = ST_FRSM_READY

%% event = EVT_T02

[]

(FrSM_State = ST_FRSM_READY) AND (EVT = EVT_T02) -->

FrSM_State' = ST_FRSM_STARTUP;

%% event = EVT_T01

[]

(FrSM_State = ST_FRSM_READY) AND (EVT = EVT_T01) AND (FrSM.reqComMode = FullCom) AND (FrSM.WUReason = NO_WU_BY_BUS) AND (FrSM.FrSMIsWakeupEcu = TRUE) AND (FrSM.FrSMIsDualChannelNode = FALSE) -->

FrSM_State' = ST_FRSM_WAKEUP;

FrSM'.startupCounter = 1;

FrSM'.wakeupType = SingleChannelWakeup;

FrSM'.wakeupTransmitted = FALSE;

FrSM'.t1 = TRUE;

FrSM'.t3 = TRUE;

%% stateName = ST_FRSM_INIT

%% event = EVT_T00

[]

(FrSM_State = ST_FRSM_READY) AND (EVT = EVT_T01) AND (FrSM.reqComMode = FullCom) AND (FrSM.WUReason = NO_WU_BY_BUS) AND (FrSM.FrSMIsWakeupEcu = TRUE) AND (FrSM.FrSMIsDualChannelNode = TRUE) -->

FrSM_State' = ST_FRSM_WAKEUP;

FrSM'.startupCounter = 1;

FrSM'.wakeupType = DualChannelWakeup;

FrSM'.wakeupTransmitted = FALSE;

FrSM'.t1 = TRUE;

FrSM'.t3 = TRUE;

[]

(FrSM_State = ST_FRSM_READY) AND (EVT = EVT_T01) AND (FrSM.reqComMode = FullCom) AND (FrSM.WUReason = PARTIAL_WU_BY_BUS) AND (FrSM.FrSMIsWakeupEcu = TRUE) -->

FrSM_State' = ST_FRSM_WAKEUP;

FrSM'.startupCounter = 1;

FrSM'.wakeupType = DualChannelWakeupForward;

FrSM'.wakeupTransmitted = FALSE;

FrSM'.t3 = TRUE;

%% stateName = ST_FRSM_INIT

%% event = EVT_T00

[]

(FrSM_State = ST_FRSM_READY) AND (EVT = EVT_T01) AND (FrSM.reqComMode = FullCom) AND (FrSM.WUReason = ALL_WU_BY_BUS OR FrSM.FrSMIsWakeupEcu = FALSE) AND (FrSM.FrSMDelayStartupWithoutWakeup = FALSE) -->

FrSM_State' = ST_FRSM_WAKEUP;

FrSM'.startupCounter = 1;

FrSM'.wakeupType = NoWakeup;

FrSM'.t2 = TRUE;

FrSM'.t3 = TRUE;

%% stateName = ST_FRSM_INIT

%% event = EVT_T00

[]

(FrSM_State = ST_FRSM_READY) AND (EVT = EVT_T01) AND (FrSM.reqComMode = FullCom) AND (FrSM.WUReason = ALL_WU_BY_BUS OR FrSM.FrSMIsWakeupEcu = FALSE) AND (FrSM.FrSMDelayStartupWithoutWakeup = TRUE) -->

FrSM_State' = ST_FRSM_WAKEUP;

FrSM'.startupCounter = 1;

FrSM'.wakeupType = NoWakeup;

FrSM'.t1 = TRUE;

FrSM'.t2 = TRUE;

FrSM'.t3 = TRUE;

%% stateName = ST_FRSM_INIT

%% event = EVT_T00

[]

```
( FrSM_State = ST_FRSM_INIT ) AND (EVT = EVT_T00) -->
        FrSM_State' = ST_FRSM_READY;
]
END;
```

Sample Theorems

```
Safe: THEOREM SYS_TwoTraffics |- G(NOT(ARR_ST_TrafficLight[1] =
ST_TrafficLight_Green AND ARR_ST_TrafficLight[2] =
ST_TrafficLight_Green));

Safe_FrSM_6: THEOREM system |- G(FrSM.reqComMode = FullCom AND
((FrSM.WUReason = ALL_WU_BY_BUS) OR (FrSM.FrSMIsWakeupEcu =
FALSE)) AND (FrSM.FrSMDelayStartupWithoutWakeup = FALSE) AND
FrSM_State = ST_FRSM_READY);

Safe_FrSM_7:
THEOREM system |- G(FrSM.startupCounter <= 255 AND
FrSM.startupCounter >= 0 );
```

SAL ATC Model

```
ATC: CONTEXT =
BEGIN
        % ------------------------
        % Max Limits
        vehicleSpeed_idx: INTEGER = 240;
        upThreshold_idx: INTEGER = 240;
        downThreshold_idx: INTEGER = 240;
        gearTimeTHreshold_idx: INTEGER = 20;
        timerStarted_idx: INTEGER =1;
        gearTimer_idx: INTEGER = 20;
```

```
turn_idx: INTEGER = 2;


% Bounded Ranges
vehicleSpeed_type: TYPE = [0..vehicleSpeed_idx];
        upThreshold_type: TYPE = [0..upThreshold_idx];
        downThreshold_type: TYPE = [0..downThreshold_idx];
        gearTimeTHreshold_type: TYPE = [0..gearTimeTHreshold_idx];
        gearTimer_type: TYPE = [0..gearTimer_idx];
        turn_type: TYPE = [0..turn_idx];

%% States:

ST_GEARPOSITION: TYPE = {
        ST_POSITION1,
        ST_POSITION2,
        ST_POSITION3,
        ST_POSITION4
};

ST_GEARCONTROLLER: TYPE = {
        ST_STEADY,
        ST_UPSHIFTING,
        ST_DOWNSHIFTING
};

%% Events:

EVT_ATC: TYPE = {
```

```
        EVT_CHECKINPUT,
        EVT_SPEEDLESSDOWNTHROTTLE,
        EVT_SPEEDMOREDOWNTHROTTLE,
        EVT_TIMEELASPEGEARDOWN,
        EVT_SPEEDMOREUPTHROTTLE,
        EVT_SPEEDLESSUPTHROTTLE,
        EVT_TIMEELASPEGEARUP,
        EVT_UP,
        EVT_DOWN
};

% ------------------------

%% Class Records:

REC_GEARCONTROLLER: TYPE = [#
        vehicleSpeed: vehicleSpeed_type,
        gearTimeTHreshold: gearTimeTHreshold_type,
        timerStarted:BOOLEAN,
        gearTimer: gearTimer_type

#];

REC_GEARPOSITION: TYPE = [#
        upThreshold: upThreshold_type,
        downThreshold: downThreshold_type
#];
```

```
% ------------------------

%% System Records:

Global_EVT: TYPE = [#
        ATC_System_EVT: EVT_ATC
#];

% ------------------------
MOD_GearController : MODULE =
BEGIN

GLOBAL CONT: REC_GEARCONTROLLER
INPUT Position: REC_GEARPOSITION
GLOBAL EVT: EVT_ATC
GLOBAL State: ST_GEARCONTROLLER

INITIALIZATION
        CONT = (# vehicleSpeed := 20, gearTimeTHreshold := 10,
timerStarted := FALSE, gearTimer := 20      #);
        EVT = EVT_CHECKINPUT;
        State = ST_STEADY;

TRANSITION
[
        (State= ST_STEADY) AND (CONT.vehicleSpeed <
Position.downThreshold) -->
          EVT'= EVT_SPEEDLESSDOWNTHROTTLE;
        []
```

(State = ST_STEADY) AND (EVT = EVT_SPEEDLESSDOWNTHROTTLE) -->

State' = ST_DOWNSHIFTING;

[]

(State = ST_DOWNSHIFTING AND CONT.timerStarted = FALSE) -->

CONT'.timerStarted = TRUE;

[]

(State = ST_DOWNSHIFTING AND CONT.timerStarted = TRUE AND EVT = EVT_SPEEDLESSDOWNTHROTTLE) -->

EVT' = IF CONT.timerStarted = TRUE THEN EVT_TIMEELASPEGEARUP ELSE EVT_SPEEDLESSDOWNTHROTTLE ENDIF;

CONT'.timerStarted = FALSE;

[]

(State = ST_DOWNSHIFTING) AND (EVT = EVT_TIMEELASPEGEARDOWN) -->

EVT' = IF (CONT.vehicleSpeed <= Position.downThreshold) THEN EVT_DOWN ELSE EVT_SPEEDMOREDOWNTHROTTLE ENDIF;

[]

(State = ST_DOWNSHIFTING) AND (EVT = EVT_CHECKINPUT) -->

State' = ST_STEADY;

[]

(State = ST_DOWNSHIFTING) AND (EVT = EVT_SPEEDMOREDOWNTHROTTLE) -->

State' = ST_STEADY;

EVT' = EVT_CHECKINPUT;

```
                    []
                    (State= ST_STEADY) AND (CONT.vehicleSpeed >
Position.upThreshold) -->
                      EVT' = EVT_SPEEDMOREUPTHROTTLE;
                    []
                    (State = ST_STEADY) AND (EVT =
EVT_SPEEDMOREUPTHROTTLE) -->
                    State' = ST_UPSHIFTING;
                []
                    (State = ST_UPSHIFTING AND CONT.timerStarted =
FALSE ) -->
                    CONT'.timerStarted = TRUE;
                    []
                    (State = ST_UPSHIFTING AND CONT.timerStarted =
TRUE AND EVT = EVT_SPEEDMOREUPTHROTTLE) -->
                    EVT' = IF CONT.timerStarted = TRUE THEN
EVT_TIMEELASPEGEARUP ELSE EVT_SPEEDMOREUPTHROTTLE ENDIF;

                    CONT'.timerStarted = FALSE;
                    []
                    (State = ST_UPSHIFTING) AND (EVT=
EVT_TIMEELASPEGEARUP)-->
                    EVT' = IF CONT.vehicleSpeed > Position.upThreshold
THEN EVT_UP ELSE EVT_SPEEDLESSUPTHROTTLE ENDIF;
                    []
                    (State = ST_UPSHIFTING) AND (EVT =
EVT_CHECKINPUT) -->
                    State' = ST_STEADY;
                    []
```

```
                    (State = ST_UPSHIFTING) AND (EVT =
EVT_SPEEDLESSUPTHROTTLE) -->
                    State' = ST_STEADY;
                    EVT' = EVT_CHECKINPUT;
                    [] ELSE --> EVT' = EVT_CHECKINPUT;
CONT'.vehicleSpeed = CONT.vehicleSpeed +1;
                ]
        END;

        MOD_GearPosition: MODULE =
        BEGIN

        GLOBAL Position: REC_GEARPOSITION
        GLOBAL PState: ST_GEARPOSITION
        GLOBAL EVT: EVT_ATC

        INITIALIZATION
                    PState = ST_POSITION1;
                    Position = (# upThreshold := 21, downThreshold := 0
    #)      ;



        TRANSITION [
                    (PState = ST_POSITION1) AND (EVT = EVT_UP)  -->
                        PState' = ST_POSITION2;
                        Position'.upThreshold = 50;
                        Position'.downThreshold = 21;
                        EVT' = EVT_CHECKINPUT;
```

```
[]
(PState = ST_POSITION2) AND (EVT = EVT_UP) -->
        PState' = ST_POSITION3;
        Position'.upThreshold = 70;
        Position'.downThreshold = 50;
        EVT' = EVT_CHECKINPUT;


[]
(PState = ST_POSITION3) AND (EVT = EVT_UP)  -->
        PState' = ST_POSITION4;
        Position'.upThreshold = 100;
        Position'.downThreshold = 71;
        EVT' = EVT_CHECKINPUT;

[]
(PState = ST_POSITION4) AND (EVT = EVT_DOWN) --
>
        PState' = ST_POSITION3;
        Position'.upThreshold = 70;
        Position'.downThreshold = 51;
        EVT' = EVT_CHECKINPUT;

[]
(PState = ST_POSITION3) AND (EVT = EVT_DOWN)  -
->
        PState' = ST_POSITION2;
```

```
                    Position'.upThreshold = 50;
                    Position'.downThreshold = 21;
                    EVT' = EVT_CHECKINPUT;

                []
                (PState = ST_POSITION2) AND (EVT = EVT_DOWN) --
>
                    PState' = ST_POSITION1;
                    Position'.upThreshold = 21;
                    Position'.downThreshold = 1;
                    EVT'= EVT_CHECKINPUT;

            ]

        END;

        %% System Module:

        system: MODULE = MOD_GearController [] MOD_GearPosition;

        %% Boundary Conditions
        Safe_Boundary_Th1: THEOREM system |- G(Position.upThreshold <
235);
        Safe_Boundary_Th2: THEOREM system |- G(Position.downThreshold <
235);
```

```
Safe_Boundary_Th3: THEOREM system |- G(Position.upThreshold < 255);

%% Requirement Mapping
%% AG : p is globally true  AF - For all paths ps is true
Req1_Th1: THEOREM system |- AG(PState = ST_POSITION1 => AF(CONT.vehicleSpeed > 0 AND  CONT.vehicleSpeed <= 21));
Req1_Th2: THEOREM system |- AG(NOT (PState = ST_POSITION2) OR (CONT.vehicleSpeed > 21 AND CONT.vehicleSpeed <= 50));
Req1_Th3: THEOREM system |- G((PState = ST_POSITION2) <=> F(CONT.vehicleSpeed > 21 AND CONT.vehicleSpeed < 50));

%%Logic Bugs
%% Cannot be in downthreshold unless vehcile speed is less than down threshold
%% G p is always true while F that p will eventuually be true
Logic_Th1: THEOREM system |- G( State = ST_DOWNSHIFTING => F(CONT.vehicleSpeed < Position.downThreshold));

%% Timer check
%% Has to be off in steady state
Timer_Th1: THEOREM system |- G( State = ST_STEADY => F(CONT.timerStarted = FALSE));
Timer_Th2: THEOREM system |- G( State = ST_DOWNSHIFTING => F(CONT.timerStarted = TRUE AND CONT.gearTimeTHreshold<20));
%% Cannot exceed timer thresold in downshifting or upshifting state


END
```

APPENDIX C

SAL Model Compiler

State Machine Marking

```
.//
.include "${te_file.arc_path}/t.sm_sm.c"
.select one sm_sm related by o_obj->SM_ISM[R518]->SM_SM[R517]
.select many instance_sm_states related by sm_sm->SM_STATE[R501]
.for each sm_state in instance_sm_states
  .select one te_state related by sm_state->TE_STATE[R2037]
  .select any sm_crtxn related by sm_state->SM_TXN[R506]->SM_CRTXN[R507]
where ( selected.SMspd_ID == sm_state.SMspd_ID )
  .select one sm_act related by sm_state->SM_MOAH[R511]->SM_AH[R513]-
>SM_ACT[R514]
  .select one te_act related by sm_act->TE_ACT[R2022]
  .if ( not_empty te_act )
    .select one te_aba related by te_act->TE_ABA[R2010]
    .// CDS relaxed same data needed
    .select any sm_txn related by sm_state->SM_TXN[R506]
    .invoke red = TE_EVT_ReceivedEventDataDeclaration( sm_txn, sm_act )
    .assign received_event_declaration = red.body
    .include "${te_file.arc_path}/t.class.sm_act.c"
  .end if
.end for
.select any sm_crtxn from instances of SM_CRTXN where ( false )
.select any te_state from instances of TE_STATE where ( false )
.//
.select many instance_sm_txns related by sm_sm->SM_TXN[R505]
```

```
.for each sm_txn in instance_sm_txns
  .select one sm_act related by sm_txn->SM_TAH[R530]->SM_AH[R513]->SM_ACT[R514]
  .select one te_act related by sm_act->TE_ACT[R2022]
  .if ( not_empty te_act )
    .select one te_aba related by te_act->TE_ABA[R2010]
    .invoke red = TE_EVT_ReceivedEventDataDeclaration( sm_txn, sm_act )
    .assign received_event_declaration = red.body
    .include "${te_file.arc_path}/t.class.sm_act.c"
  .end if
.end for
.//
.select one sm_sm related by o_obj->SM_ASM[R519]->SM_SM[R517]
.select many class_sm_states related by sm_sm->SM_STATE[R501]
.for each sm_state in class_sm_states
  .select one te_state related by sm_state->TE_STATE[R2037]
  .select one sm_act related by sm_state->SM_MOAH[R511]->SM_AH[R513]->SM_ACT[R514]
  .select one te_act related by sm_act->TE_ACT[R2022]
  .if ( not_empty te_act )
    .select one te_aba related by te_act->TE_ABA[R2010]
    .// CDS relaxed same data needed
    .select any sm_txn related by sm_state->SM_TXN[R506]
    .invoke red = TE_EVT_ReceivedEventDataDeclaration( sm_txn, sm_act )
    .assign received_event_declaration = red.body
    .include "${te_file.arc_path}/t.class.sm_act.c"
  .end if
.end for
.select any te_state from instances of TE_STATE where ( false )
```

```
.//
.select many class_sm_txns related by sm_sm->SM_TXN[R505]
.for each sm_txn in class_sm_txns
 .select one sm_act related by sm_txn->SM_TAH[R530]->SM_AH[R513]->SM_ACT[R514]
 .select one te_act related by sm_act->TE_ACT[R2022]
 .if ( not_empty te_act )
  .select one te_aba related by te_act->TE_ABA[R2010]
  .invoke red = TE_EVT_ReceivedEventDataDeclaration( sm_txn, sm_act )
  .assign received_event_declaration = red.body
  .include "${te_file.arc_path}/t.class.sm_act.c"
 .end if
.end for
.//
.if ( ( ( empty instance_sm_states ) and ( empty class_sm_states ) ) and ( ( empty instance_sm_txns ) and ( empty class_sm_txns ) ) )
/*
 * This class is modeled as having a state chart, but it has no states.
 * This makes good sense in a supertype class receiving polymorphic events.
 * If this is not the intention, add states to the model or unmark the
 * instance or class state chart setting.
 */
static void empty_state_chart_action( void );
static void empty_state_chart_action( void ) {}
.end if
.//
```

Classes Marking

```
.select many te_classes related by te_c->TE_CLASS[R2064] where ( not selected.ExcludeFromGen )
.// Prepare the Instance subsystem for translation.
.select any i_ins from instances of I_INS
.if ( not_empty i_ins )
  .select many o_objs related by te_classes->O_OBJ[R2019]
  .invoke PEIInstanceSubsystemInit( o_objs )
.end if
.for each te_class in te_classes
  .// Generate declaration implementation file.
  .invoke implementation = CreateObjectImplementation( te_class, te_c, true )
  ${implementation.body}
  .emit to file "${te_file.domain_include_path}/${te_class.class_file}.${te_file.hdr_file_ext}"
  .//
  .// Generate definition implementation.
  .invoke implementation = CreateObjectImplementation( te_class, te_c, false )
${implementation.body}
  .emit to file "${te_file.domain_source_path}/${te_class.class_file}.${te_file.src_file_ext}"
.end for
.//
```

DataTypes Marking

```
.//=============================================================================
==============
.function GetBaseTypeForUDT .// s_dt
  .param inst_ref s_udt
  .select one s_dt related by s_udt->S_DT[R18]
```

```
  .select one s_udt related by s_dt->S_UDT[R17]
  .if ( not_empty s_udt )
    .invoke r = GetBaseTypeForUDT( s_udt )
    .assign s_dt = r.result
  .end if
  .assign attr_result = s_dt
.end function
.//
.//=============================================================
==============
.//   Get the S_DT and S_CDT object references for a given attribute
.//   (O_ATTR) instance.
.//=============================================================
==============
.function GetAttributeCodeGenType .// te_dt
  .param inst_ref o_attr
  .//
  .select one s_dt related by o_attr->S_DT[R114]
  .select one s_udt related by s_dt->S_UDT[R17]
  .if ( not_empty s_udt )
    .invoke r = GetBaseTypeForUDT( s_udt )
    .assign s_dt = r.result
  .end if
  .select one te_dt related by s_dt->TE_DT[R2021]
  .select one s_cdt related by s_dt->S_CDT[R17]
  .//
  .if ( empty s_cdt )
    .select one s_edt related by s_dt->S_EDT[R17]
    .if ( empty s_edt )
```

```
    .select one s_sdt related by s_dt->S_SDT[R17]
    .if ( empty s_sdt )
     .select one s_irdt related by s_dt->S_IRDT[R17]
     .if ( empty s_irdt )
      .print "Error in attribute ${o_attr.Name}"
      .print "with data type ${s_dt.Name}"
      .exit 100
     .end if
    .end if
   .else
    .// Enum, use integer type.
    .// CDS Some day we should pass along the enumeration type.
    .select any s_cdt from instances of S_CDT where ( selected.Core_Typ == 2 )
   .end if
  .end if
  .//
  .if ( not_empty s_cdt )
   .if ( 7 == s_cdt.Core_Typ )
    .// s_cdt.Core_Typ is "same_as<Base_Attribute>"
    .select one base_o_attr related by o_attr->O_RATTR[R106]->O_BATTR[R113]->O_ATTR[R106]
    .if ( empty base_o_attr )
     .select one o_obj related by o_attr->O_OBJ[R102]
     .print "\nCould not find O_BATTR for object ${o_obj.Name} (${o_obj.Key_Lett}) attribute ${o_attr.Name} !"
     .print "\nDid you combine a referential and then rename the combined attribute?"
     .exit 101
    .end if
    .// Note: the following is a recursive call to this function
```

```
      .invoke r = GetAttributeCodeGenType( base_o_attr )
      .assign te_dt = r.result
    .end if
  .end if
  .assign attr_result = te_dt
.end function
.//
.//============================================================
==============
.// Map a user defined data types precision into its corresponding instance
.// of Data Type (S_DT).
.// Note:  Might prefer POSIX type support here, but doubt we can count
.// on most embedded targets thinking this way.  Thus brute force the types.
.//============================================================
==============
.function MapUserSpecifiedDataTypePrecision .// boolean
  .param inst_ref te_dt
  .param string mapping
  .assign error = false
  .assign type = mapping
  .if ( (type == "uchar_t") or ((type == "u_char") or (type == "unsignedchar")) )
    .assign te_dt.ExtName      = "unsigned char"
  .elif ( (type == "char_t") or (type == "char") )
    .assign te_dt.ExtName      = "char"
  .elif ( type == "signedchar" )
    .assign te_dt.ExtName      = "signed char"
  .elif ( (type == "ushort_t") or ((type == "u_short") or (type == "unsignedshort")) )
    .assign te_dt.ExtName      = "unsigned short"
  .elif ( (type == "short_t") or (type == "short") )
```

```
  .assign te_dt.ExtName      = "short"
.elif ( type == "signedshort" )
  .assign te_dt.ExtName      = "signed short"
.elif ( (type == "uint_t") or ((type == "u_int") or (type == "unsignedint")) )
  .assign te_dt.ExtName      = "unsigned int"
.elif ( type == "s1_t" )
  .assign te_dt.ExtName      = "s1_t"
.elif ( type == "u1_t" )
  .assign te_dt.ExtName      = "u1_t"
.elif ( type == "s2_t" )
  .assign te_dt.ExtName      = "s2_t"
.elif ( type == "u2_t" )
  .assign te_dt.ExtName      = "u2_t"
.elif ( type == "s4_t" )
  .assign te_dt.ExtName      = "s4_t"
.elif ( type == "u4_t" )
  .assign te_dt.ExtName      = "u4_t"
.elif ( type == "i_t" )
  .assign te_dt.ExtName      = "i_t"
.elif ( (type == "int_t") or (type == "int") )
  .assign te_dt.ExtName      = "int"
.elif ( type == "signedint" )
  .assign te_dt.ExtName      = "signed int"
.elif ( (type == "ulong_t") or ((type == "u_long") or (type == "unsignedlong")) )
  .assign te_dt.ExtName      = "unsigned long"
.elif ( (type == "long_t") or (type == "long") )
  .assign te_dt.ExtName      = "long"
.elif ( type == "signedlong" )
  .assign te_dt.ExtName      = "signed long"
```

```
 .elif ( (type == "u_longlong_t") or ((type == "u_longlong_t") or (type == "unsignedlonglong")) )
   .assign te_dt.ExtName      = "unsigned long long"
 .elif ( (type == "longlong_t") or ((type == "longlong") or (type == "signedlonglong")) )
   .assign te_dt.ExtName      = "long long"
   .//
 .elif ( type == "float" )
   .assign te_dt.ExtName      = "float"
 .elif ( type == "r4_t" )
   .assign te_dt.ExtName      = "r4_t"
 .elif ( type == "double" )
   .assign te_dt.ExtName      = "double"
 .elif ( type == "r8_t" )
   .assign te_dt.ExtName      = "r8_t"
   .//
 .elif ( type == "size_t" )
   .assign te_dt.ExtName      = "size_t"
 .elif ( type == "ssize_t" )
   .assign te_dt.ExtName      = "ssize_t"
 .elif ( type == "time_t" )
   .assign te_dt.ExtName      = "time_t"
 .elif ( type == "clock_t" )
   .assign te_dt.ExtName      = "clock_t"
 .elif ( type == "volatile_clock_t" )
   .assign te_dt.ExtName      = "volatile unsigned long"
   .//
 .else
   .assign error = true
 .end if
```

```
  .assign attr_result = error
.end function
.//
.// Return the structure type for persistent links.
.function UserSuppliedDataTypeIncludes .// string
  .select any te_file from instances of TE_FILE
  .assign sys_types_file_name = ( te_file.types + "." ) + te_file.hdr_file_ext
  .select many special_te_dts from instances of TE_DT where ( ( selected.Include_File !=
"" ) and ( selected.Include_File != sys_types_file_name ) )
  .assign s = ""
  .for each special_te_dt in special_te_dts
    .assign s = ( s + "#include """ ) + ( special_te_dt.Include_File + """\n" )
    .invoke oal( "s = Escher_strcpy( s, Escher_stradd( Escher_stradd( s, #include  ),
Escher_stradd( special_te_dt->Include_File, \n ) ) ); // Ccode" )
  .end for
  .assign attr_result = s
.end function
.//
```

Action Language Marking

```
.//
.function TE_ABA_rollup
  .invoke oal( "char b[1000000]; // Ccode" )
  .assign parseSuccessful = ( 1 ) .COMMENT ParseStatus::parseSuccessful
  .select any empty_act_blk from instances of ACT_BLK where ( false )
  .select many te_cs from instances of TE_C where ( selected.included_in_build )
  .for each te_c in te_cs
    .select many te_abas related by te_c->TE_ABA[R2088]
    .for each te_aba in te_abas
      .select one te_blk related by te_aba->TE_BLK[R2011]
```

```
    .if ( not_empty te_blk )
      .invoke oal( "te_aba->code = &b[0]; *te_aba->code = 0; // Ccode" )
      .invoke blck_xlate( te_c.StmtTrace, te_blk, te_aba )
      .invoke oal( "te_aba->code = Escher_strcpy( te_aba->code, &b[0] ); // Ccode" )
    .else
      .assign te_aba.code = ( "\n  /" + "* WARNING!  Skipping unsuccessful or unparsed action.  *" ) + "/\n"
    .end if
   .end for
  .end for
  .// Process EEs outside of components.
  .select many te_ees from instances of TE_EE where ( ( selected.RegisteredName != "TIM" ) and selected.Included )
  .for each te_ee in te_ees
   .select one te_c related by te_ee->TE_C[R2085]
   .if ( empty te_c )
    .select many s_brgs related by te_ee->S_EE[R2020]->S_BRG[R19]
    .for each s_brg in s_brgs
     .select one act_blk related by s_brg->ACT_BRB[R697]->ACT_ACT[R698]->ACT_BLK[R666]
     .select one te_aba related by s_brg->TE_BRG[R2025]->TE_ABA[R2010]
     .if ( not_empty act_blk )
      .select one te_blk related by act_blk->TE_BLK[R2016]
      .invoke oal( "te_aba->code = &b[0]; *te_aba->code = 0; // Ccode" )
      .invoke blck_xlate( false, te_blk, te_aba )
      .invoke oal( "te_aba->code = Escher_strcpy( te_aba->code, &b[0] ); // Ccode" )
     .end if
    .end for
   .end if
```

```
  .end for
.end function
```

SAL File Generation

```
.select any s_sys from instances of S_SYS
.select any c_cs from instances of C_C
%-------------------------------------------------------------------------
% File:  ${c_cs.Name}.sal
% SAL Generated Model
% Component/Module Name:  ${c_cs.Name}
%
%  Copyright - AUC 2017
% -------------------------------------------------------------------------


${c_cs.Name}: CONTEXT =
.print "Starting the generation of ${c_cs.Name} SAL Model"
BEGIN

.print "Generating SAL Bounded Values"
%% Max Limits
.select many o_ob from instances of O_OBJ
.for each o_obj in o_ob
.if (o_obj.Name != "Test")
        .select many o_attr related by o_obj->O_ATTR[R102]
        .for each attribute in o_attr
        .select one datatype related by attribute->S_DT[R114]
        .if (datatype.Name != "state<State_Model>" )
        ${attribute.Name}_idx: INTEGER = ${attribute.Descrip};
        .end if
```

```
        .end for
.end if
.end for


%% Bounded Ranges
.select many o_ob from instances of O_OBJ
.for each o_obj in o_ob
.if (o_obj.Name != "Test")
        .select many o_attr related by o_obj->O_ATTR[R102]
        .for each attribute in o_attr
        .select one datatype related by attribute->S_DT[R114]
        .if (datatype.Name != "state<State_Model>" )
        ${attribute.Name}_type: TYPE = [0..${attribute.Name}_idx];
        .end if
        .end for
.end if
.end for

.print "Generating SAL States Structures"
%% States:
.select many o_ob from instances of O_OBJ
.for each o_obj in o_ob
        .if (o_obj.Name != "Test")
                ST_${o_obj.Name} : TYPE = {
                        .select one sm_ism related by o_obj->SM_ISM[R518]
                                .if ( not_empty sm_ism )
                                        .select one sm_sm related by sm_ism-
>SM_SM[R517]
```

```
                                        .select many sm_states related by sm_sm->SM_STATE[R501]
                                        .assign objCount = cardinality sm_states
                                        .assign count = 1
                                        .for each sm_state in sm_states
                                                .if (count < objCount)
                                                        ST_${sm_state.Name},
                                                .else
                                                        ST_${sm_state.Name}
                                                .end if
                                                .assign count = count + 1
                                        .end for
                                .end if
                        };
                .end if
.end for

.print "Generating SAL Events mapping"
%% Events
EVT_${c_cs.Name}: TYPE = {
.assign obCount2 = 1
.select many o_ob from instances of O_OBJ
.assign objCount2 = cardinality o_ob
.for each o_obj in o_ob
        .select one sm_ism related by o_obj->SM_ISM[R518]
   .if ( not_empty sm_ism )
                .select one sm_sm related by sm_ism->SM_SM[R517]
                .select many sm_evt related by sm_sm->SM_EVT[R502]
                .assign objCount = cardinality sm_evt
```

```
                .assign count = 1
                .for each smevt in sm_evt
                .if ((count < objCount) AND (obCount2 != objCount2))
                        EVT_${smevt.Mning},
                .else
                .if ((count == objCount) AND (obCount2 == objCount2))
                        EVT_${smevt.Mning}
                .else
                        EVT_${smevt.Mning},
                .end if
                .end if
                .assign count = count + 1
                .end for
        .end if
        .assign obCount2 = obCount2 + 1
.end for
};

.print "Generating SAL Class Mapping"
.select many o_ob from instances of O_OBJ
.for each o_obj in o_ob
.if (o_obj.Name != "Test")
REC_${o_obj.Name} : TYPE = [#
        .select many o_attr related by o_obj->O_ATTR[R102]
        .assign objCount = cardinality o_attr
        .assign count = 1
        .for each attribute in o_attr
                .select one datatype related by attribute->S_DT[R114]
                .assign comma = 1
```

```
                .if (count < objCount)
                        .assign comma = 1
                .else
                        .assign comma = 0
                .end if
                .if ((datatype.Name == "integer" ) AND (comma == 1))
                        ${attribute.Name} : ${attribute.Name}_type,
                .elif ((datatype.Name == "integer" ) AND (comma == 0))
                        ${attribute.Name} : ${attribute.Name}_type
                .elif ((datatype.Name == "boolean" ) AND (comma == 1))
                        ${attribute.Name} : BOOLEAN,
                .elif ((datatype.Name == "boolean" ) AND (comma == 0))
                        ${attribute.Name} : BOOLEAN
                .//elif ((datatype.Name == "state<State_Model>" ) AND (comma == 1))
                        .//${attribute.Name} : ST_${o_obj.Name},
                .//elif ((datatype.Name == "state<State_Model>" ) AND (comma == 0))
                        .//${attribute.Name} : ST_${o_obj.Name}
                .elif ((datatype.Name == "inst_ref<Timer>" ) AND (comma == 1))
                        ${attribute.Name} : ${attribute.Name}_type,
                .elif ((datatype.Name == "inst_ref<Timer>" ) AND (comma == 0))
                        ${attribute.Name} : ${attribute.Name}_type
                .elif ((comma == 1) AND (datatype.Name != "state<State_Model>" ))
                        ${attribute.Name} : ${datatype.Name},
                .elif ((comma == 0) AND (datatype.Name != "state<State_Model>" ))
                        ${attribute.Name} : ${datatype.Name}
                .end if
                .assign count = count + 1
        .end for
#];
```

```
.end if
.end for


.print "Generating SAL Modules"
.select many o_ob from instances of O_OBJ
.for each o_obj in o_ob
.if (o_obj.Name != "Test")
MOD_${o_obj.Name} : MODULE =
BEGIN
.print "Generating Module Global Section"
%% Global Section
        GLOBAL ${o_obj.Name}: REC_${o_obj.Name}
        GLOBAL EVT: EVT_${c_cs.Name}
        GLOBAL ${o_obj.Name}_State: ST_${o_obj.Name}
        .select many o_ob2 from instances of O_OBJ
        .for each o_obj2 in o_ob2
                .if (o_obj.Name != o_obj2.Name)
                        .if (o_obj2.Name != "Test")
        INPUT ${o_obj2.Name}: REC_${o_obj2.Name}
                        .end if
                .end if
        .end for

.print "Generating SAL Initialization Section"
INITIALIZATION

.select many o_attr related by o_obj->O_ATTR[R102]
        .assign objCount = cardinality o_attr
```

```
        .assign count = 1
        .assign classInit = ""
        .assign initState = ""
        .for each attribute in o_attr
        .select one datatype related by attribute->S_DT[R114]
        .if (datatype.Name == "state<State_Model>" )
                .assign initState = attribute.DefaultValue
        .else
                .if (objCount == count)
                        .assign classInit = classInit + " ${attribute.Name} :=
${attribute.DefaultValue} #); "
                .else
                        .assign classInit = classInit + " ${attribute.Name} :=
${attribute.DefaultValue}, "
                .end if
        .end if
        .assign  count = count + 1
        .end for
        ${o_obj.Name}_State = ST_${initState};
        ${o_obj.Name} =  (# ${classInit}

.print "Generating SAL Transitions"
TRANSITION
[
.assign firstEntry = 0
.select any sm_instance related by o_obj->SM_ISM[R518]
.if ( not_empty sm_instance )
   .select one sm_sm related by sm_instance->SM_SM[R517]
        .select many states related by sm_sm->SM_STATE[R501]
```

```
.for each state in states
        %% stateName = ST_${state.Name}
        .select many MatrixEntrys related by state->SM_SEME[R503]
        .select many salStat from instances of ACT_SR
        .if (not_empty salStat)
            .assign nehadStat = cardinality salStat
                all state count = ${nehadStat}

        .else
          .//no statement found at all xxssss
        .end if
        .for each MatrixEntry in MatrixEntrys
          .select one event related by MatrixEntry->SM_SEVT[R503]->SM_EVT[R525]
                    .select one newState related by MatrixEntry->SM_NSTXN[R504]
                    .if (not_empty newState)
                    %% event = EVT_${event.Mning}
                    .select one tranSition related by newState->SM_TXN[R507]
                    .select one destStateX related by tranSition->SM_STATE[R506]
                    .select any block1 related by state->SM_MOAH[R511]->SM_AH[R513]->SM_ACT[R514]->ACT_SAB[R691]->ACT_ACT[R698]->ACT_BLK[R666]
                    .select many block2 related by state->SM_MOAH[R511]->SM_AH[R513]->SM_ACT[R514]->ACT_SAB[R691]->ACT_ACT[R698]->ACT_BLK[R601]
```

```
                    .select any  block3  related by state->SM_MOAH[R511]-
>SM_AH[R513]->SM_ACT[R514]->ACT_SAB[R691]->ACT_ACT[R698]-
>ACT_BLK[R650]
                    .select many block4 related by state->SM_MOAH[R511]-
>SM_AH[R513]->SM_ACT[R514]->ACT_SAB[R691]->ACT_ACT[R698]-
>ACT_BLK[R612]
                    .select any block5 related by state->SM_MOAH[R511]-
>SM_AH[R513]->SM_ACT[R514]->ACT_SAB[R691]->ACT_ACT[R698]-
>ACT_BLK[R699]
                    .assign blkCount2 = cardinality block2
                    .assign blkCount4 = cardinality block4
                    .if (not_empty block1)
                      .//block 1 is not empty
                    .end if
                    .if (not_empty block3)
                      .//block 3 is not empty
                    .end if
                    .if (not_empty block5)
                      .//block 5 is not empty
                    .end if
                    .if (firstEntry == 0)
                    .assign firstEntry = 1
                    .else
                    []
                    .end if
                    ( ${o_obj.Name}_State = ST_${state.Name} ) AND (EVT
= EVT_${event.Mning}) -->
                              ${o_obj.Name}_State' = ST_${destStateX.Name};
```

```
            .end if
                .end for
            .end for
.end if
]
END;
.end if

.end for
.print "Generating System Module"
.assign output = ""
.select many o_ob from instances of O_OBJ
.assign objCount = cardinality o_ob
.assign count = 1
.for each o_obj in o_ob
.assign count = count + 1
.if (o_obj.Name != "Test")
 .assign output = output + "MOD"
 .assign output = output + "_${o_obj.Name}"
 .if (count < objCount)
 .assign output = output + " [] "
 .end if
.end if
.end for

%% System Module:
system: MODULE = ${output};
END
```

```
.print "Generating THEOREMS"
%%Generate Theroems
.select many o_ob from instances of O_OBJ
.assign objCount = cardinality o_ob
.assign count = 1
.for each o_obj in o_ob
.assign count = count + 1
.if (o_obj.Name != "Test")
.select any sm_instance related by o_obj->SM_ISM[R518]
.if ( not_empty sm_instance )
   .select one sm_sm related by sm_instance->SM_SM[R517]
        .select many states related by sm_sm->SM_STATE[R501]
        .assign count=0
        .for each state in states
                .assign count = count +1
                .select one action related by state->SM_MOAH[R511]->SM_AH[R513]-
>SM_ACT[R514]
                .if (not_empty action)
                .if (action.Descrip != "")
%% stateName = ST_${state.Name}
Safe_${o_obj.Name}_${count}: THEOREM system |- G(${action.Descrip} AND
${o_obj.Name}_State = ST_${state.Name})
                .end if
                .end if
                .//Safe_${o_obj.Name}_${count}: THEOREM system |-
G(${o_obj.Name}_State)
        .end for
.end if
.end if
```

```
.end for

.//select any sm_instance related by o_obj->SM_ISM[R518]
.//if ( not_empty sm_instance )
   .//select one sm_sm related by sm_instance->SM_SM[R517]
        .//select many states related by sm_sm->SM_STATE[R501]
        .//for each state in states
                .//%% stateName = ST_${state.Name}
                .//select one action related by state->SM_MOAH[R511]->SM_AH[R513]-
>SM_ACT[R514]
                .//if
        .//end for
.//end if
.emit to file "${c_cs.Name}.sal"
```