Automated Architecture-Based Verification of Safety-Critical Systems
Abstract

Safety-critical systems require high quality and dependability levels, where system correctness and safety are major features to avoid any severe outcome. Time and cost are also important challenges that are imposed during the development process. Describing the behavior of a system in a high level provides a realistic vision and anticipation of the system. This presents a valuable opportunity for verifying the system before wasting the intended resources to develop the system. Architecture Description Languages (ADLs) provide the ability to comprise and represent the system level details of components, interactions and configuration. Architecture Analysis and Design Language (AADL) as a family member of ADLs proved its effectiveness in designing software intensive systems. In this report, we present a case study to validate “An Architecture-Based Verification Technique for AADL Specifications”. The technique involves a combination of model checking and model-based testing approaches adapted to an architectural perspective. The objectives of the verification process are 1) to ensure completeness and consistency of an AADL specification, and 2) to ensure conformance of an implementation with respect to its AADL specification. The technique has only been applied to small examples, and the goal of this thesis work is to validate it against a safety-critical system developed by a major vehicle manufacturer. Validation of the technique begins by investigating the system and specifying it in AADL. The defined verification criteria are subsequently applied to the AADL specification which drives the verification process. The case study presents interesting results while performing the model checking (the completeness and consistency checking). Conformance testing, on the other hand, could not be performed on the implemented system but is an interesting topic for future work.
Acknowledgement

I would like to thank Andreas Johnsen for his extensive support, patience, and guidance during the entire period of this thesis. His suggestions and contributions have helped me a lot to complete the objectives of this thesis. Moreover, I would like to thank my examiner Kristina Lundqvist for her encouragement, which was an enthusiasm source for me. Special thanks go to the family of the school of Innovation, Design and Engineering at Mälardalen University. I could feel that I am a member of your family and this increased my self-confidence.

Finally, I would like to thank Mattias Nyberg for providing me the required documents and information needed to finish this thesis.

Thank you all.

Omar Jaradat
Contents

1. Introduction ............................................................................................................................................ 5
  1.1 Background ........................................................................................................................................ 5
  1.1 Problem Statement and Purpose ........................................................................................................ 7
  1.2 Hypotheses ........................................................................................................................................ 8
  1.3 Candidate System .............................................................................................................................. 8
  1. Related Work ..................................................................................................................................... 9
  1.1 Software Engineering and Formal Specifications ............................................................................ 9
  1.2 Architecting and Modeling Automotive Embedded Systems ....................................................... 9
  1.3 Establishing Formal Regulatory Requirements for Safety-Critical Software Certification ...... 10
  1.4 MITRE’s Architecture Quality Assessment ....................................................................................... 11
  1.5 Relationship on Path Coverage Criteria for Software Architecture Testing ............................... 11
  1.6 A Software Architecture-based Testing Technique ........................................................................ 12
  1.7 Formalization and Validation of Safety-Critical Requirements .................................................... 12
  1.8 System Dependability Evaluation using AADL ................................................................................ 13
  1.9 Modeling Airborne Mission Systems using the Architecture Analysis and Design Language .. 14
  3. Theoretical Background ...................................................................................................................... 16
  3.1 Glance for Software Testing and Verification .................................................................................. 16
  3.1.1 Formal Methods ............................................................................................................................. 17
  3.1.2 Formal Verification ........................................................................................................................ 18
  3.2 Safety-Critical Systems ..................................................................................................................... 20
  3.2.1 Architecture-Based Development ............................................................................................... 20
  3.2.2 Architecture-Based Testing .......................................................................................................... 22
  3.3 Model Checkers ................................................................................................................................ 24
  3.3.1 UPPAAL Overview ....................................................................................................................... 25
  3.4 Architecture Analysis and Design Language (AADL) .................................................................. 28
  3.4.1 AADL language Abstractions ...................................................................................................... 29
  3.4.2 Component Interactions ................................................................................................................. 32
  3.5 An Architecture-Based Verification Technique for AADL Specifications .................................. 40
  3.5.1 The Verification Technique Steps ............................................................................................... 40
  3.5.2 AADL Verification Criteria ......................................................................................................... 42
  3.5.2.1 Verification Objectives ............................................................................................................. 42
  3.5.2.2 Verification Criteria ................................................................................................................ 43
  3.5.3 AADL to UPPAAL Transformation ............................................................................................. 47
4. **Validation of the Architecture-Based Verification technique for AADL Specifications** ............. 49

4.1 *Fuel Level Estimation System Overview* .................................................................................. 49

4.2 *Fuel level Estimation System Analysis* ..................................................................................... 50

4.3 *AADL Model of the Fuel Level System* ..................................................................................... 51

4.4 *Applying Verification Technique to the Fuel Level Estimation System* ................................. 57

4.4.1 *Applying the Verification Criteria: Step 1* .......................................................................... 58

4.4.2 *AADL Model Transformation to UPPAAL: Step 2* ............................................................... 61

4.4.3 *Mapping the Verification Sequences in the UPPAAL Model and Perform the Model checking: Step 3 & 4* ......................................................................................... 66

5. **Discussion** ................................................................................................................................ 75

6. **Summary and Conclusion** ......................................................................................................... 78

7. **Future Work** ................................................................................................................................. 80

8. **References** ................................................................................................................................ 81

*Appendix A* .................................................................................................................................... 84

*Appendix B* .................................................................................................................................... 85
1. Introduction

1.1 Background

Safety-critical systems impose difficult demands in systems engineering. Safety as a system property is required to be assured to avoid any severe outcome. In addition, high quality and high dependability must be achieved within a limited budget and schedule. Nevertheless, software projects in common spend at least 50 percent of their development resources on rework [35]. Detecting and resolving faults in the later phases are more expensive and it is one of the reasons that increase the development cost and time. Thus, the ability to detect faults in the early stages to prevent any increase on the project budget and time is critical. From this brief premise, many efforts have begun and persevered extensively to reduce the cost and the time of the system, and to achieve the optimal possible quality.

The majority of the efforts made in this context focused on the development processes, since they define all the tasks required for building and maintaining the systems. Additionally, the development process indicates when to verify the system during the development. System verification can be performed at the end or along with the system development. For instance, each phase of the development process is followed by a verification activity, but the major verification is performed in the latest phase, where the majority of testing activities are conducted, as shown in figure 1.

![Waterfall Model](image)

*Figure 1: Waterfall Model*

Architecture-based development is a software engineering approach, which helps to manage the implementation complexity by providing design level models and guidelines for composing
systems in terms of components [36]. An architecture-based development process consists of six steps, namely: eliciting the architectural requirements, design the architecture, document the architecture, analyze the architecture, realize the architecture, and maintain the architecture [27], as shown in Figure 2. These steps show that the system architecture is modeled and analyzed before implementing (realizing) the system.

![Figure 2: Steps of the Architecture-Based Development Process [27]](image)

Architecture Description Languages (ADLs) have been developed to describe system architectures, and principal architecture design decisions. ADLs are used in the context of designing software and/or hardware architectures. An ADL can comprise and represent the system level details of components, interactions and configuration. The Society of Automotive Engineers (SAE) has released the Architecture Analysis and Design Language (AADL) in 2004. This language is effective for model-based analysis, with the ability to specify complex real-time embedded systems. [28]

Architecture design is a very critical phase in safety critical-systems development. In this phase, the architecture model represents the first design decisions. These design decisions define the functional properties and the quality attributes of the system. Moreover, the architecture model is very crucial since the rest of development process will highly depend on it. Thus an architecture model is used as a blueprint among system’s stakeholders, and it can be updated iteratively until closure on a design is reached [27]. Evaluating the final design decision is vital, because any incorrect structural, functional, or nonfunctional properties will generate number of problems in the upcoming phases. These failures are considered very expensive to be fixed and resolved. Moreover, to preserve the valuable efforts and the benefits of the verified architecture model, the system implementation must be developed according to the architecture.
specifications. To cope with the previous challenges of safety-critical systems, which are reducing the cost and the time, a development is vested in two additional challenges: 1) evaluating the architecture specifications, 2) testing the conformance of an implementation with respect to its architecture specification.

This report will present a well-defined architecture-based verification technique for AADL specifications. The technique has been innovated by a group of researchers at Mälardalen University in Sweden to automate the verification process of safety-critical systems. The technique is based on formal constructs enabling automation of the verification activities, where challenge 1) and 2) listed above are tackled, by adapting model checking and model-based testing approaches to an architectural perspective. This technique employs the properties and their relations that can be described by AADL, and then extracts verification objectives with corresponding verification criteria from AADL properties. These verification criteria generate verification sequences, where these sequences are used for simulating the model checking of the system architectural, and used later to generate test cases to perform model-based testing. Moreover, the idea of this technique is to evaluate the integration of components at both the specification-level using UPPAAL as a model checker, and the implementation level using the generated test cases. [1]

1.1 Problem Statement and Purpose

The introduced verification technique in the previous subsection is not yet validated. The purpose of this thesis is to validate the practical applicability of the verification technique. Through this work we apply the architecture-based verification technique for Architecture Analysis and Design Language (AADL) specifications to an architectural design for a candidate system. During the application of the technique, we are going to record difficulties, invalidations, or pitfalls that may encounter the application. Moreover, the application of the technique is expected to reveal the feasibility of evaluating the architecture consistency and completeness. The feasibility is measured by a direct comparison with the requirements of the automotive safety standard ISO 26262. More specifically, we consider the safety standard ISO 26262 requirements for verifying the software architecture design and check how the application of the verification technique can contribute to fulfill these requirements.
1.2 Hypotheses

Strictly speaking, the hypothesis that we take into account in our work is that: The architecture-based verification technique for AADL specifications is feasible because it describes the purpose, goals and the process unambiguously.

1.3 Candidate System

The chosen system for this case study is the fuel level estimation system. This is a real safety-critical system, which was developed by a major vehicle manufacture. This system estimates the volume of fuel in a heavy road vehicle’s tank and presents this information to the driver through a dashboard mounted fuel gauge. Additionally, the system must warn the driver when this volume falls below a predefined threshold. This system is considered safety critical because its failure could lead to loss of control of the vehicle.

The system is a subsystem within the vehicle. Therefore, the hardware and the software dependencies of this system, as well as all shared functions and tasks need to be understood to make sure that it can be modeled separately.
1. Related Work

The work in this thesis falls into the area of architecture-based verification of safety-critical systems. In this section, we summarize some related works.

1.1 Software Engineering and Formal Specifications

Sommerville [2] describes how the importance of formal specification in some systems is higher than other systems. The main reason of this disparity is the cost. For critical systems development, “The high cost of failure in these systems means that companies are willing to accept the high introductory costs of formal methods to ensure that their software is as dependable as possible” [2].

The book shows the advantages of using the formal specifications in terms of the high capability to reduce the requirement errors as it forces a detailed analysis of the requirements. This will reduce the amount of rework, which will reduce the system validation cost. A system can be described algebraically, in terms of operations and their relationships. A system can also described by a model-based approach, where it is built using mathematical constructs, and the system operations are defined by how they modify the system state.

The Author ended the description of the formal specification, by saying: “It takes several years to refine a formal specification language, so most formal specification research is now based on these languages and is not concerned with inventing new notations” [2].

1.2 Architecting and Modeling Automotive Embedded Systems

Larses [3] shows how to use the architectures and the models as keys to enhance the cost and the dependability of automotive embedded systems. The thesis represents two methods that are utilizing mathematical tools to analyze and synthesize the architecture. The thesis represents case studies in which these methods were applied to embedded system architectures. The methods are then combined to produce a complete architecture engineering process.

The Design Structure Matrix (DSM) is a tool that contains a set of scripts developed for MATLAB. It is used to support the analysis and the synthesis of the architecture by relying on the matrix
that represents the relations between the architecture objects. The tool combines the objects into clusters where the positive relations within the cluster are maximized and the relations between the clusters are minimized. The idea is that strong and positive relations preferably should be kept within the module, whereas weak or negative relations should be kept between the modules. The Keyfigures is another tool used to analyze and compare architecture designs. The analysis relied on Excel workbook file, where the analysis blocks are the core of the analysis tool and the relations between them provided in a Relations matrix. The Connections and Interface sheets provide data on the implementation of the relation. The thesis shows that "The supporting mathematical methods clarified design decisions and made them more explicit". The thesis concludes that: "With an extended modeling of embedded control systems the architecting will become more engineering and less an art, increasing the usefulness of supporting mathematical methods".

1.3 Establishing Formal Regulatory Requirements for Safety-Critical Software Certification

Paper [4] gives an overview of Formal Methods (FMs), and how these methods are used in system specification and verification. Additionally, the paper proposes a new approach to use the formal methods. The approach is called “formalization of the regulatory requirements for software of safety-critical control systems”. The main notion of the approach is to formalize the regulatory requirements for safety-critical system. The approach relies on two main things: 1) the generic nature that most safety-critical systems share, and 2) the fact that formal regulatory requirements are the basis for certification or licensing processes.

The idea of formalizing the regulatory requirements is to avoid any ambiguous specification that may come from informal definitions. In general, the regulatory body consists of two main tasks, establishing the regulatory requirements for the system, and assess this regulatory. But the introduced use of Formal Methods was developed to perform the first task more efficiently.

Formal regulatory requirements are not applicable directly on the software only, but it can be applied to the process of software development as well. Therefore, the assessment of a software testing process is one of the most important stages of regulatory assessments, where testing criteria can be used as regulatory requirements. The paper demonstrates two examples of using Z notation, where the first example shows how this notation is used to formalize the requirements of protection against unauthorized access, and the second example shows how this notation is used to formalize the requirements of protection against common mode failures. Regulatory of the system requirements has proved that it can be practically used for aiding regulatory assessment. In addition, "the proposed schemas are constructive requirements that are able to determine how the system requirements can be checked and confirmed".
1.4 MITRE’s Architecture Quality Assessment

Paper [5] proposes an Architecture Quality Assessment (AQA) repeatable technique. The AQA is narrowly focused on architecture related artefacts or deliverables. This technique can be used in different ways: 1) evaluate the architecture, 2) review the architecture development, 3) assess the architecture, or 4) compare two or more alternate architectures in a consistent fashion.

The AQA uses the terminology of architecture meta-model, and here are some representative definitions: the architecture is the highest-level concept of a system in its environment; the architecture is documented as an architectural description. An architectural element may be a component, connection or constraint. A component depicts a major element of view. A connection depicts a relationship between components. A constraint depicts a law which a component or connection must obey. The methodology used in AQA simply depends on three layers: Quality area, factors, and measures to organize the assessment. There are six identified quality areas: understandability, feasibility, openness, maintainability, evolvability, and client satisfaction. Based on the quality factors that are a set of questions, the assessment of the architecture is conducted through five concrete steps: 1) Perform Needs Analysis (when one is not available), 2) Gather relevant documents and other artifacts related to the architecture. 3) Evaluate documentation against measures and score results. 4) Interpret results and identify architecture related risks. 5) Document results for client. The assessment result for each quality factor will be represented by one of six values (IDEAL, GOOD, MARGINAL, UNACCEPTABLE, INCOMPLETE, and NON-APPLICABLE).

Finally, there are three main assessment products to demonstrate the quality status of the system architecture, namely, 1) Executive Summary, 2) Detailed Evaluation and Interpretation of results, and 3) A set of Open Issues and Questions).

1.5 Relationship on Path Coverage Criteria for Software Architecture Testing

Lun and Chi [6] present a software architecture testing technique based on Linear Temporal Logic (LTL). The technique defines coverage criteria based on path coverage for software architecture testing. Moreover, the technique expresses the software architecture by three main parts, components, connections and the interaction between these components and connections. Software Architecture SA can be expressed as finite set of components, where component is a data unit or a computation unit, which is composed of the component interface and components internal computation model, i.e. $SA = \{\text{Comp}, \text{Conn}, \text{Cons}\}$. A connector is composed of connector interfaces that connect these connectors with the components, in addition to the connector’s internal computation.
Formalizing the architecture properties and evaluating these properties over paths (i.e., over linear sequences of states using linear Temporal Logic LTL, and interface connectivity graph ICG) determine the testing paths. Subsequently, three architecture testing coverage criteria were defined, so that, a set of test paths is adequate if all the identified architecture relations have been fully exercised.

1.6 A Software Architecture-based Testing Technique

Zhenyi Jin in his dissertation [7] has exploits how software ADLs can offer a significant opportunity for testing, and that is because these ADLs can describe how the software should behave in high-level view. Jin depicted how the architectural problems can be addressed through the architectural relations that define the behavior and connectivity among software components via connectors. The dissertation demonstrates an architecture-based testing technique to test the relation among the architecture component. The technique depends on the architecture relations that are based on possible bindings: data transfer, control transfer, and execution ordering rules. Defining these relations will allow the derivation of testing requirements and criteria based on the architecture relation coverage. Accordingly, six architecture relations have been defined: Component (Connector) Internal Transfer Relation, Component (Connector) Internal Sequencing Relation, Component (Connector) Internal Relation, Component Connector Relation, Direct Component Relation, and Indirect Component Relation. However, before defining test criteria, it is important to determine what needs to be covered in the architectural level. Therefore, the dissertation proceeded with defining the testing paths. Testing path has been identified as a path between two interfaces, either component interfaces or connector interfaces. Nine architecture-based testing paths have been defined: Component internal transfer path, Component internal ordering rules, Connector internal transfer path, Connector internal ordering rules, Component to connector, Connector to component, Direct component to component path, Indirect component to component path and Connected components path. Afterword, five Software architecture-based test criteria are defined: Individual component interface coverage, Individual connector interface coverage, All direct component-to-component coverage, All indirect component-to-component coverage, and All connected components coverage.

The result of Jin's technique shows that it is effective to find faults at the architecture level.

1.7 Formalization and Validation of Safety-Critical Requirements

Cimatti et al. [8] propose a methodology and a series of techniques for the formalization and validation of high-level requirements for safety critical applications. This methodology depends
on three main steps: 1) informal analysis of the high level requirements, where the requirements are categorized based on their characteristics and then structured based on their dependencies. 2) Formalizing each category by specifying the corresponding formal counterpart. This step requires tracing between the informal textual documented requirements and the formal categorized requirements to validate them. 3) Validating the requirements, it is designed to improve the quality of the requirements and increase the confidence that the categorized requirement fragment and its corresponding formalized counterpart meet the design intent. The validation step includes three checks, checking logical consistency, scenario compatibility, and property entailment.

To specify the safety-critical systems requirements, the four researchers adopt a fragment of first order temporal logic, which allows specifying constraints on objects, their relationships, and their attributes. Consequently, they use a class diagram to define the classes of objects specified by the requirements, their relationships and their attributes. This class diagram defines the signature of the first order temporal logic. The temporal structure of the logic encompasses the classical linear time temporal operators combined with regular expressions.

The validation depends on three steps: 1) fix a number of objects per class where it is possible to reduce the formula to an equisatisfiable formula free of quantifiers and functional symbols. 2) Translating the results of quantifier free hybrid formula into an equisatisfiable formula in the classical temporal logic over discrete traces. 3) Compiling the resulting formula into a Fair Transition System. The researchers have validated their methodology against The European Train Control System (ETCS). Domain experts that are not involved in the consortium validated the results of the project. The evaluation was carried out in form of a workshop, followed by hands-on training courses.

A plan to investigate the application of automated techniques for Natural Language Processing is considered as a future work.

1.8 System Dependability Evaluation using AADL

Paper [9] presents an approach for system dependability modeling and evaluation, where the AADL dependability models are built on the architecture skeleton by using features of the AADL Error Model Annex, a draft annex to the AADL standard. Moreover, the GSPNs (Generalized Stochastic Petri Nets) are used in the introduced example as well. GSPN is considered as a modeling framework, which provides a systematic integrated representation of the timed, and the logical behavior [10]. The introduced approach contains four steps as follows: The first step starts by modeling the system architecture using AADL, where the modeling focuses on the architectural components and the operational modes of these components. The second step is to model the behavior of the system with the existence of faults through AADL error models.
associated to components of the AADL architecture model. Hence, the set of error models associated to the architectural components represents the AADL system error model. The difference between the first and the second steps is that the first step models the behavior of each component, as if it was isolated from its environment, whereas the second step and incrementally models the dependencies among these components. “The final model represents the behavior of each component not only in presence of its own faults and repair events, but also in its environment”. The third step constructs a global analytical dependability model by extracting particular information from the AADL model. This model is generated in the form of a GSPN by applying model transformation rules. This step can be incremental as well, because it is possible to enrich the global analytical model based on the number of iterations in the previous step. The forth step processes the outputted GSPN model in the third step to obtain a dependability measure. This step is based on classical GSPN processing algorithms (used by existing tools), which include syntactic and semantic validation of the model and evaluation of quantitative dependability measures.

1.9 Modeling Airborne Mission Systems using the Architecture Analysis and Design Language

Paper [11] describes an initial research used to model and analyze Airborne Mission Systems (AMS). The AADL was used in the work because it can fit for model-based development and analysis of real-time embedded system. The paper gives an overview of AADL and its advantages in providing different ways of expressing the system model. Moreover, two case studies are presented to understand AADL and to employ its capabilities. The objective of the first case study was to assess the benefits that can be gained from AADL, by developing an AADL model related to the air domain (T-REX model helicopter), where the pilot can control the helicopter by a controller. This model includes major components that are electrical, electronic or mechanical. The goal of the second case study (S-70B-2 SEAHAWK HELICOPTER) was to investigate how to specifically model a subset of the mission system hardware and software of a Royal Australian Navy S-70B-2 Seahawk helicopter. The model was based on the documentation and source code (written on Ada) of the Seahawk’s Display Graphics Unit (DGU).

The paper also represents a high level graphical AADL model by using the OSATE tool for both case studies. The outputted result of the first case study is summarized as: “Developing the T-REX AADL model proved a relatively straightforward exercise. It involved obtaining information about the composition and operation of the T-REX and then translating this into a model”. Furthermore, the outputted result of the second case study stated that modeling the Seahawk DGU brought forward different issues to the ones encountered with the T-REX”.

14
This is because Seahawk DGU had a large amount of ad hoc information with no particular structure. This means that it is necessary to manually trace the Ada code and try to check the included tasks and how they operated. Accordingly, “it was proved difficult to extract the required information in order to construct an accurate model and populate it with parameters that depict the actual operation”. The better ways to obtain this information is to either devote addition resources to the problem, or to obtain more information from the manufacturer.

In [12] as an earlier research, Dodd was able to conduct analysis on his petri net model and obtain simulated processor utilization values that were comparable to the real system. Dodd’s advantage in this case was that he focused on one aspect, but the challenge in the research reported in [11] is to use AADL to model the entire system, where the complexity is very large. However, abstracting parts of the system, where the fidelity of the model decreases its utility for analysis diminishes, can reduce this complexity.
3. Theoretical Background

3.1 Glance for Software Testing and Verification

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are by definition, not smart enough to debug it.”

Brian Kernighan.

Software testing is very important to assure software quality. However, the required testing efforts vary from a system to another and it is, typically, subject to the system functional requirements. For instance, the required efforts to test a hotel reservation system are not similar to the required efforts to test an autopilot system. The required level of quality will determine the objectives of the testing activities, should we test to see if the software works? Find errors? Check consistency? Etc. Accomplishment of test objective for software will give enough confidence that it works correctly.

Software testing has two major categories, dynamic testing and static testing. In dynamic testing, the system or part of it must be compiled and executed, based on the designed test cases, then the expected outputs will be compared with the actual generated output by the running system. Dynamic testing contains three main testing categories, functional testing, structural testing and random testing. Functional testing is used to test functions of a system based on its requirements. Most often, functional testing is called “black box testing” since the knowledge about internal details of the system is not needed. Black box testing is designed to check a particular specification of the system requirements.

For structural testing, also known as “white box testing”, it requires knowledge about internal details of the system, where the test cases are designed by the information from the internal structure of the system. Random testing comes from the idea that a complete System verification is not possible. Therefore, random testing can increase the confidence that the system is correct. Simply, it captures a random set of test cases and executes them as an attempt to detect faults that could not be detected before.

Static testing can contain both automated and manual techniques. The static testing focuses on two aspects of the system, namely, consistency and correctness. Using static testing enables software testers to analyze the requirements consistency by investigating the correctness of the system properties (i.e., correct requirement specifications, correct data types, correct parameter matching between subprograms or procedures, etc.).
Based on the development phases, testing can be performed at six different levels: unit, module, integration, subsystem, system and acceptance testing [14]. A testing technique is needed in order to execute any of these six levels of testing. For each phase of testing it is necessary to understand when a system has been sufficiently tested. This is determined by the adequacy criteria of the testing technique. Testers must understand these adequacy criteria to decide if the system has been tested sufficiently for a particular testing criterion, where testing criteria are set of rules that represent the system requirements as a set of test cases [15].

Generally speaking, the testing efforts are spent along with the software development process. Different types of testing may be used to reach an acceptable level of confidence that the system can operate as intended. In this thesis work, we show how formal architecture verification can contribute to increase our confidence level that the system architecture is complete and consistent enough. More specifically, we show the feasibility of using a Model Checking technique to check the architectural model properties.

There are many architecture verification techniques that depend on the formal architecture specification. The most prominent techniques are those whose take advantages of using formal architecture specification, which can be modeled and defined by one of the ADLs, where system architecture analysis is supported. [16]

3.1.1 Formal Methods

Although there is no explicit definition of the Formal Methods FMs in the literature of software engineering, but we adopt the FMs definition that has been implicitly known between software engineers. “The term formal methods is used to refer to any activities that rely on mathematical representations of software including formal system specification, specification analysis and proof, transformational development, and program verification” [20]. According to this definition, FMs propose a mechanism or a technique to specify, develop, or test complex systems by using mathematical notations. It is worth noting that FMs have a clear limitation in scalability. [20]

The growing of the FMs usage and popularity in the last decade could give an illustration about its importance and efficiency to represent and describe different software specifications. In particular, those systems with high complexity levels. For example, FMs have been employed by some researchers at NASA as an attempt to catch the possible errors in their safety-critical systems. NASA’s researchers could indeed detect severe errors and problems in the requirements specifications of these systems. Rockwell aviation, as another example, has used the FM techniques to catch the errors in the flight guidance system [18].
In 1994, Lutz and Ampo [19] have investigated the effectiveness of FMs in critical system requirements analysis, after applying these methods on critical spacecraft software, and they reported good effects as a result. This leads to say that FMs have been used in practical systems, and according to many studies, it showed a big success in safety-critical systems development.

However, it is worth asking: **When and how can FMs be beneficial?**

It is worthwhile for system developers to detect and resolve system faults as early as possible. Faults that are left out uncovered may propagate as severe failures. Resolving those failures will be more costly and time consuming if they are detected in the later stages of the software development lifecycle. Standish Group report [21] has proved that the problems of the system requirements are responsible for half of all software project failures. Avoiding problems originating from the requirements is therefore critical. The report showed that FMs could effectively be used for requirements analysis, requirements specification and high level design stages. In other words, FMs may decrease the failures that are caused by system requirements and eventually, decrease the overall cost of the system.

FMs can be exploited in two stages. The first stage is the formal specification, where the formal specification means that the system specifications are represented mathematically. The second stage is the formal verification, where mathematical analyses may be used to check the completeness and consistency of the system requirements.

Generally speaking, the use of FMs is costly. However, it has increased in the area of critical systems where safety, reliability or securities is highly considered. The reason behind that is because failures in critical systems may cause harmful consequences and more costly that using FMs. Therefore, decision makers may tradeoff between the cost of using FMs and the cost of system failures.

**3.1.2 Formal Verification**

Realizing the benefits of the formal specifications has paved the way to think of different ways to uncover system faults as early as possible. Formal verification is one of these ways. Formal verification can be exploited in different phases of the system development process. It can be used in the requirement phase to detect the faults in the requirement specifications. Formal verification is also beneficial in the design phase to assess the correctness, adequacy and consistency of the architecture design. Formal verification can be used in the implementation phase to evaluate the source code implements the required functionality [22].
The main objective of the formal verification is to mathematically prove if the system implementation or design conforms to the requirements specifications of that system. Requirement specifications are critical factors through the system development since the quality of the design and the implementation is derived from the quality of the requirement specifications [23].

Unlike informal verification, formal verification depends on formal specifications of the system requirements or system design. An easy way to formalize these specifications is to use a design tool that can automatically formalize the design into one of formal languages. However, if the used design tool lacks the formal semantic, then a transformation from informal textual requirements or design specifications to a formal description is required.

Model checking is a formal verification method. It requires binding the finite state presentation (as shown in figure 1) with a formal method to detect any sequences of states that violate the requirements correctness [24]. Thus, the method can be used to check the architectural model properties. Exploiting this method can be useful since it has the ability to detect and tell why a property is not satisfied. Additionally, it can detect the deadlocks, wrong interactions or inconsistency among the model states that can cause model crashes, or harm the correctness, completeness or concurrency properties of the model [34].

Model checking can improve the verification quality in terms of saving the time and reducing the cost, as well as, it provides a high coverage that can reach one hundred percent of the possible cases.

Moreover, there is another approach for formal verification, it is called proof-theoretical approach and it is based on theorem proving methods [22]. Generally, this approach considers
the declarative statements that are derived from the requirement specifications. Based on the system nature, these statements mathematically specify the behaviors and the relationships between the properties of real-world or system objects. Then the specification will be the theory and it is assumed to be true according to laws of the real-world application. For instance, the statement says “Every employee has an identification number” and “Every identification number must be owned by an employee”, the previous statements can be valid for some organizations, but in the same time no one can prove that is logically true. Therefore, formal verification is responsible for ensuring that system properties and constraints are logical statements derived from non-logical axioms.

3.2 Safety-Critical Systems

Safety-critical system is a system where any disorder or defect of this system can dramatically cause a loss of life, severe damages, or damage of the environment [25]. There are many examples of safety-critical systems, such as, autopilot systems, embedded systems in the medical devices, nuclear reactor cooling systems, etc. The main challenge of building safety-critical systems is to deliver a product that is free from any unreasonable risk.

It is well known that detecting faults in the later stages of software development process will significantly increase the development cost and the time. Development of safety-critical systems focuses on delivering a fault free product, where the expected cost of building such systems is high. Therefore, verification techniques that can detect faults as early as possible are highly required.

3.2.1 Architecture-Based Development

The rationale of the importance of the early verification efforts in the development process is that system faults in those phases will, most likely, propagate in the subsequent phases. Therefore, the verification efforts require adequate awareness that the requirements must meet several key attributes to ensure quality. System requirements must be verified to be correct, consistent, and complete enough. This requires an efficient way to analyze and test these requirements.

The required efforts for verification in safety-critical systems may be unlike other systems in terms of rigor (i.e., obligations to safety standards, required evidence, compelling evidence, etc.). This may influence the way of representing, analyzing and testing the requirements. Software architecture design is not only a phase of the system development process but it is also a discipline that all the subsequent phases depend on. Thus, it is important to explain what the architecture should represent and how is going to be verified.
Generally speaking, the system architecture is a set of components (nodes), connected to each other by connectors (relations), and these components and connectors can be annotated with properties. Basically, properties are described as a subset of the architecture configurations. These properties represent related information used to distinguish between different types of components and connectors. More importantly, properties represent information about the system quality attributes, such as, performance, security, safety, reliability, etc. [27]. Figure 2 shows an example of the architecture design, including the representation of components, connectors and properties.

![Architecture Design Example Using AADL](image)

*Figure 2: Part of an architecture design example using AADL*

In particular, the architecture design of safety-critical systems describes the runtime environment. This means that the architecture represents the hardware platform and the software descriptions. Moreover, it shows how both are linked together in one consistent model.

The use of Architecture Description Languages (ADLs) provides the capability to represent and analyze system architecture designs. The description of the architecture can be expressed textually “and/or” graphically depending on the used description language. There are many Architecture Description Languages (ADLs) used to model the architecture, such as, AADL (SAE) Architecture Analysis & Design Language, Rapide (Stanford), MetaH (Honeywell), etc. ADLs are not similar to traditional programming languages (as shown in figure 3). They are mainly used to describe architecture design decisions in the context of software and hardware.

There exist ADLs solely for hardware, solely for software and for both. For software architectures, ADLs capture the components behavioral specifications in addition to the
interactions between these components. But for hardware architecture, ADLs capture the hardware components and their connections, as well as, the behavior of the hardware components architecture [29].

![Diagram](image.png)

*Figure 3: ADLs VS Non-ADLs [29]*

Regardless of the ADLs type or name, all of them must share the ability to describe system software and/or hardware architecture. However, they can vary in terms of the extra beneficial features that they may contain.

### 3.2.2 Architecture-Based Testing

Architecture-based testing means that the testing should be guided by the definition of the architecture. Modeling the system architecture using an ADL may yield to a well-described architecture. Consequently, when the system implementation becomes available, the implementation can be tested with respect to the architecture. However, it is important to understand what architecture aspects that an ADL can represent.

In 2001, Jin and Offutt [30] defined general criteria for testing at the architectural level. They started by defining six architecture relations between the architecture components. These relations have been defined according to the possible bindings of data transfer (data flow), control transfer (control flow), and execution sequencing rules between components interfaces. Jin and Offutt use these relations to define six architecture testing paths which are used to define the architecture level testing criteria representing the testing requirements, as shown in figure 4.
The underlying goal of the technique is to ensure that all architectural relations are covered when performing the verification. This is achieved through the defined criteria that require all paths to be tested.
Jin and Offutt intend to adapt these general criteria to derive specific test criteria for specific ADLs. Applying the five testing criteria above to an ADL specification must yield to derive the test paths. Generated paths are tested and simulated to detect any inconsistency among the architectural relations. These test paths can later be used for testing the system that is implemented to satisfy the architecture description. [31]. The foregoing is giving the essence of an architecture-based testing method, where it is clear that the architecture guides the implementation phase.

### 3.3 Model Checkers

The model checking is an automatic formal verification technique. It can detect any existence of deadlocks, wrong interactions or inconsistency among the model states that can cause model crashes, or harm the correctness, completeness and concurrency properties of the model [32]. There are many advantages that can be obtained by using a model check technique. The most important advantage is that the model check can detect and tell why a property is not satisfied.

Timing properties in safety-critical systems can be represented as timed finite state automata using the Computation Tree Logic (CTL) approach. These properties can be evaluated over paths (as shown in figure 6). The model checker algorithm comes to verify every timed automaton (path), by checking whether all executions of the model satisfy the test queries. These test queries (also called formulae) are expressions by which can be inserted in the model checker to check a specific property.

Model checkers are able to verify four main properties, namely, reachability, safety, liveness and concurrency. This will be covered in more details in chapter 4.
Utilizing a model checker to simulate and test the paths of systems architectures is beneficial since it can evaluate some specifications in those architectures.

### 3.3.1 UPPAAL Overview

UPPAAL is a joint venture toolbox developed for verification of real-time systems, by UPPsala University in Sweden and AALborg University in Denmark. UPPAAL is a model checker and based on the theory of timed automata. It can be used to model a system as a network of timed automata, and by using the UPPAAL verifier, testers can insert computation tree logic (CTL) queries to check properties of the model. A timed automaton is a finite state machine driven by clock variables that progress synchronously and evaluates to real numbers. The model is extended with bounded discrete variables that are used as any programming language variables in term of the accessibility [33]. A state defines the locations of all automata, the clock constraints, and the values of the discrete variables. A system enters a new state if each automaton triggers an edge separately or synchronously with another automaton.

Figure 7 shows two timed automata. The first automaton on the left side represents the three locations of the lamp (Off, Dim and bright). User pressing action will be checked by (press?) which synchronizes the pressing action, clock (c) is the time constrains which will check the period between the user presses to decide the appropriate location. On the right side of the same figure, we can see the (User) automaton which has only one location, and since the interaction between the user and the lamp can be occurred by pressing, so pressing the button can change the system state which requires synchronizing (press?) with the lamp automaton [33].

The illustrated example (in figure 7) describes the behavior of a lamp with respect to a button. When the user presses the lamp button, then the lamp will start dimming for five seconds and
then it will turn off. But if the user presses the lamp button once and follows the first press with one more press within five seconds, then the lamp will be bright, and it will remain bright unless the user press the lamp button to turn it off.

Figure 7: UPPAAL touch lamp model.

The UPPAAL query language is used to verify the model requirement specifications. The result of each query could be valid or invalid. Since UPPAAL uses a simplified version of a Computation Tree Logic (CTL), the query language includes the path formulae that trace the paths of the model. In addition, the state formulae that describe individual states can be evaluated to be true whenever it is valid (regardless of the model behavior). Path formulae can be classified into three properties reachability, safety and liveness. [33]

Reachability properties are considered the simplest form of properties, a reachability formula tests if a state formula "q" can be valid in a reachable state, or in other words, it checks if there are states satisfying a specific q should be reachable according to a specific given path, and we can express this formula by \( E <> q \), where q is the state formula. Safety properties represent the fact “something bad will never happen”. In UPPAAL these properties are formulated positively, “something good is always true”. A safety formula checks whether a propositions "q" invariantly is true in all reachable states with the path formulae with the path formulae (A [ ] q; where A = all reachable states), whereas (E [ ] q; where E = some path globally q holds) says that there should exist a maximal path such that (q) is always true. Liveness Properties represents the fact “something will eventually happen”, in the previous Lamp example, when pressing the lamp’s button then eventually the lamp should turn on. In other words, Liveness Properties checks if a proposition eventually is satisfied, and in UPPAAL it can be said that A [ ] q, where q has eventually satisfied, e.g. whenever a button is pressed. A <> q, then eventually the lamp will be turned on q \( \rightarrow \) p (where p is the satisfied state, \( \rightarrow \) denotes “leads to”). A deadlock state means
that “there are no outgoing action transitions neither from the state itself or any of its delay successors”. In UPPAAL deadlock state formula can only be used with reachability and invariently path formulae. [33] [31]

UPPAAL is free download tool, and it can be downloaded at (www.uppaal.org for more details).


3.4 Architecture Analysis and Design Language (AADL)

As described in (3.2.2), the outcomes of applying test criteria to a particular ADL will be very dependent on the characteristics of that ADL and how it describes the architecture model. Therefore, any approach relies on the architecture-based testing technique must take into account the various capabilities of the Architecture Description Languages (ADLs). Since the work in this thesis is dependent on the Architecture Analysis and Design Language (AADL) as one of the ADLs family, it is very important to provide a background about this language.

In specialty for model-based analysis and specification of complex real-time embedded systems, the aerospace standard “AS5506” has been released by the Society of Automotive Engineers (SAE) in 2004, under the name of Architecture Analysis & Design Language (AADL). [28]

As a member of Architecture Description Languages, AADL is used to model the software and the execution platform (hardware) architectures. Taking into account the performance-critical characteristics through an extendable notations, tool framework, and defined semantics. AADL architecture models are described as a component connector structure. This means that a model of a system describes the components, components’ interfaces and the connectors among these components. Moreover, AADL can describe various characteristics (timing, runtime, synchronization, etc.) of the architecture elements by associating them to specific property annotations.

AADL provides the capabilities to describe the message passing, event passing, synchronized access to shared components, thread scheduling protocols, timing requirements, remote procedure calls, etc. Dynamic reconfiguration of the runtime environment can be specified using mode and mode transitions. [28]

This language has a valuable advantage since it can be used even with limited architectural details. It allows incomplete description for either systems already in use or new systems. Furthermore, the description of the architecture can be expressed both graphically and textually. There are many used tools for the AADL. The most known tools are OSATE, Stood and Ocarina. In this thesis we use OSATE v1.5.8.

For more information please see (http://www.aadl.info/aadl/currentsite/tool/osate-down.html)
3.4.1 AADL language Abstractions

An AADL Component is defined by a unique identity name, and it is created by type and implementation declarations. A component must have a type declaration, where the descriptions of external interfaces are defined. The component implementation describes the internal structure i.e. subcomponents, subprogram call sequences, modes, flow implementations, and internal properties. Figure 8 illustrates an example to a process component (component categories will be discussed later in more details). The left side of the figure shows the type declaration, while the right side shows the implementation declarations.

![Diagram of CalculateSpeed component](image)

**Figure 8**: Graphical and textual descriptions for a Process component type and implementation declarations

Upon the example in figure 8, we can notice that the declaration of the component type is defining the external characteristics (interfaces) of the process component within the component features. The declaration of the component implementation shows that the process component has a thread as a subcomponent, and the execution time property for this subcomponent. It also shows how the connections are established between the process component and its subcomponent (thread). Basically, there are three main and separated component categories used in AADL, namely, application software, execution platform and composite. Both the application software and the execution platform consist of a set of component types. The composite category, however, consists of the system as a component type.
- **Application Software**

Application software consists of the following five components based on [28]:

1. **Thread**: An active component that can be executed concurrently. Threads can be organized into thread groups. Threads are concurrent schedulable units of sequential execution through source code. Thread can be specified with property annotations describing different characteristics, such as, execution time, deadline, dispatch protocol etc.

2. **Thread Group**: is a component abstraction for logically organizing threads, data, and thread group components within a process. Since the AADL allows the thread group component to be associated with properties, thread group can unify the properties for the threads that it includes.

3. **Process**: is a protected address space whose boundaries are enforced at runtime. The address space contains executable binary images directly associated with 1) the process, 2) subcomponents of the process, 3) server subprograms, and 4) data that are referenced by external components. A process must contain at least one thread.

4. **Data**: is a component represents a static data and data types within a system. Data component declarations are used to represent application data types, the substructure of data types via data subcomponents within data implementation declarations, and data instances.

5. **Subprogram**: subprogram component abstraction represents sequentially executable source text that operates on data or provides server functions to components that call it. A subprogram and its parameter signature are declared through component declarations. Subprogram cannot be a subcomponent for any other component.

- **Execution Platform**

Execution platform consists of the following three components based on [28]:

6. **Processor**: is an abstraction of hardware and associated software that is responsible for scheduling and executing threads. Processors can execute threads that are declared in application software systems or threads that reside in components accessible from those processors.
7. **Memory**: Memory component represents storage components for data and executable code (i.e., subprograms, data, and processes are bound to memory components). Memory components include randomly accessible physical storage (e.g., RAM, ROM) or complex permanent storage such as disks or reflective memory. Memory can include a memory as a subcomponent.

8. **Device**: Device abstractions represent entities that interface with the external environment of a system. Those devices often have complex behaviors. They may have internal processors, memory, or software that are not necessary modeled. Device components may require driver software that is executed on an external processor. A device's external driver software may be considered part of the processor execution overhead, or it may be treated as an explicitly declared thread with its own execution properties.

9. **Bus**: A bus represents hardware and associated communication protocols that enable interactions among other execution platform components (i.e., memory, processor, and device).

    - **Composite**

Composite abstraction contains only one component type, which is System component.

10. **System**: System represents a combination of software and/or hardware components and it is used to instantiate the system model.

The graphical representations of these components are shown in figure 9.

*Figure 9: General Graphical symbol of AADL type components.*
3.4.2 Component Interactions

Components interact through interfaces defined within component type declarations, under the features sub clause for a particular component. This makes component type responsible to identify the interaction interfaces, while component implementation responsible to draw the paths of this interaction.

- Component Types

The component type can specify the interactions through [28]:

- Data ports for unqueued state data.
- Event data ports for queued message data.
- Event ports for asynchronous events.
- Synchronous subprogram calls.
- Explicit access to data components.

We can sum up what can be contained under features sub clause by four main elements, ports, access, subprogram, and parameter.

1. Ports

Ports are communication interfaces for the directional exchange of data, event or both data and event, between the architecture components, and their graphical representations are shown in figure 10:

![Figure 10: ports Graphical representations. [28]](image-url)
Ports are directional and can be specified to be in or out or in out.

- **Data ports**: are connections between data ports are either (immediate) or (delayed). Figure 8 in page 29 shows an example of data port declaration and in the example there are declaration for three data ports, two in ports and one out port.

- **Event ports**: are interfaces for the communication of events by subprograms, threads, processors, or devices that may be queued.

- **Data Event ports**: are interfaces for message transmission. These interfaces enable the queuing of the data associated with an event.

### 2. Component access (Data Access & Bus Access)

Unlike the ports, component access is declared as **provides** or **requires**, according to the design where components may provide or require the connections.

In AADL, a component access is directed and can either be a **data access** or **bus access**. This requires a data component or bus component to be part of the connection respectively. For example figure 11 below shows a two bus accesses, **GetValues** represents access connection between CeleronCPU and SpeedBus, as well as, **SendValues** represents and access connection between the Sensor and the Speedbus component.

![Figure 11: Graphical and textual representations for bus access.](image)
3. Subprogram calls

Subprograms are not declared as subcomponents; instead they are called through call declarations within a thread or subprogram implementation. The subprogram that is called must be declared through a subprogram type declaration and possibly a subprogram implementation declaration [28]. Figure 12 shows the subprogram declaration.

4. Parameter

Represents call and return data values that are transmitted into and out of a subprogram component. These exchanges by value are declared as typed data features in the type declaration of a subprogram, similar to data port declarations [28] Figure 12 shows parameter representation within a subprogram.

![Figure 12: Graphical and textual representations for subprogram and its parameters.](image)

In addition to the features sub clause, a component type can be modeled with other elements, as shown in figure 13. For extends sub clause it is used as any object oriented programming language. For instance, when a class A extends class B in Java, A automatically has all variables and methods defined in class B. Similarly, in AADL when component X extends component Y, then component X automatically has all the descendants included in Y, i.e. features, it can be said that component type Y forms the core for component type X. It is possible to modify the extended descendants but this can be done within the component implementation declaration (see next section).
Within the flows sub clause, it is possible to define specifications of logical flows through the component from incoming interaction points to outgoing interaction points. These logical flows can be used to specify End-to-End flows without the need to expose or have any available implementation detail of the component. Flows can trace data, control, or mixed flow by connecting event and data ports. Flows are directional, so specifying a complete flow requires a declaration in component type and component implementation. Flows are defined within the component type, where the start of the flow, the way to the end, and the end of the flows, determined by declaring: source: a feature of a component, sink: a feature of a component, and flow path: a flow through a component from one feature to another. Flows within component implementation will be discussed in the next section.

Properties: specifies properties of the component that apply to all instances of this component unless overwritten in implementations or extensions. [28]

- **Component Implementations**

Component implementation is responsible on the internal structure in addition to draw the paths of the interactions (connections). That is why it is significant to know how the AADL component implementation is declared especially in term of interaction. Figure 14 below shows the semantic declaration view and including all the possible sub clauses for component implementations.

The declaration begins by identifying the component name. Since the declaration implements a component type, the implementation name should begin with the name of the component type followed by the implementation name, separated by a dot. Furthermore, it is possible to model multiple implementations of a component type and this will allow the designer to reuse the
same features of the component type. Multiple implementations can be done as the same way of identification but by giving a different implementation name after the dot.

![Figure 14: AADL component Implementation summary.](image)

**Extends** sub clause has the same definition as the component. But it is important to mention here that designers can modify or overwrite the extended descendants, by using **refines type** sub clause. So it is used to add property values to the features of its corresponding type.

**Subcomponents** are specified within the declaration of the component implementation. The example shown in (figure 15) represents that the process component has two threads called “Computation.Imp and GetResult.Imp” as subcomponents. Moreover, the declaration can assign a property for these subcomponents as shown.

![Figure 15: Example of subcomponents and connections](image)

As explained earlier, the interaction elements are used to enable the components to interact within the architecture. In a component type declaration, these elements (interfaces) can be defined to form the infrastructure of the components communication. Consequently, the
component implementation links these elements and builds the bridges between the architecture components. In AADL the connections between the predefined interfaces elements are explicitly declared under the "connection" sub clause in the component implementation. Connections have an identifier, descriptor, source and destination. According to the interfaces elements, there are three types of connections: port connections, component access connections and parameter connections. Figure 16 gives an example of the port connection in AADL.

![Diagram](https://via.placeholder.com/150)

*Figure 16: Graphical and textual Example of data port connections.*

The process implementation "CalculatedSpeed.Imp" consists of one thread as a subcomponent called "Computation.Imp". The component types for these are not shown in example. The connection declarations defined three data port connections as the following: "Time" from in data port "GetTime" in process "CalculatedSpeed.Imp" to in data port "Time" in process "Computation.Imp" thread, and there is a data port connection named "Distance" from in data port "GetDistance" in process "CalculatedSpeed.Imp" to in data port "Distance" in thread "Computation.Imp". Finally, there is out data port connection "Speed" from thread "Computation.Imp" to out data port "GetSpeed" in process "CalculatedSpeed.Imp".

An access connection (as introduced in Figure 11 in the previous section) represents the (requires, provides) relation. The same Figure shows the type declaration for the access connection, while figure 17 implemented the connection as shown.
The system implementation “PC.Imp” contains three subcomponents processor “CeleronCPU.Imp”, bus “SpeedBus.Imp”, and Device “Sensor.Imp”. Moreover, there are two bus access connections, “Link1” and “Link2”, as shown the bus is a side of both connections because it is the provider, while processor “CeleronCPU.Imp” and Device “Sensor.Imp” require the access.

Parameter connections represent flow of data into and out of subprograms and data flow through a sequence of subprogram calls. Parameter connections can be declared between subprogram parameters or between a data port and a subprogram parameter. [31]

Flows as introduced in the previous section enable the detailed description and analysis of an abstract information path through a system. But within a component implementation, flow declarations define the details of flow paths through a component, and end-to-end flows within the component. Figure 18 shows a flow example.

Figure 17: Graphical and textual Example of access connections.

Figure 18: Example of flow in component implementation. [28]
**Modes** represent alternative operational modes that may manifest themselves as alternate configurations of subcomponents. In other words, modes are used to represent the system in term of states. Of course, the transition from a state to another is bound with specific configurations. And logically, there must be two modes at least and one mode must be defined as the default mode (initial mode). Otherwise, the system will stick in one mode (state). Each mode must have an explicit defined configuration and an event that cause transition to another mode.

![Figure 19: Example of modes. [28]](image-url)
3.5 An Architecture-Based Verification Technique for AADL Specifications

Johnsen et al. [1] have proposed a verification technique for AADL specifications. The technique relies on the model checking to evaluate the completeness and consistency of the system architecture. The technique relies also on the model-based testing approach test the system implementation with respect to the architecture specification.

The architecture specification is critical since it represents the earliest design decisions that are made to represent the system structure, functional properties and quality attributes. Disorder within or absence of some established design decisions at the design phase might lead to faulty structure, functional, and non-functional properties. Detecting these faults at the design phase will avoid their propagation in the later phases. However, if the incorrect properties could not be detected until later phases, then the correction is considered more costly and time consuming. This verification technique aims to evaluate the architecture specification (model checking) to detect possible faults and inconsistencies before proceeding with the next phases. In addition to, test the conformance of the system implementation with respect to its architecture specification (model-based testing). However, The technique objective should not be confused with the thesis objective as described earlier in this report.

ADLs specify system architecture by describing the properties of the system and the relations among these properties. Nevertheless, ADLs do not have unified properties in term of specifying the architecture. However, if the properties of an ADL have been addressed, then the extraction of specific verification objectives with corresponding verification criteria is possible. AADL as one of these ADLs is adopted by this verification technique, where applying the defined verification criteria on AADL specification will generate the sequences of the integration verification. These sequences will be transformed into UPPAAL model to perform the model checking, and will be used to create concrete test cases for the model-based testing.

AADL lacks the implemented semantics are are needed to automate the simulation of its specification. This problem has been solved by formally specifying semantics of AADL constructs using the UPPAAL language (model checker).

3.5.1 The Verification Technique Steps

In section (3.1.2 Formal verification), I have introduced the formality of the requirement specifications and the importance of formalizing them to pave the way for the formal verification
process. Using AADL to formalize the requirements and their specifications is possible, but what not possible is to prove that the formalized requirements really conform the informal requirements. This technique will not consider this conformation but rather, it will employ the AADL specifications to prove or deny the inconsistencies, ambiguities, and completeness of the system requirements.

The proposed verification technique consists of five steps in order to apply it to any AADL model. Consequently, steps of this technique start mainly from formalizing the AADL semantics, as shown in figure 20.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{flowchart.png}
\caption{Flowchart of the verification technique [1]}
\end{figure}

Steps are [1]:

1. Use the transformation rules to transform an AADL specification to an UPPAAL model upon which automated formal verification can be performed.
2. Apply the architecture-based verification criteria (section 3.5.2) to the AADL specification. They define the test selection, i.e., what samples of the specification to evaluate and how they are extracted, and the coverage requirement, i.e., how many samples to evaluate. The samples generated from the criteria are sequences of component integrations in terms of control flows and data flows.

3. Sequences from the previous step are transformed, in this step, to the corresponding automata paths in the UPPAAL model through a structural mapping between them.

4. The outcomes from step 3 i.e. a set of automata paths are required in this forth step to be fully simulated in UPPAAL, by using temporal logics, in order to satisfy the criteria. The verdict from the simulations reveals the consistency and completeness of the AADL specification, where a correction of the specification should be made if it is shown inconsistent or incomplete.

5. The paths are later used in this step to generate test cases to the implementation, to test the conformance of the implementation with respect to the architecture specification. Test paths are transformed to concrete test cases through a mapping between the architecture specification and its implementation (the technique assumes identical namespaces between the AADL specification and the system).

### 3.5.2 AADL Verification Criteria

Jin and Offutt defined general architecture-based testing criteria formulated to be applicable to any ADL (as presented in section 3.2.2). Their definition was based on the data flows and control flows properties. Unlikely, AADL connectors have no interfaces but alternatively they rely on the interfaces of the components that they connect. This makes Jin and Offutt defined criteria not applicable on AADL. Therefore, the verification technique for AADL specifications [1] used the general verification criteria to define its own verification criteria specific to AADL, based on the possible bindings of both data flow and control flow properties.

#### 3.5.2.1 Verification Objectives

The main objective of the verification is to ensure the consistency and the completeness of the system architecture, in addition to the conformance of an implementation with its AADL specification. The assurance is based on the evaluation of interactions among components.
As described in section (3.4.3), there are four possible connections that specify the architectural control flows and data flows of an AADL specification, *port connections, component access connections, subprogram call connections, and parameter connections.*

Architectural data flows are the relations between definitions of data elements from the source component to the destination component, while the control flows are the different execution orders of architectural elements. These straightforward flows can be influenced in terms of completeness and consistency by mode state machines, behavioral annex (BA) [34] refinements, and properties association. Therefore, the verification criteria objectives must uncover any conflict that may threaten the completeness and the consistency of the architectural control flows and data flows in general. Verification criteria objective can ensure the consistency and the completeness by analyzing three main areas [1]:

1. **Control flow Reachability:**
   Every executable element within an execution order in the architecture should be able to reach the subsequent element to be executed in the order. *"The subsequent element should be reached without conflicting with properties of the execution order".*

2. **Data flow Reachability:**
   Every data element should be able to reach its target component, where the data is used, from its source component, where the data is defined. *"The target component should be reached without conflicting properties of the data flow".*

3. **Concurrency among flows:**
   Analysis of single interactions of data or control is not enough since there are implicit relations between them that may cause deadlocks in the system. *"The relations between the flows should not prevent control flow reachability or data flow reachability, and where the system should be free from deadlocks".*

### 3.5.2.2 Verification Criteria

Basically, this verification technique defines the verification criteria form the flows. Since the atomic bindings (AADL relations) generate the flows, so these relations are employed to define integration verification sequences of both control and data flows. Then from these sequences the architecture-based verification criteria are defined, as shown in figure 21.
AADL Specification is represented as a 5-tuple:

$$\text{AADLSPEC} = \langle N, I, C, BAC, PAC \rangle$$

Where:

- $N$ is the set of Components $= \{n_1, n_2, \ldots, n_n\}$.
- $I$ is the set of component interfaces $= \{n_x.i \mid n_x.i \text{ is a port, data access, subprogram or parameter interface of } n_x \text{ and } n_x \in N\}$.
- $C$ is the set of Connections $= \{c(s, d) \mid c(s, d) \text{ is a port, a data access, a subprogram call or a parameter connections that connect the source interface } s \in I \text{ to the destination interface } d \in I\}$.
- $BAC$ is the set of BA Connections $= \{bac(s, d) \mid bac(s, d) \text{ is an automaton path and the initial location in the path or a transition from the initial location is labeled with } s \in I \text{ and the last location in the path or a transition to the last location is labeled with } d \in I\}$.
- $PAC$ is the set of Property Associated Constructs $= \{pac \mid pac \in I \cup C \cup BAC \text{ and is constrained by at least one associated property}\}$.

Based on this representation of an AADL specification, the defined relations are:

1. **Connection Transfer Relation**: defines the data or control transfer that is generated between two interfaces connected through a connection.

   $CTR$ is the set of Connection Transfer Relations where $CTR \subseteq I \times I$

   Such that $<n_x.i_1, n_y.i_2> \in CTR$ if $c(n_x.i_1, n_y.i_2) \in C$.

2. **Connection Property Relation**: defines the constrained data or control transfer that is generated between two interfaces connected through a connection.

   $CPR$ is the set of Connection Property Relations where $CPR \subseteq I \times I$

   Such that $<n_x.i_1, n_y.i_2> \in CPR$ if $c(n_x.i_1, n_y.i_2) \in C \cup BAC$ and $n_x.i_1$, or $n_y.i_2$ or $c(n_x.i_1, n_y.i_2) \in PAC$.  

---

Figure 21: coverage criteria.
3. **Component Internal Relation:** defines the (possibly constrained) data or control transfer that is generated between two interfaces of a component that are connected through a connection or a BA.

CIR is the set of Component Internal Relations where $\text{CIR} \subseteq I \times I$

Such that $<n_1.i_1, n_1.i_2> \in \text{CIR}$ iff $<n_1.i_1, n_1.i_2> \in \text{CTR} \cup \text{CPR}$ or $<n_1.i_1, n_1.i_2> \in \text{BAC}$.

Relation $b$ in figure 22 demonstrates this definition, where the connection $C_2$ internally connects $(N_2.\text{Int}_1 \rightarrow N_2.\text{Int}_2)$ through the BA.

4. **Direct Component-to-Component Relation:** defines the (possibly constrained) data or control transfer that is generated between two components that are directly connected through a connection.

$\text{DCCR}$ is the set of Direct Component to Component Relations where

$\text{DCCR} \subseteq I \times I$ such that

$<n_1.i_1, n_2.i_2> \in \text{DCCR}$ iff $<n_1.i_1, n_2.i_2> \in \text{CTR} \cup \text{CPR}$

Relation $a$ in figure 22 represents this definition, where connection $C_1$ is connecting $N_1$ and $N_2$ directly as: $(N_1.\text{Int}_1 \rightarrow N_2.\text{Int}_1)$. $N_2$ and $N_3$ are also connected directly through $C_3$ where $c$ relation represents the same definition as: $(N_2.\text{Int}_2 \rightarrow N_3.\text{Int}_1)$.

5. **Indirect Component-to-Component Relation:** defines the (possibly constrained) data or control transfer that is generated between two components that are indirectly connected through one or several component(s). The relation is recursive in order to cover any number of interconnected components. The base case is:

$\text{ICCR}$ is the set of Indirect Component to Component Relations where

$\text{ICCR} \subseteq I \times I \times I^*$ such that $<n_1.i_1, n_3.i_4, t> \in \text{ICCR}$ iff $<n_1.i_1, n_2.i_2> \in \text{DCCR}$

$<n_2.i_2, n_2.i_3> \in \text{CIR}$ and $<n_2.i_3, n_3.i_4> \in \text{DCCR}$ and $t = <<n_1.i_1, n_2.i_2>, <n_2.i_2, n_2.i_3>, <n_2.i_3, n_3.i_4>>$

The recursive definition is:

$\text{ICCR}$ is the set of Indirect Component to Component Relations where

$\text{ICCR} \subseteq I \times I \times I^*$ such that $<n_1.i_1, n_3.i_4, t> \in \text{ICCR}$ iff $<n_1.i_1, n_2.i_2> \in \text{DCCR}$ and $<n_2.i_2, n_2.i_3> \in \text{CIR} <n_2.i_3, n_3.i_4, t'> \in \text{ICCR}$ and $t' = <<n_1.i_1, n_2.i_2>, <n_2.i_2, n_2.i_3>, <t>>$
As shown in figure 22 $N_1$ is connected indirectly with $N_3$ through $C_1$ and $C_2$, where relation $d$ represents this definition as: $(N_1, \text{Int}_1 \rightarrow N_3, \text{Int}_1)$.

![Graphical definitions of the Architecture Relations](image)

Figure 22: Graphical definitions of the Architecture Relations

From these AADL relations three verification sequences are derived, which are paths of the architecture specification:

1. **Component Internal Transfer Path**: If there exists a $<n_1.i_1, n_1.i_2> \in CIR$, there exists a path from $n_1.i_1, n_1.i_2$. The path is constrained if $<n_1.i_1, n_1.i_2> \in CPR$.

2. **Direct Component-to-Component Path**: If there exists $<n_1.i_1, n_2.i_2> \in DCCR$, there exists a path from $n_1.i_1, n_2.i_2$. The path is constrained if $<n_1.i_1, n_2.i_2> \in CPR$.

3. **Indirect Component-to-Component Path**: If there exists a $<n_1.i_1, n_x.i_y, t> \in ICCR$, there exist a path from $n_1.i_1, n_x.i_y, t$ via $t$. The path is constrained if any pair in $t \in CPR$.

"The AADL specification is consistent if each path is free from contradictory behavior, that is, each path does not contradict Control flow reachability, Data flow reachability and Concurrency among flows. The AADL specification is complete if each path not yielding an end-to-end flow (typically a sensor-to-actuator flow) is subsumed in another path". [1]

Consequently, from the previous verification sequences, three coverage criteria have been defined to ensure the adequacy of the requirement simulations or the test cases. Within the following coverage criteria definitions, "S" is either a set of simulations of an AADL specification or a set of test cases for an implementation implemented in correspondence to an AADL specification:
1. **Component Internal Coverage**: requires that $S$ covers all Component Internal Transfer Paths.

2. **Direct Component-to-Component Coverage**: requires that $S$ covers all Direct Component to Component Paths.

3. **Indirect Component-to-Component Coverage**: requires that $S$ covers all Indirect Component to Component Paths.

![Diagram showing relations, test paths, and test criteria](image)

**Figure 23: Architecture-Based verification criteria specific to AADL.**

### 3.5.3 AADL to UPPAAL Transformation

This automated formal verification technique requires formal semantics to allow formal verification, and implemented semantics to allow the automation. But as mentioned before, AADL does not provide implemented semantics. The only way to solve this problem is to formalize the AADL constructs. Formalization is done by transforming these constructs to timed automata constructs in the UPPAAL modeling language which has formally specified and implemented semantics. The transformation process is controlled by a set of rules.
AADL execution model (instance) will be transformed to timed automata model. This AADL execution model has different aspects of a runtime environment, where the runtime execution behavior can be configured by the associated properties. For instance, the runtime environment can be configured to be synchronous interactions or asynchronous interactions based on (Synchronized_Component) property, and it can behave as a recovery execution or nominal execution. By default, AADL standard specifies the runtime environment behaviors as synchronous interactions and preemptive scheduling. The default model consists of periodic threads communicating through their data ports. Restrictedly, the transformation rules of this verification technique considers default specification of the model instance, so the rules are set to consider a synchronous with preemptive scheduling instance model.
4. Validation of the Architecture-Based Verification technique for AADL Specifications

As previously discussed, the main goal of this thesis is to validate the introduced architecture-based verification technique for AADL specifications presented in the previous chapter. The validation procedure starts by investigating one of the real-world safety-critical systems to model and analyze the system by using the AADL standard. Subsequently, the AADL model is transformed to an UPPAAL model based on the transformation rules. The desired validation has been decided to be against the “Fuel Level Estimation System”, which was developed by a major vehicle manufacturer.

4.1 Fuel Level Estimation System Overview

The main functionality of the Fuel Level Estimation System is to estimate the fuel level in the vehicle tanks and present this level by a fuel meter located on the dashboard. Additionally, the fuel level estimation system must warn the vehicle driver if the fuel level drops below a predefined level. This requires an accurate measurement by avoiding the rapid changes that may occur when the vehicle is not driven on a flat surface (i.e., steep hills and tough terrain).

If the system fails to indicate the real fuel level in the tank, more specifically, if the real fuel level is low while the fuel level meter presents a higher level, then the driver will be unable to determine the correct time to refuel. Consequently, there will be a possibility by which the engine will run out of the fuel required to keep it in operation. The sudden shutting down of the engine may force the truck to stop in the middle of the road and this may yield to a hazardous consequences. Therefore, this system is considered a safety-critical system.

The estimation process of the fuel level is based on the fuel type (i.e. liquid, gas), and vehicle type (i.e. Bus or truck). Taking into account that a vehicle can contain more than one type of fuel, in this case, the vehicle will have one tank for each type. The diversity of the used fuel type and the vehicle type will play a major role in terms of the used elements (i.e. sensors and actuators) in the system. For instance, the used elements and ECUs (Electronic Control Units) in gas fuel estimation is different than with liquid fuel. Since these elements will be connected to different ECUs, each case has its own design configuration according to the used element types, used ECUs and their connectivity. The work in this thesis is adopting the fuel estimation for trucks with liquid fuel type, and with one fuel tank.
4.2 Fuel level Estimation System Analysis

The fuel level estimation system in trucks with liquid fuel relies on an ECU called COO (Coordinator), which contains many functions (abbreviated as AEs) beside the fuel estimation. COO executes the fuel level estimation and sends the results via CAN bus (Controller Area Network) to another ECU called ICL (Instrument Cluster) to display the current fuel level, and trigger warning lamp if fuel level is lower than a predefined level. Thus, the fuel level estimation and warning are distributed over two ECUs i.e. COO and ICL respectively.

![Figure 24: Fuel Level Estimation for trucks with liquid fuel.](image)

The fuel estimation function (AE201) begins with the analog fuel level sensor as shown in figure 24. The sensor is located in the fuel tank, and directly connected to an A/D converter in the COO.

The COO also has a software layer called Basic Software (B-Software In, B-Software Out), and through this B-Software, COO can store the converted values into RTDB (Real-time Database) which represents a shared memory. The RTDB is a middleware between the B-Software and the COO AEs, and since the RTDB is accessible shared memory, it is the only communication channel between 1) all COO AEs, and 2) between these AEs and B-Software, as shown in figure 25.

(AE201) reads the converted voltage value from the RTDB, and transform it to the correspondent volume in percentage. The result will be used together with the maximum capacity of the tank in liters to determine the current fuel volume (C). A Kalman filter algorithm is used as another step to estimate the fuel level, and here the previous estimated fuel volume will be used with (C), as well as the fuel consumption in the engine to calculate a new level estimation. AE201 writes the new estimated fuel level into the RTDB.
The stored estimated value will be read by the fuel level warning function $AE202$ to evaluate the warning level, and by $B$-$Software$-$Out$ to be forwarded to $ICL$ via the $CAN$ bus. $AE202$ reads the estimated fuel level from the $RTDB$ and compares it with a predefined level to decide if the estimated level is a low fuel level or not. The result is written into $RTDB$. $B$-$Software$-$Out$ reads the level evaluation (low or not low) from the $RTDB$ and forward it to $ICL$ via $CAN$ bus again. $COO$ $AEs$ in addition to $B$-$Software$ layer (In and out) are dispatched periodically every 10 milliseconds.

Finally, $ICL$ gets the values sent by the $B$-$Software$ $Out$ (estimated fuel level and low fuel level, with respect of Fuel level estimation system only) via $CAN$ bus. Thereafter, $ICL$ converts the received values from digital to analog values. $ICL$ is responsible to control the fuel level meter, as well as to trigger the low fuel level warning lamp.

### 4.3 AADL Model of the Fuel Level System

This section will explain the AADL model of the Fuel Level Estimation System, using $OSATE$ (v1.5.8) as a modeling tool. To model the system in AADL it is important to match each system element to the appropriate AADL component. Table 1 shows the main explicit elements of the Fuel Level Estimation System and the correspondent AADL components:

<table>
<thead>
<tr>
<th>Main Element</th>
<th>AADL component</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 COO ECU</td>
<td>System Component</td>
</tr>
<tr>
<td>2 ICL ECU</td>
<td>System Component</td>
</tr>
<tr>
<td>3 Fuel Level Sensor</td>
<td>Device Component</td>
</tr>
<tr>
<td>4 Fuel Level Meter (Actuator)</td>
<td>Device Component</td>
</tr>
<tr>
<td>5 Low Fuel Level Lamp (Actuator)</td>
<td>Device Component</td>
</tr>
</tbody>
</table>
Table 1: Fuel Level Estimation System main elements with the correspondent AADL components

AADL has a good flexibility as it allows development of partial specifications, which especially is beneficial in the early stages where details may be undecided or may not be known yet. But to instantiate a system implementation the model must be fully specified, where all software components are bound to the correspondent hardware components. Therefore a system component is needed to assemble and integrate hardware and software components as a preparation to instantiate the system model. A system component named “Item_FuelLevelModel.imp” is created to assemble all Fuel system components, as shown in figure 26 and 27. We will present this system component in a top-down fashion.”

Figure 26: Graphical AADL representation for the Fuel Level Estimation System high level design
Figure 27: Textual AADL representation for the Fuel Level Estimation System high level design

The ECU COO (as shown in figure 28) has three interfaces and one bus access: SensorValue in data port to get the sensor value, FuelLevel out data event port to send the estimated fuel level value, and LowFuelLevelWarning out data event port to send the fuel low level evaluation. It has a software layer called Basic Software (B-Software In, B-Software Out), and it is modeled as a process subcomponent "BasicSoftware". This process has four interfaces: SensorValue_2 in data port to handle the converted sensor value, EstimatedFuelValue out data event port to send the estimated fuel level value, LevelWarningValue out data event port to send the fuel low level evaluation, and AEs_Connection requires Data Access to enable the accessibility between the shared data and the COO AEs. The BasicSoftware process has three subcomponents: SoftwareIN thread which has two interfaces SensorValue_2 in data port to handle the converted sensor value, and SetSensorData requires Data Access to write the sensor value into the RTDB (Real-time Database). The second subcomponent of the BasicSoftware is SoftwareOUT thread which has three interfaces: GetFinalValues requires Data Access to read the final estimated fuel level and low fuel level values from the RTD, FinalEstimationValue out data event port to send the final fuel estimation value, and FuelLevelWarningValue out data event port to send the final low fuel level warning value. The third subcomponent of the BasicSoftware is the RTDB which represents a shared data component and is considered as a communication channel 1) Among all COO AEs (ex. Fuel level estimation and fuel level warning), and 2) between the COO AEs and B-Software. The RTDB component has only one interface SendReceiveSharedData provides Data Access to allow all COO AEs to read from and write to this data component.
Finally, the fuel level estimation AE201, the fuel level warning AE202, and the rest of AEs are represented as threads contained in separated processes:

- **FuelEstimationCalculation** process: has FuelEstimation thread as a subcomponent with one interface `GetSensorValue_2 requires Data Access` to read the converted stored sensor value, and write the final level estimated value.

- **FuelLevelWarningCalculation** process: has FuelLevelWarning thread as a subcomponent with one interface `GetEstimatedFuelLevel_2 requires Data Access` to read the level estimated value, and write the final fuel low level warning value.

- **COO_Other_AEs**: this process abstracts the AEs included in the COO ECU, but not involved in the fuel level estimations system. It has one thread subcomponent called OtherAEs and one interface requires data access interface `ReadWrite` to read from and write to the RTDB.

---

**Figure 28**: Graphical AADL representation for the COO system model
ICL ECU is less complex since it does not contain any calculations. ICL receives the values in percentage from the COO over CAN, and then displays these values on the fuel gauge. The AADL model of the ICL ECU is quite simple, where the ICL is a system that contains one process TriggerSignals which has four interfaces: WarningLamp in data event port, FuelLevel in data event port, WarningLampTrigger out data port, and FuelLevelDisplay out data port. The TriggerSignals process includes one thread TriggerLampAndMeter with four interfaces. As shown in, TriggerLampAndMeter thread is responsible to send the data required by the warning lamp and the fuel level meter.
The textual representations for the COO threads in addition to the ICL thread are not shown in figure 29 or 31, since they are declared in separated components as explained before. COO and ICL as system components consider the processes as subcomponents, while these processes consider threads as subcomponents. Threads contained in the fuel level estimation system (COO and ICL) are presented in the figure 32.
Figure 32: Full specifications of the threads contained in the Fuel Level Estimation.

The fuel level estimation system has various properties such as timing properties (execution time, deadline), dispatch properties (priority, periodic, aperiodic, speriodic, etc). These properties can be represented in AADL through property annotations associated to each component.

### 4.4 Applying Verification Technique to the Fuel Level Estimation System

As an example of applying the verification technique to the fuel level estimation system, we will focus on the AADL specification of COO system.
### 4.4.1 Applying the Verification Criteria: Step 1

As stated in section (3.5.1), the verification technique requires the verification criteria to be applied to the AADL specification as a first step. These criteria define the test objective, the test selection, and the coverage requirements based on the possible control and data flows in an AADL specification. Flows are represented by interactions between interfaces of AADL components, as explained in section (3.4.2 component interaction). In the COO system, the interactions are specified by the data access connections among the system subcomponents interfaces.

The verification technique has formally specified five types of AADL relations (in section 3.5.2.2). These relations define the possible control and data flows, and are used to generate the control and data flow diagrams from an AADL specification. COO system contains multiple control and data relations. Thus, generating the control and data flow diagram requires identifying the relations in the system.

1. **Connection Transfer Relation**

This relation defines the data or control transfer that is generated between two interfaces connected through a connection. Table 2 represents the connection transfer relations within the COO system. As an example, there is a **Connection Transfer Relation** (data transfer) between SoftwareIN thread and RTDB data component, through the connection (SoftwareIN.SetSensorData → RTDB.SendReceiveSharedData).

<table>
<thead>
<tr>
<th>Connection Name</th>
<th>Feature type</th>
<th>Source interface</th>
<th>Destination interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>FuelLevelSensorValue</td>
<td>data port</td>
<td>COO.SensorValue</td>
<td>BasicSoftware.SensorValue_2</td>
</tr>
<tr>
<td>GetSavedSensorValue</td>
<td>data access</td>
<td>BasicSoftware.AEs_Connection</td>
<td>FuelEstimationCalculation.GetSensorValue</td>
</tr>
<tr>
<td>GetSavedFuelEstimatedLevel</td>
<td>data access</td>
<td>BasicSoftware.AEs_Connection</td>
<td>FuelLevelWaningCalculation.GetEstimatedFuelLevelValue</td>
</tr>
<tr>
<td>ReadWrite</td>
<td>data access</td>
<td>BasicSoftware.AEs_Connection</td>
<td>COO.Other_AEs.MemoryReadWrite</td>
</tr>
<tr>
<td>CalculatedFuelLevel</td>
<td>event data port</td>
<td>BasicSoftware.EstimatedFuelValue</td>
<td>FuelLevel</td>
</tr>
<tr>
<td>CalculatedLevelWarning</td>
<td>event data port</td>
<td>BasicSoftware.LevelWarningValue</td>
<td>LowFuelLevelWarning</td>
</tr>
<tr>
<td>ConvertedSensorValue</td>
<td>data port</td>
<td>BasicSoftware.SensorValue_2</td>
<td>SoftwareIN.SensorValue_3</td>
</tr>
<tr>
<td>SetSensorConvertedValue</td>
<td>data access</td>
<td>RTDB.SendReceiveSharedData</td>
<td>SoftwareIN.SetSensorData</td>
</tr>
<tr>
<td>GetCalculatedValue</td>
<td>data access</td>
<td>RTDB.SendReceiveSharedData</td>
<td>SoftwareOUT.GetFinalValues</td>
</tr>
<tr>
<td>GetSetAEsValues</td>
<td>data access</td>
<td>RTDB.SendReceiveSharedData</td>
<td>AEs_Connection</td>
</tr>
<tr>
<td>SendFinalFuelEstimationValue</td>
<td>event data port</td>
<td>SoftwareOUT.FinalEstimationValue</td>
<td>EstimatedFuelValue</td>
</tr>
<tr>
<td>SendFinalFuelLevelWarningValue</td>
<td>event data port</td>
<td>SoftwareOUT.FuelLevelWarningValue</td>
<td>EstimatedFuelValue</td>
</tr>
<tr>
<td>ReadWriteCon</td>
<td>data access</td>
<td>COO.Other_AEs.MemoryReadWrite</td>
<td>OtherAEs.ReadWrite</td>
</tr>
</tbody>
</table>

*Table 2: COO Connection Transfer Relations*
2. **Connection Property Relation**

A connection property relations is a “line and arrows” relation but with a property which constraints “the line and arrow” to a specific value (e.g. latency) or order (e.g. priority properties constraints the interaction between threads to a specific order). Connection property relation defines the constrained data or control transfer that is generated between two interfaces connected through a connection.

For example, AE_201 thread (as shown in figure 30) has *Compute_Deadline property*. From the time of dispatch, the AE_201 thread must write the estimated fuel level into RTDB within 8 milliseconds. Property relations are not limited to the given deadline property which is only an example from many other examples as shown in table 3.

<table>
<thead>
<tr>
<th>Relation No.</th>
<th>Property</th>
<th>Source Component</th>
<th>Destination Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Compute_Deadline</td>
<td>SoftwareIN</td>
<td>RTDB</td>
</tr>
<tr>
<td>2</td>
<td>Compute_Deadline</td>
<td>FuelEstimation</td>
<td>RTDB</td>
</tr>
<tr>
<td>3</td>
<td>Compute_Deadline</td>
<td>FuelLevelWarning</td>
<td>RTDB</td>
</tr>
<tr>
<td>4</td>
<td>Compute_Deadline</td>
<td>Other_AEs</td>
<td>RTDB</td>
</tr>
<tr>
<td>5</td>
<td>Compute_Deadline</td>
<td>SoftwareOUT</td>
<td>RTDB</td>
</tr>
<tr>
<td>6</td>
<td>Priority</td>
<td>SoftwareIN</td>
<td>FuelEstimation</td>
</tr>
<tr>
<td>7</td>
<td>Priority</td>
<td>FuelEstimation</td>
<td>FuelLevelWarning</td>
</tr>
<tr>
<td>8</td>
<td>Priority</td>
<td>FuelLevelWarning</td>
<td>Other_AEs</td>
</tr>
<tr>
<td>9</td>
<td>Priority</td>
<td>Other_AEs</td>
<td>SoftwareOUT</td>
</tr>
</tbody>
</table>

*Table 3: connection property relations among COO system*

3. **Component Internal Relation**

AADL software components can have internal interactions. For instance, *SoftwareOUT* reads the final results of fuel level estimation, Fuel level warning evaluation, and the other functions of the COO system by *GetFinalValues* interface, and then transfers these values to other components. Consequently, there are two *Component Internal Relations* between (*GetFinalValues* → FinalEstimationValue) and (*GetFinalValues* → FuelLevelWarningValue).

4. **Direct Component to Component Relation**

There are three interfaces of the COO system, and these interfaces are connected *directly* to the “BasicSoftware process”, and the two threads in the BasicSoftware SoftwareIN and SoftwareOUT are connected *directly* to the RTDB. Moreover, the COO system with its subcomponents contains fifteen *direct component to component relations* (as shown figure 28). However, there is no direct thread to thread relation among the COO system.
5. *Indirect Component to Component Relation*

In order to estimate the fuel level in the system, FuelEstimation thread should read the sensor values from the RTDB, where these values cannot be written into RTDB unless they pass through SoftwareIN first, it means that there is a relation between SoftwareIN and FuelEstimation through the RTDB component. Therefore, there is an *indirect component-to-component relation* between SoftwareIN and FuelEstimation threads. Generally, the RTDB component is the mediator between the different threads in the COO, thus the threads are indirectly connected as shown in table 4.

<table>
<thead>
<tr>
<th>Relation No.</th>
<th>Connected through</th>
<th>Source Component</th>
<th>Destination Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RTDB</td>
<td>SoftwareIN</td>
<td>FuelEstimation</td>
</tr>
<tr>
<td>2</td>
<td>RTDB</td>
<td>FuelEstimation</td>
<td>FuelLevelWarning</td>
</tr>
<tr>
<td>3</td>
<td>RTDB</td>
<td>FuelEstimation</td>
<td>SoftwareOUT</td>
</tr>
<tr>
<td>4</td>
<td>RTDB</td>
<td>Other_AEs</td>
<td>SoftwareOUT</td>
</tr>
<tr>
<td>5</td>
<td>RTDB</td>
<td>FuelLevelWarning</td>
<td>SoftwareOUT</td>
</tr>
<tr>
<td>6</td>
<td>RTDB</td>
<td>SoftwareIN</td>
<td>SoftwareOUT</td>
</tr>
</tbody>
</table>

*Table 4: Samples of Indirect component-to-component relations among COO system*

After extracting the relations of the AADL specification, it is possible to generate the control and data flow diagram of COO system as shown in figure 33.

*Figure 33: Directional data and control relations among the threads of COO system.*
Generating the verification sequences begins with extracting the possible paths in the control and data flow diagram. The flows diagram of the COO system (represented in figure 33) will be used to extract these possible paths.

The following paths are three examples of the paths contained in the COO; they will be used for the rest of the verification technique validation steps:

- **Path 2**: SoftwareIN.SetSensorData → FuelEstimation.GetSensorValue_2 → SoftwareOUT.GetFinalValues → SoftwareOUT.FinalEstimationValue
- **Path 3**: SoftwareIN.SetSensorData → OtherAEs.ReadWrite → SoftwareOUT.GetFinalValues

### 4.4.2 AADL Model Transformation to UPPAAL: Step 2

COO system contains four processes (BasicSoftware, FuelEstimationCalculation, FuelLevelWarningCalculation, and COO_Other_AEs). Each process contains one thread, except BasicSoftware, which contains two threads. The features of these five threads are represented in figure 28, which shows the interfaces and the connections of these threads. The threads included in the system model have a periodic dispatch protocol, where all the threads must be executed every 10 Ms, and sequentially based on the priorities of the threads. According to the transformation rules, each thread will be represented as an automaton in UPPAAL in addition to the scheduler automaton. Each thread automaton has a clock (cl) to keep track of the dispatches of the thread. The dispatch edge is synchronized with the scheduler automaton to notify the dispatch based on each thread priority, execution time, and deadline. Figure 34 shows the transformation of SoftwareIN thread, where priority, execution time, and deadline are 1ms, 1ms, and 8ms respectively. The other threads (i.e. AE201_FuelEstimation, AE202_FuelWarning, Other_AEs, and SoftwareOut) have similar form but of course they have different interfaces and values.
The SoftwareIN thread has two interfaces, the in data port SensorValue_3 and the data access interface SetSensorData. These interfaces are transformed into UPPAAL by mapping them to local variables, which are assigned at dispatch by global variables representing the connections of these interfaces (local variables). Threads in the ready location are assigned to be executed by the corresponding processor (that they are bound to), figure 30 shows the processors bound to each thread in the system. After threads dispatch, and synchronously with the scheduler, SoftwareIN thread will alone enter the running edge to be executed first because it has the highest priority. Then the thread with the second priority will follow (i.e. FuelEstimation), and so forth. A thread in the running location that completes its execution transits to the awaiting dispatch location to repeat its life cycle.

An AADL processor, which is responsible for scheduling and executing threads, is transformed to a scheduling automaton as shown in figure 35. Table 5 shows the automaton variables, arrays, channels, etc.
Figure 35: The Scheduler Automaton [1]

<table>
<thead>
<tr>
<th>Scheduler (Lists, Functions &amp; channels)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(int)</strong> ready_queue[x]</td>
<td>Is a sorted queue of currently dispatched threads. The queue is sorted according to a given scheduling policy where the first element in the queue (x=0) is the (identifier of the) thread being processed and where the second element is the next thread to be processed, and so forth.</td>
</tr>
<tr>
<td><strong>(clock)</strong> sch_clocks[x][2]</td>
<td>Is a list of clocks in sets of two, each set referenced by an identifier x of a currently dispatched thread. Each dispatched thread has two clocks, the first (sch_clocks[x][0]) of thread with identifier x is used to keep track of a thread’s execution time, and the second (sch_clocks[x][1]) of thread with identifier x) is used to keep track of a thread’s deadline.</td>
</tr>
<tr>
<td><strong>(int)</strong> sch_info[x][3]</td>
<td>Is a list of threads’ scheduling properties (integers) in sets of three, each set referenced by an identifier x of a currently dispatched thread. Each dispatched thread has three scheduling properties, the first (sch_info[x][0] of thread with identifier x) is the execution time, the second (sch_info[x][1]) of thread with identifier x) is the deadline, and the third (sch_info[x][2]) of thread with identifier x) is the priority. Note that the required properties are related to a given scheduling policy. For example, we consider priorities of threads since we assume a fixed priority scheduler in this particular example.</td>
</tr>
<tr>
<td><strong>(int)</strong> preempt_stack[x][2]</td>
<td>Is a stack of sets of currently preempted threads</td>
</tr>
</tbody>
</table>
(integer identifiers) and the amount time each thread has been preempted. Given a stack of preempted threads, the first set of elements in the stack (preempt_stack[0][0] is the thread identifier and preempt stack[0][1] is the amount of time) corresponds to the thread that first was preempted.

<table>
<thead>
<tr>
<th>(int)nr_preempted</th>
<th>Is the number of currently preempted threads.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(int)threads</td>
<td>Is the number of currently dispatched threads.</td>
</tr>
<tr>
<td>(int)check_preempt</td>
<td>It holds the identity of a thread that is dispatched at the same time as another thread is running. It is used to check if the dispatched thread preempts the running thread.</td>
</tr>
<tr>
<td>(chan)dispatched[(int)x], (chan)run[(int)x], (chan)complete[(int)x], (chan)preempt[(int)x]</td>
<td>Are channels used to synchronize every thread transition of every thread in the system. Synchronization with a particular thread is done through its identity. For example, run[2] is a synchronization channel with thread having identity equal to 2.</td>
</tr>
<tr>
<td>(void)schprotocol((int)x)</td>
<td>Is a function sorting threads in the ready queue according to a given scheduling policy. The function is called each time a thread dispatches where the thread's identity is given as argument to the function. In this example, we assume fixed priority scheduling.</td>
</tr>
<tr>
<td>(void)completion((int)x)</td>
<td>Is a function removing threads from the ready queue. The function is called each time a thread completes its execution, where the thread's identity is given as argument to the function.</td>
</tr>
<tr>
<td>(void)addTime()</td>
<td>Is a function adding preempted time to the threads in the preempt stack. The function is called when preemption occurs, whereupon the execution time of the thread causing the preemption is added to the preemption time of every preempted thread.</td>
</tr>
<tr>
<td>(void)checkTime((int)x)</td>
<td>Is a function adding preempted time to the threads in the (int)preempt_stack[x][2] stack. The function is called when a thread dispatch not causing any preemption occurs, to check if the dispatched thread is prior to any preempted threads in the ready queue whereupon preemption time is added.</td>
</tr>
</tbody>
</table>

Table 5: The Scheduler Automaton elements [1]

The scheduler is initially in the **Empty** location, waiting for the five threads (i.e. SoftwareIn, AE201_FuelEstimation, AE202_FuelWarning, Other_AEs, and SoftwareOut) to be dispatched. The thread will be dispatched simultaneously since they share a common period of 10ms. Once these threads have been dispatched, the scheduler transits to the **schedule1** location which is a committed repetition of the **Empty** location, allowing several threads to be dispatched (through the edge to the same location) simultaneous. In **schedule1** the five threads will be sorted in the **ready_queue**, where the first element in the queue is the thread being processed and the second element is the next to be processed ... etc. Sorting is performed by the **schprotocol()** function which is called each time a thread is dispatched. Sorting the threads in the **ready_queue** depends
on the priority of each thread, thus, SoftwareIn, AE201_FuelEstimation, AE202_FuelWarning, Other_AEs, and then SoftwareOut will be sorted in that order, where SoftwareIn has the highest priority and SoftwareOut has the lowest. The scheduler synchronizes with the first thread in the 
ready_queue and transits to the Running location through one of three different edges depending on the number of currently preempted threads, i.e. nr_preempted, and preempt_stack.

If there are no preempted threads, or there is at least one preempted thread and the latest preempted thread is not the first in the ready_queue, then the execution time clock of the thread to be run is reset. For example, if AE201_FuelEstimation is preempted where SoftwareIn is the first thread in the ready_queue, then the execution time of the AE201_FuelEstimation will be reset. However, if there are more than one preempted threads and the latest preempted thread is the first thread in the ready_queue, the scheduler transits to the Running location without resetting its execution time clock. In our case, SoftwareIn is the first thread in the ready_queue and therefore it will run first (in each dispatch) then AE201_FuelEstimation and so forth until SoftwareOut. The scheduler will remain in the Running location until SoftwareIn threads completes its execution, misses its deadline, or another thread is dispatched. The Running location has been modeled with an invariant to force a fire of the completion edge at completion time. When SoftwareIn has been successfully executed the scheduler will transit to the Anonymous location. The transition to the Anonymous location is dependent on the thread execution time and whether the thread is preempted or not. For instance, if SoftwareIn has not been preempted then it will be removed from the ready_queue by the completion() function, otherwise, it will be removed from both ready_queue and preempt_stack. From the Anonymous location, there are two locations that the scheduler can transit to: the Empty location or the Schedule1 location, this depends if there are any dispatched threads or not.

If Other_AEs thread is dispatched while SoftwareIn is being executed (scheduler in the Running location), an edge is fired to the Schedule2 location, where Other_AEs is added to the ready_queue and the corresponding execution time clock is reset. From Schedule2 location there are two different edges. If Other_AEs is scheduled as a first thread in the ready_queue (after SoftwareIn, AE201_FuelEstimation, and AE202_FuelWarning), then the scheduler will transit to the Preemption location where the thread is synchronized for execution. Otherwise, Other_AEs will wait until executing AE202_FuelWarning, which requires adding this thread to the preempt_stack and adding additional time to it through addTime() function. However, if Other_AEs thread does not cause a preemption, no further actions are taken other than adding preempted time to the preempted thread through checkTime() function. For Schedulability analysis, MissedDeadline location was modeled to indicate the missed deadline for running threads, where a transit to MissedDeadline location indicates a missed deadline.
4.4.3 Mapping the Verification Sequences in the UPPAAL Model and Perform the Model checking: Step 3 & 4

The third step of the verification technique is to transform the verification sequences which have been generated before (in section 4.4.1) to the corresponding timed automata paths through a structural mapping between them. Since it is not feasible to show the graphical transformations of the paths contained in the three verification sequences, thus an example is given to demonstrate the structural mapping. The following underlined path will be considered in the example.

**Sequence 1:** SoftwareIN.SetSensorData, FuelEstimation.GetSensorValue_2, FuelLevelWarning.GetEstimatedFuelLevelValue_2, SoftwareOUT.GetFinalValues, SoftwareOUT.FuelLevelWarningValue

This path starts in SoftwareIN.SetSensorData and ends in FuelEstimation.GetSensorValue_2, and as explained before this path represents a data flow between SoftwareIN and FuelEstimation threads, where the data which will be used by FuelEstimation should be first set by SoftwareIN. Transforming this to UPPAAL requires that the input of AE201_FuelEstimation automaton must be equal to the output of SoftwareIn automaton. Therefore, SetSensorData and GetSensorValue_2 interfaces must be assigned to the same connection name which is in this example SetSensorConvertedValue as shown in figure 36.

![Figure 36: Representation of a path mapping in UPPAAL.](image-url)
Mapping of the rest sub paths of the given path is listed in table 6.

<table>
<thead>
<tr>
<th>Paths in AADL</th>
<th>Paths in UPPAAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FuelLevelWarning.GetEstimatedFuelLevelValue_2</td>
<td>GetEstimatedFuelLevelValue_2_In = GetSensorValue_SetEstimatedValue_RTDB</td>
</tr>
<tr>
<td>SoftwareOUT.GetFinalValues</td>
<td>GetFinalValues = GetSensorValue_SetEstimatedValue_RTDB</td>
</tr>
<tr>
<td>SoftwareOUT.FuelLevelWarningValue</td>
<td>SendFinalFuelEstimationValue = FinalEstimationValue, SendFinalFuelLevelWarningValue = FuelLevelWarningValue</td>
</tr>
</tbody>
</table>

Table 6: structural mapping between a verification path to the corresponding timed automata paths

Before proceeding with the fourth step, it is important to show the constraints of the timed automata paths which are specified in the sequences generated from the AADL specification. For the three examples of the extracted sequences, there are four constrains for each sequence: Priority, Compute_Deadline, Compute_Execution_Time, and Period. However, these constraints are assigned to the threads included in the sequences, which has a different constraint(s) depending on the included threads. The only shared constraint for all sequences is the Period property which is equal 10ms, and it is shared because all the thread will be dispatched periodically every 10ms. All other thread constraints are represented in figure 32.

**Sequence 1: SoftwareIN.SetSensorData, FuelEstimation.GetSensorValue_2,**
**FuelLevelWarning.GetEstimatedFuelLevelValue_2, SoftwareOUT.GetFinalValues,**
**SoftwareOUT.FuelLevelWarningValue**

- **Priority constraints:**
  SoftwareIN.SetSensorData (1st priority), FuelEstimation.GetSensorValue_2 (2nd priority),
  FuelLevelWarning.GetEstimatedFuelLevelValue_2 (3rd priority), SoftwareOUT.GetFinalValues,
  SoftwareOUT.FuelLevelWarningValue (5th priority)

- **Compute_Deadline constraints** (Deadline = 8 ms):
  SoftwareIN.SetSensorData (1 ms), FuelEstimation.GetSensorValue_2 (2 ms),
  FuelLevelWarning.GetEstimatedFuelLevelValue_2 (3 ms), SoftwareOUT.GetFinalValues,
  SoftwareOUT.FuelLevelWarningValue (8 ms)
- **Compute Execution Time constraints:**

  SoftwareIN.SetSensorData (1 ms), FuelEstimation.GetSensorValue_2 (1 ms), FuelLevelWarning.GetEstimatedFuelLevelValue_2 (1 ms), SoftwareOUT.GetFinalValues, SoftwareOUT.FuelLevelWarningValue (1 ms)

**Sequence 2:** SoftwareIN.SetSensorData, FuelEstimation.GetSensorValue_2, SoftwareOUT.GetFinalValues, SoftwareOUT.FinalEstimationValue

  - **Priority constraints:**

    SoftwareIN.SetSensorData (1st priority), FuelEstimation.GetSensorValue_2 (2nd priority), SoftwareOUT.GetFinalValues, SoftwareOUT.FinalEstimationValue (5th priority)

  - **Compute Deadline constraints:** {Deadline = 8 ms}

    SoftwareIN.SetSensorData (1ms), FuelEstimation.GetSensorValue_2, (2ms) SoftwareOUT.GetFinalValues, SoftwareOUT.FinalEstimationValue (8ms)

  - **Compute Execution Time constraints:**

    SoftwareIN.SetSensorData (1 ms), FuelEstimation.GetSensorValue_2 (1 ms), SoftwareOUT.GetFinalValues, SoftwareOUT.FinalEstimationValue (1 ms)

**Sequence 3:** SoftwareIN.SetSensorData, OtherAEs.ReadWrite, SoftwareOUT.GetFinalValues

**Priority constraints:**

SoftwareIN.SetSensorData (1st priority), OtherAEs.ReadWrite (4th priority), SoftwareOUT.GetFinalValues (5th priority)

**Compute Deadline constraints:** {Deadline = 8 ms}

SoftwareIN.SetSensorData (1 ms), OtherAEs.ReadWrite (7 ms), SoftwareOUT.GetFinalValues (8ms)

**Compute Execution Time constraints:**

SoftwareIN.SetSensorData (1 ms), OtherAEs.ReadWrite (4ms), SoftwareOUT.GetFinalValues (1ms)

Subsequent to the extraction of verification sequences and the transformation to UPPAAL, model checking is used to evaluate the completeness and the consistency (verification objective) of the AADL specification of the COO. The completeness and the consistency can be assured by analyzing the Control flow reachability, Data flow reachability, and Concurrency among flows. The coverage requirements for the model checking will require set of simulations of an AADL to cover all three types of sequences which are defined by the technique (i.e. Component Internal Transfer, Direct Component to Component, and Indirect Component to Component).
- **Control flow Reachability**

Control flow reachability means that every architectural element in an execution order should be able to reach the subsequent element to be executed in the order. The subsequent element should be reached without conflicting properties (constraints) of the execution order. Examining the control flow reachability in COO system requires a set of UPPAAL queries for each verification sequence. The following queries under each sequence are not comprehensive and it is possible to have more combinations of test queries. Nevertheless, the objective of these queries is to give a good understanding of how the Control flow reachability, Data flow reachability, and Concurrency are satisfied in fuel level estimation system. However, more queries can be found in the appendix section.

**Sequence 1:** SoftwareIN.SetSensorData, FuelEstimation.GetSensorValue_2, FuelLevelWarning.GetEstimatedFuelLevelValue_2, SoftwareOUT.GetFinalValues, SoftwareOUT.FuelLevelWarningValue

**Query 1:** (Property is satisfied) 
\[ E<>\text{SoftwareIN.SetSensorData} \text{ and } \text{FuelEstimation.GetSensorValue}_2 \text{ and } \text{FuelLevelWarning.GetEstimatedFuelLevelValue}_2 \text{ and } \text{SoftwareOUT.GetFinalValues} \text{ and } \text{SoftwareOUT.FuelLevelWarningValue} \]

This query means that there exists a path where sensor value can be received by the SoftwareIN thread where FuelEstimation, FuelLevelWarning and SoftwareOUT threads can be reached.

It is efficient to test the reachability of the control flow within one single dispatch, which is 10ms. In the UPPAAL model, there is a clock (cl), which is responsible to monitor the dispatch time and after the dispatching it will reset. Therefore, the dispatching is periodic and it will be understandable why SoftwareOUT can be in the Running location while SoftwareIN in the schedule1 location. This will not allow us to verify the threads order and it is needed to represent the threads in a sequence rather than a cycle. Thus, another clock has been added (i.e. test) which simulates the time between every two dispatches. Since the dispatch period is 10 ms, then the test clock will work after 10ms for another 10ms. This means that dispatch time will seems to be 20 ms, but of course this is only for testing purpose while the period is still 10ms. Based on that, the second query checks the execution order for SoftwareIN and FuelEstimation.

**Query 2:** (Property is satisfied) 
\[ \text{SoftwareIN.running and test<20 } \Rightarrow \text{FuelEstimation.running and test < 20} \]

The query means that whenever the dispatch period less than 20 (based on the explanation above, it means for one single dispatch) and the SoftwareIN in the running location, FuelEstimation will eventually enter the running location. This proves that the order is correct.
since FuelEstimation has a lower priority value and it must be executed after SoftwareIN which has the higher priority in the execution order.

In the third query, the order will be set incorrectly and intentionally to test if it will be satisfied or not.

**Query 3:** (Property is not satisfied)

\[ \text{SoftwareOUT.running and test < 20} \rightarrow \text{SoftwareIN.running and test < 20} \]

The query is not satisfied since the priority of SoftwareOUT thread is lower than SoftwareIN thread. Therefore, SoftwareOUT cannot be executed before SoftwareIN thread through one single dispatch.

**Query 4:** (Property is satisfied)

\[ \text{FuelEstimation.running and test < 20} \rightarrow \text{FuelLevelWarning.running and test < 20} \]

The same description of the second query

**Query 5:** (Property is satisfied)

\[ \text{FuelLevelWarning.running and test < 20} \rightarrow \text{SoftwareOUT.running and test < 20} \]

The same description of the second and fourth queries

**Sequence 2:** SoftwareIN.SetSensorData, FuelEstimation.GetSensorValue_2, SoftwareOUT.GetFinalValues, SoftwareOUT.FinalEstimationValue

The first path in the second sequence is similar to the first path in the first sequence and the second query shows that this path is tested and satisfied.

**Query 6:** (Property is satisfied)

\[ \text{FuelEstimation.running and test < 20} \rightarrow \text{SoftwareOUT.running and test < 20} \]

The query shows that Whenever FuelEstimation is in the running and within one dispatch period, then SoftwareOUT will eventually enter the running location.

**Query 7:** (Property is not satisfied)

\[ \text{SoftwareOUT.running and test < 20} \rightarrow \text{FuelEstimation.running and test < 20} \]

This query is not satisfied since FuelEstimation has a higher priority than SoftwareOUT, thus SoftwareOUT cannot run before FuelEstimation

**Sequence 3:** SoftwareIN.SetSensorData, OtherAEs.ReadWrite, SoftwareOUT.GetFinalValues

**Query 8:** (Property is satisfied)

\[ \text{SoftwareIN.running and test < 20} \rightarrow \text{OtherAEs.running and test < 20} \]

Whenever SoftwareIN thread in the running state and within one dispatch, then the OtherAEs thread will eventually run. This is because the priority of SoftwareIN is higher than OtherAEs thread priority.
Query 9: (Property is satisfied)

OtherAEs.running and test < 20 → SoftwareOUT.running and test < 20

Whenever OtherAEs thread in the running location and within one dispatch, then the SoftwareOUT will eventually run. This is because the priority of OtherAEs is higher than SoftwareOUT thread priority.

Query 10: (Property is not satisfied)

SoftwareOUT.running and test < 20 → OtherAEs.running and test < 20

Not satisfied because SoftwareOUT thread can be executed before OtherAEs thread which has the higher priority.

- Data flow Reachability

Data flow reachability means that every data element should be able to reach its target component, where the data is used, from its source component, where the data is defined. The target component should be reached without conflicting properties of the data flow. Therefore, it is important to define a data and check if this defined data can reach the components (threads) in the verification sequences. Consequently, an assumption has been made to consider “6” as the value received by the fuel level sensor. Regardless of the processes or the calculations that can be made on this value through the sequence, it must be able to reach all components (threads).


Query 11: (Property is satisfied)

SoftwareIN.SensorValue_3 == 6 → FuelEstimation.GetSensorValue_2_In == 6

Whenever SoftwareIN.SensorValue_3 is equal to 6, FuelEstimation.GetSensorValue_2_In will eventually equal to 6.

Query 12: (Property is not satisfied)

SoftwareIN.SensorValue_3 == 6 → FuelEstimation.GetSensorValue_2_In == 10

Whenever SoftwareIN.SensorValue_3 is equal to 6, FuelEstimation.GetSensorValue_2_In will eventually equal to 10. FuelEstimation thread cannot have different value.

Query 13: (Property is satisfied)


Whenever FuelEstimation.GetSensorValue_2_In is equal to 6, FuelLevelWarning.GetEstimatedFuelLevelValue_2 will eventually equal to 6.

Query 14: (Property is satisfied)

FuelLevelWarning.GetEstimatedFuelLevelValue_2_Out == 6 → SoftwareOUT.GetFinalValues == 6
Whenever FuelLevelWarning.GetEstimatedFuelLevelValue_2 is equal to 6, SoftwareOUT.GetFinalValues will eventually equal to 6.

**Query 15:** (Property is satisfied)

\[ \text{SoftwareOUT.GetFinalValues} == 6 \rightarrow \text{SoftwareOUT.FuelLevelWarningValue} == 6 \]

Whenever SoftwareOUT.GetFinalValues is equal to 6, SoftwareOUT.FuelLevelWarningValue will eventually equal to 6.

**Sequence 2:** SoftwareIN.SetSensorData, FuelEstimation.GetSensorValue_2, SoftwareOUT.GetFinalValues, SoftwareOUT.FinalEstimationValue

**Query 16:** (Property is satisfied)

\[ \text{SoftwareIN.SetSensorData} == 6 \rightarrow \text{FuelEstimation.GetSensorValue_2} == 6 \]

Whenever SoftwareIN.SensorValue_3 is equal to 6, FuelEstimation.GetSensorValue_2 will eventually equal to 6.

**Query 17:** (Property is satisfied)

\[ \text{FuelEstimation.GetSensorValue_2} == 6 \rightarrow \text{SoftwareOUT.GetFinalValues} == 6 \]

Whenever FuelEstimation.GetSensorValue_2 is equal to 6, SoftwareOUT.GetFinalValues will eventually equal to 6.

**Query 18:** (Property is satisfied)

\[ \text{SoftwareOUT.GetFinalValues} == 6 \rightarrow \text{SoftwareOUT.FinalEstimationValue} == 6 \]

Whenever SoftwareOUT.GetFinalValues is equal to 6, SoftwareOUT.FinalEstimationValue will eventually equal to 6.

**Sequence 3:** SoftwareIN.SetSensorData, OtherAEs.ReadWrite, SoftwareOUT.GetFinalValues

**Query 19:** (Property is satisfied)

\[ \text{SoftwareIN.SetSensorData} == 6 \rightarrow \text{OtherAEs.ReadWrite} == 6 \]

Whenever SoftwareIN.SetSensorData is equal to 6, OtherAEs.ReadWrite will eventually equal to 6.

**Query 20:** (Property is satisfied)

\[ \text{OtherAEs.ReadWrite} == 6 \rightarrow \text{SoftwareOUT.GetFinalValues} == 6 \]

Whenever OtherAEs.ReadWrite is equal to 6, SoftwareOUT.GetFinalValues will eventually equal to 6.

---

**Concurrency**

Concurrency among flows is the property where dependencies do not prevent control flow reachability or data flow reachability, and where the system should be free from deadlocks. UPPAAL provides a special form to verify the deadlock property, which means that all paths in
the model do not conflict each other, as following, “A [ ] not deadlock”. The query has been checked using UPPAAL verifier and it is satisfied as shown in figure 37. In addition to the form provided by UPPAAL, the Schedule automaton contains “MissedDeadline” location, and as described before, this location is modeled for Schedulability analysis. In other words, the scheduler transits to this location only if a thread in the Running location missed its deadline, where the transition to this MissedDeadline location yields a deadlock. Therefore, verifying that there is no transition to the MissedDeadline location means that there is no MissedDeadline, which results a deadlock fee system. The query: A[] not Scheduler.MissedDeadline has been check and the property is satisfied.

![Figure 37: Uppaal Verification](image)

Table 7 summarizes the amount of relations, amount of checked paths, and amount of satisfied paths in the COO ECU:
A set of the model checking queries can be found in appendix A.
5. Discussion

In this section, we firstly describe our experience in validating the practical applicability of the verification technique (i.e., feasibility). Secondly, we partially describe the efficacy of the verification technique.

We reveal the feasibility of applying the technique by taking each step (see section 3.5.1) in isolation and describe our experience while performing it.

**Step 1. Use the transformation rules to transform an AADL specification to an UPPAAL model upon which automated formal verification can be performed.**

The technique guide [1] is not clear about which parts of the AADL model that should be transformed. This case study considered only the threads to be transformed. All other AADL elements were not transformed (e.g., data, processes, thread group, etc.). Furthermore, the transformation from an AADL model to an UPPAAL model is done manually. This means that there is a possibility to omit some parts of the AADL. This may mask some consistency or completeness issues. Having an automated method may mitigate the risk during the transformation.

**Step 2. Apply the architecture-based verification criteria (section 3.5.2) to the AADL specification.**

Achieving this step was not hard so to speak. This step might be simple for small systems. But extracting the control and data flows in bigger systems might be more challenging. It is worth noting that the fuel level estimation system is quite straightforward system and this has limited the provision of the generated paths. This means that not all relations’ types could be validated in this case study.

The verification technique is very dependent on the execution paths. However, extracting the control and data flow paths according to the relations’ types has been performed manually. This may increase the risk of omitting some paths. Omitting some paths means that some relations and interface may not be model checked.

**Step 3. Sequences from the previous step are transformed, in this step, to the corresponding automata paths in the UPPAAL model through a structural mapping between them.**

Mapping between four AADL threads and four UPPAAL automata was painstaking. This is because specifying the relations between the automata should be according the specified relations between the AADL threads. For more complex systems with, for example, more than 20
threads, this step is expected to be uneasy. The suggestion here is to automate the mapping between the AADL and the UPPAAL model.

**Step 4. The outcomes from the previous 3 are required, in this step, to be fully simulated in UPPAAL by using temporal logics.**

This step was performed easily. However, it is very dependent on the previous steps. If all previous steps were performed properly, achieving this step will be very simple. The good side in this step is that it reveals any problem that can preclude the simulation. Simulating all the automata may reveal the consistency and completeness of the AADL specification if all these specification were represented correctly in the UPPAAL model. A possible risk in this step is that it does not tell about the missing specifications if the current specifications do not preclude the simulation.

**Step 5. The paths are later used in this step to generate test cases to the implementation, to test the conformance of the implementation with respect to the architecture specification.**

This step could not be covered through the case study. Consequently, verification technique feasibility was not validated with respect to this step.

ISO 26262 recommends formally verifying software architectural designs at the highest ASIL levels C and D [37]. Model checking is one means of such verification. Model checking is particularly desirable because model checkers can produce counterexamples illustrating how and where verification checks fail. Model checkers can detect deadlocks, incorrect interactions, especially within the concurrency, of the model. The techniques can also identify correctness, completeness, or coherency issues with the model. All these issues can lead to the final system failing and possibly causing harm [36].

In the previous section, we used the UPPAAL verifier to write temporal logic queries to check if whether the paths’ properties are satisfied or not. More specifically, we checked:

- Whether the scheduler automaton dispatch the automata (i.e., transformed AADL threads) according to the specified priorities.
- Whether the automata meet their specified deadlines.
- The data-flow reachability for each automaton in the paths.
- The control-flow reachability for each automaton in the paths.

The following table summarizes where the results of the verification technique application contributed to fulfill some software architecture verification requirements from the ISO 26262.
<table>
<thead>
<tr>
<th>Req. Satisfied</th>
<th>Evidence Source(s)</th>
<th>ISO26262 Clause Satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataAccess</strong></td>
<td>The Verification Technique</td>
<td>§7.4.5-b: Dynamic design aspects address the data flow between the software components; data flow at external interfaces.</td>
</tr>
<tr>
<td><strong>DeadlocksFree</strong></td>
<td>The Verification Technique</td>
<td>Part 6 – Annex D.2.2 - Timing and execution: effects of faults deadlocks can be considered for the software elements executed in each software partition</td>
</tr>
<tr>
<td><strong>LivelocksFree</strong></td>
<td>The Verification Technique</td>
<td>Part 6 – Annex D.2.2 - Timing and execution: effects of faults livelocks can be considered for the software elements executed in each software partition</td>
</tr>
</tbody>
</table>

*Table 8: The contribution of the verification technique to ISO26262 SW architecture verification requirements.*
6. Summary and Conclusion

The work in this thesis represents a case study of a well-defined architecture-based verification technique for AADL specifications. The technique has been innovated by a group of researchers at Mälardalen University in Sweden. This technique is formally specified but it was not validated against any real-world system. The goal of this master thesis work is to validate the technique and assess its feasibility.

The goal of the technique is to evaluate the integration of components at both the specification level and the implementation level. The technique is driven by the architecture verification criteria. These criteria define the test objectives (i.e., what should be tested). Moreover, the criteria define test selection in terms of what samples to evaluate, and how they are generated. Furthermore, the criteria define the test coverage, which determines how many samples to evaluate.

Applying the verification criteria to an AADL specification generates integration verification sequences. Thereafter, the sequences were used for simulations of the architecture specification (model checking). These sequences can be used to generate test cases to test the system implementation (model-based testing). The technique objectives are to evaluate some architecture specification and assess the conformance of the implementation with respect to its architecture specification. The technique is based on formal constructs enabling automation of the verification activities. However, the AADL lacks the formal and implemented semantics that are needed to automate the formal verification. Thus, the technique states to transform the AADL constructs to timed automata model, where UPPAAL (model checker) was adopted for this purpose. [1]

The case study aimed to validate the verification technique by applying it to a fuel level estimation system as a real safety-critical system. This system is already implemented and used by a major vehicle manufacturer in Sweden. It is considered a safety-critical system because if it fails to measure the fuel in the tank correctly, then it will not indicate the actual fuel level. Moreover, the fuel may run out and the driver cannot determine the correct time to refuel.

Consequently, there will be a probability of a sudden stop of the engine, which may yield to a hazardous situation while driving. The system has been investigated and specified using AADL specifications. The documentation of the implemented system was not updated and not fully specified, thus, it was uneasy to get a concrete understanding or reach particular information related to the system functionality. This forced us to make two main assumptions but without
affecting the main functionality of the system. We assumed the deadline values of the functions (AEs) involved in the system (i.e. SoftwareIn, AE201, AE202, etc.). In addition, we abstracted all functions contained in the COO ECU, which are not involved in the fuel level estimation system, where all these functions have been represented by OthersAEs thread.

The AADL model of the fuel level estimation system was included the full system. However, in this paper only the COO ECU has been considered in the verification. According to the verification technique first step, the defined verification criteria have been applied to AADL model. Applying these criteria yielded to generate the verification paths (sequences) as different paths of the control flow and data flow in the AADL specification of the fuel level estimation system. The extraction of these paths carried out based on three main relations: 1) Component to component relations, 2) Direct component-to-component relations, and 3) indirect component to component relations. Subsequently, and based on the second the AADL model of the system has been transformed into an UPPAAL model according to the transformation rules. Each AADL thread has been transformed into an automaton. This led to have five automata as, SoftwareIN, FuelEstimation, FuelLevelWarning, OtherAEs, SoftwareOUT, and the scheduler, which represents the processor of the COO ECU. In the third step, the verification paths (sequences) that were generated by step 1 have been mapped to the corresponding timed automata paths modeled in step 2. This structural mapping was represented in figure 36 and table 6. Finally, in the fourth step, all generated paths by step 1 have been checked against their constraints (i.e. the priority, deadline, and execution time). Checking these paths required UPPAAL queries that are expressed in temporal logics.

It is worth noting that the fuel level estimation is a single core system with non-preemptive scheduling and little computation done in interrupt handler. Therefore, there is no concurrency that exists during thread execution.

Using the verification technique has showed interesting results while performing the model checking. However, the fuel level estimation system is quite straightforward system and this has limited the provision of the generated paths. This means that not all the relations’ types could be validated. More complex system leads to more possible checks. For instance, if the AADL model of the tackled system contained Behavior Annex (BA) in one of its threads, this would give a chance to validate the technique in terms of checking the consistency and completeness with the existence of the internal relations. This added some limitation to the case study.

The feasibility of applying the verification technique was discussed. The efficacy of the technique with respect to ISO26262 requirements was also discussed.
7. Future Work

The fifth step of the verification technique states to generate test cases to test the conformance of the implementation with respect to the architecture specification (model-based testing). Test paths should be transformed to concrete test cases through a mapping between the architecture specification and its implementation. Covering this step required a prepared test environment supported with the proper hardware and ECUs needed for the fuel level estimation system. Since the thesis work was not carried in the company where the system is in use, thus, this step could not be covered through the case study. Consequently, the feasibility of verification technique was not validated with respect to the fifth step. However, it is an interesting topic for future work.

In addition, it will also be very interesting to apply the technique to more complex systems, where more design details and options can be specified by the AADL. This will emerge a plenty of generated paths when applying the verification criteria to the AADL models of these systems.
8. References


[26] System Design Analysis. US. Department of Transportation, Federal Aviation Administration, (July 1982).


### Appendix A

Table of the checked path with the corresponding UPPAAL queries and their status:

<table>
<thead>
<tr>
<th>No</th>
<th>Path</th>
<th>Query</th>
<th>Check Type</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SoftwareIN.SetSensorData (\rightarrow) FuelEstimation.GetSensorValue_2_In</td>
<td>SoftwareIN.SetSensorData (==) 3 (\rightarrow) FuelEstimation.GetSensorValue_2_In (==) 3</td>
<td>Data flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>2</td>
<td>FuelEstimation.GetSensorValue_2_In (\rightarrow) FuelLevelWarning.GetEstimatedFuelLevelValue_2_In</td>
<td>FuelEstimation.GetSensorValue_2_In (==) 3 (\rightarrow) FuelLevelWarning.GetEstimatedFuelLevelValue_2_In (==) 3</td>
<td>Data flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>3</td>
<td>FuelLevelWarning.GetEstimatedFuelLevelValue_2_Out (\rightarrow) SoftwareOUT.GetFinalValues</td>
<td>FuelLevelWarning.GetEstimatedFuelLevelValue_2_Out (==) 3 (\rightarrow) SoftwareOUT.GetFinalValues (==) 3</td>
<td>Data flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>4</td>
<td>FuelLevelWarning.GetEstimatedFuelLevelValue_2_Out (\rightarrow) SoftwareOUT.FuelLevelWarningValue</td>
<td>FuelLevelWarning.GetEstimatedFuelLevelValue_2_Out (==) 3 (\rightarrow) SoftwareOUT.FuelLevelWarningValue (==) 3</td>
<td>Data flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>5</td>
<td>FuelEstimation.GetSensorValue_2_Out (\rightarrow) SoftwareOUT.GetFinalValues</td>
<td>FuelEstimation.GetSensorValue_2_Out (==) 6 (\rightarrow) SoftwareOUT.GetFinalValues (==) 6</td>
<td>Data flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>6</td>
<td>SoftwareIN.SetSensorData (\rightarrow) OtherAEs.ReadWrite_In</td>
<td>SoftwareIN.SetSensorData (==) 6 (\rightarrow) OtherAEs.ReadWrite_In (==) 6</td>
<td>Data flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>7</td>
<td>OtherAEs.ReadWrite_Out (\rightarrow) SoftwareOUT.GetFinalValues</td>
<td>OtherAEs.ReadWrite_Out (==) 6 (\rightarrow) SoftwareOUT.GetFinalValues (==) 6</td>
<td>Data flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>8</td>
<td>SoftwareIN (\rightarrow) FuelEstimation</td>
<td>SoftwareIN.running and test&lt;20 (\rightarrow) FuelEstimation.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>9</td>
<td>FuelEstimation (\rightarrow) FuelLevelWarning</td>
<td>FuelEstimation.running and test&lt;20 (\rightarrow) FuelLevelWarning.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>10</td>
<td>FuelLevelWarning (\rightarrow) SoftwareOUT</td>
<td>FuelLevelWarning.running and test&lt;20 (\rightarrow) SoftwareOUT.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>11</td>
<td>FuelEstimation (\rightarrow) SoftwareOUT</td>
<td>FuelEstimation.running and test&lt;20 (\rightarrow) SoftwareOUT.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>12</td>
<td>OtherAEs (\rightarrow) SoftwareOUT</td>
<td>OtherAEs.running and test&lt;20 (\rightarrow) SoftwareOUT.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>13</td>
<td>SoftwareIN (\rightarrow) SoftwareOUT</td>
<td>SoftwareIN.running and test&lt;20 (\rightarrow) SoftwareOUT.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>14</td>
<td>OtherAEs (\rightarrow) FuelLevelWarning</td>
<td>OtherAEs.running and test&lt;20 (\rightarrow) FuelLevelWarning.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>15</td>
<td>FuelEstimation (\rightarrow) OtherAEs</td>
<td>FuelEstimation.running and test&lt;20 (\rightarrow) OtherAEs.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Satisfied</td>
</tr>
<tr>
<td>16</td>
<td>SoftwareOUT (\rightarrow) FuelEstimation</td>
<td>SoftwareOUT.running and test&lt;20 (\rightarrow) FuelEstimation.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Not Satisfied</td>
</tr>
<tr>
<td>17</td>
<td>SoftwareOUT (\rightarrow) OtherAEs</td>
<td>SoftwareOUT.running and test&lt;20 (\rightarrow) OtherAEs.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Not Satisfied</td>
</tr>
<tr>
<td>18</td>
<td>SoftwareOUT (\rightarrow) OtherAEs</td>
<td>SoftwareOUT.running and test&lt;20 (\rightarrow) OtherAEs.running and test&lt;20</td>
<td>Control flow Reachability</td>
<td>Not Satisfied</td>
</tr>
<tr>
<td>19</td>
<td>[A] not Scheduler.MissedDeadline</td>
<td></td>
<td>Concurrency</td>
<td>Satisfied</td>
</tr>
<tr>
<td>20</td>
<td>[A] not deadlock</td>
<td></td>
<td>Concurrency</td>
<td>Satisfied</td>
</tr>
</tbody>
</table>
Appendix B

The AADL textual representation for the fuel level estimation system:

```plaintext
system Item_FuelLevelModel
end Item_FuelLevelModel;

system COO
  features
    SensorValue: in data port;
    LowFuelLevelWarning: out event data port;
    FuelLevel: out event data port;
  end COO;

system ICL
  features
    FuelLevelDisplaySignal: out data port;
    LampSignal: out data port;
    LowFuelLevelWarningValue: in event data port;
    EstimatedFuelLevelValue: in event data port;
  end ICL;

process B_Software
  features
    SensorValue_2: in data port;
    AES_Connection: provides data access;
    EstimatedFuelValue: out event data port;
    LevelWarningValue: out event data port;
  end B_Software;

process AE1_FuelEstimation
  features
    GetSensorValue: requires data access;
  end AE1_FuelEstimation;

process AE2_LevelWarning
  features
    GetEstimatedFuelLevelValue: requires data access;
  end AE2_LevelWarning;

thread FuelLevelEstimation
  features
    GetSensorValue_2: requires data access;
  end FuelLevelEstimation;

thread FuelLevelWarning
  features
    GetEstimatedFuelLevelValue_2: requires data access;
  end FuelLevelWarning;

thread B_Software_IN
  features
    SensorValue_3: in data port;
    SetSensorData: requires data access;
  end B_Software_IN;

thread B_Software_OUT
  features
    GetFinalValues: requires data access;
    FinalEstimationValue: out event data port;
    FuelLevelWarningValue: out event data port;
```
end B_Software_OUT;

data RTDB_Array
features
  SendReceiveSharedData: provides data access;
end RTDB_Array;

system implementation COO.imp
subcomponents
  BasicSoftware: process B_Software.imp;
  FuelEstimationCalculation: process AE1_FuelEstimationCalculation.imp;
  FuelLevelWaningCalculation: process AE2_LevelWarningCalculation.imp;
  SharedMemo: memory SharedMemory.imp;
  COO CPU: processor COO7_CPU.imp {
    SEI::cycle_time => 1000 ps;
  };
  COO_Other_AEs: process COO_Other_AEs.imp;
connections
  FuelLevelSensorValue: data port SensorValue ->
  BasicSoftware.SensorValue_2 {
    Allowed_Connection_Binding_Class => bus FuelLevelSensorWire.imp;
  };
  GetSavedSensorValue: data access BasicSoftware.AEs_Connection ->
  FuelEstimationCalculation.GetSensorValue;
  GetSavedFuelEstimatedLevel: data access BasicSoftware.AEs_Connection ->
  FuelLevelWaningCalculation.GetEstimatedFuelLevelValue;
  ReadWrite: data access BasicSoftware.AEs_Connection ->
  _CalculatedFuelLevel: event data port BasicSoftware.EstimatedFuelValue ->
  FuelLevel;
  CalculatedLevelWarning: event data port BasicSoftware.LevelWarningValue ->
  LowFuelLevelWarning;
properties
  -- Actual_Connection_Binding => CANBus;
  Actual_Processor_Binding => reference coo_cpu;
  Actual_Processor_Binding => reference coo_cpu applies to
  BasicSoftware;
  Actual_Processor_Binding => reference coo_cpu applies to
  FuelEstimationCalculation;
  Actual_Processor_Binding => reference coo_cpu applies to
  FuelLevelWaningCalculation;
  Actual_Memory_Binding => reference sharedmemo;
  Available_Processor_Binding => reference coo_cpu;
  Allowed_Processor_Binding => reference coo_cpu;
end COO.imp;

thread implementation B_Software_OUT.imp
properties
  Allowed_Processor_Binding_Class => processor COO7_CPU.imp;
  Dispatch_Protocol => Periodic;
  Period => 10 Ms;
  SEI::Priority => 5;
  Compute_Deadline => 8 Ms;
  Compute_Execution_Time => 100 Us .. 1 Ms;
end B_Software_OUT.imp;

thread implementation B_Software_IN.imp
properties
  SEI::Priority => 1;
  Compute_Deadline => 8 Ms;
  Compute_Execution_Time => 600 Us .. 1 Ms;

86
Dispatch_Protocol => Periodic;
Period => 10 Ms;
Allowed_Processor_BINDING_Class => processor COO7_CPU.imp;
end B_Software_IN.imp;

data implementation RTDB_Array.imp
properties
Source_Data_Size => 1024 Bits;
Source_Language => C;
end RTDB_Array.imp;

process implementation AE1_FuelEstimation.imp
subcomponents
FuelEstimation: thread FuelLevelEstimation.imp {
  Compute_Deadline => 8 Ms;
};
connections
GetSensorValue_SetEstimatedValue_RTDB: data access GetSensorValue ->
FuelEstimation.GetSensorValue_2;
end AE1_FuelEstimation.imp;

thread implementation FuelLevelEstimation.imp
properties
Allowed_Processor_BINDING_Class => processor COO7_CPU.imp;
Dispatch_Protocol => Periodic;
SEI::Priority => 2;
Compute_Deadline => 8 Ms;
Compute_Execution_Time => 500 Us .. 1 Ms;
Period => 10 Ms;
end FuelLevelEstimation.imp;

process implementation AE2_LevelWarning.imp
subcomponents
FuelLevelWarning: thread FuelLevelWarning.imp {
  Compute_Deadline => 8 Ms;
};
connections
GetEstimatedValue_SetLowFuellevelValue_RTDB: data access
GetEstimatedFuelLevelValue ->
FuelLevelWarning.GetEstimatedFuelLevelValue_2;
end AE2_LevelWarning.imp;

thread implementation FuelLevelWarning.imp
properties
Dispatch_Protocol => Periodic;
SEI::Priority => 3;
Compute_Deadline => 8 Ms;
Compute_Execution_Time => 300 Us .. 1 Ms;
Allowed_Processor_BINDING_Class => processor COO7_CPU.imp;
Period => 10 Ms;
end FuelLevelWarning.imp;

process implementation B_Software.imp
subcomponents
SoftwareIN: thread B_Software_IN.imp {
  Compute_Deadline => 8 Ms;
};
SoftwareOUT: thread B_Software_OUT.imp {
  Compute_Deadline => 8 Ms;
};
RTDB: data RTDB_Array.imp;
connections
 ConvertedSensorValue: data port SensorValue_2 ->
SoftwareIN.SensorValue_3 {
  Allowed_Connection_Binding_Class => bus FuelLevelSensorWire.imp;
};
SetSensorConvertedValue: data access RTDB.SendReceiveSharedData ->
SoftwareIN.SetSensorData;
GetCalculatedValue: data access RTDB.SendReceiveSharedData ->
SoftwareOUT.GetFinalValues;
  GetSetAEsValues: data access RTDB.SendReceiveSharedData ->
AEs_Connection;
SendFinalFuelEstimationValue: event data port SoftwareOUT.FinalEstimationValue -> EstimatedFuelValue;
SendFinalFuelLevelWarningValue: event data port SoftwareOUT.FuelLevelWarningValue -> LevelWarningValue;
end B_Software.imp;

system implementation ICL.imp
subcomponents
  TriggerSignals: process TriggerLamps.imp;
  ICL_CPU: processor ICL_CPU.imp;
  ICL_Memory: memory SharedMemory.imp;
connections
  WarningSignalTrigger: data port TriggerSignals.WarningLampTrigger ->
LampSignal;
  FuelLevelPointTrigger: data port TriggerSignals.FuelLevelDisplay ->
FuelLevelDisplaySignal;
  WarningSignal: event data port LowFuelLevelWarningValue ->
TriggerSignals.LampTrigger;
  FuelLevelSignal: event data port EstimatedFuelLevelValue ->
TriggerSignals.FuelLevel;
properties
  Actual_Processor_Binding => reference icl_cpu applies to
TriggerSignals.TriggerLampAndMeter;
end ICL.imp;

system implementation Item_FuelLevelModel.imp
subcomponents
  ECU_COO: system COO.imp;
  ECU_ICL: system ICL.imp;
  Fuel_Level_Sensor: device FuelLevelSensor.imp;
  FuelLevel_Indicator: device FuelLevelIndicator.imp;
  WarningLamp: device Lamp.imp;
  CAN: bus CANbus.imp;
  FuelSensorWire: bus FuelLevelSensorWire.imp;
  IndicatorWire: bus IndicatorWire.imp;
  LampWire: bus LampWire.imp;
connections
  EstimatedFuelLevel_Analog: data port ECU_ICL.FuelLevelDisplaySignal ->
FuelLevel_Indicator.FuelLevelDisplay;
  WarningTrigger: data port ECU_ICL.LampSignal ->
WarningLamp.WarningSignal;
  ReadyLowFuelLevelWarning: event data port ECU_COO.LowFuelLevelWarning ->
  ECU_ICL.LowFuelLevelWarningValue;
  ReadyFuelLevelEstimatedValue: event data port ECU_COO.FuelLevel ->
ECU_ICL.EstimatedFuelLevelValue;
  FuelLevel_AnalogValue: data port Fuel_Level_Sensor.OutValue ->
ECU_COO.SensorValue;
properties
  Actual_Connection_Binding => reference fuelsensorwire applies to
FuelLevel_AnalogValue;
device FuelLevelSensor
  features
    OutValue: out data port;
  end FuelLevelSensor;

device implementation FuelLevelSensor.imp
end FuelLevelSensor.imp;

device Lamp
  features
    WarningSignal: in data port;
  end Lamp;

device implementation Lamp.imp
end Lamp.imp;

device FuelLevelIndicator
  features
    FuelLevelSignal: in data port;
  end FuelLevelIndicator;

device implementation FuelLevelIndicator.imp
end FuelLevelIndicator.imp;

memory SharedMemory
end SharedMemory;

memory implementation SharedMemory.imp
end SharedMemory.imp;

processor CPUs
end CPUs;

processor implementation CPUs.imp
  subcomponents
    CPU_RAM: memory SharedMemory.imp;
  end CPUs.imp;

process TriggerLamps
  features
    WarningLampTrigger: out data port;
    FuelLevelDisplay: out data port;
    WarningLamp: in event data port;
    FuelLevel: in event data port;
  end TriggerLamps;

process implementation TriggerLamps.imp
  subcomponents
    TriggerLampAndMeter: thread Trigger.imp;
  connections
    TriggerWarningLamp: data port TriggerLampAndMeter.ActuateWarningLamp -> WarningLampTrigger;
    FuelLevelPointTriggerSignal: data port
    TriggerLampAndMeter.PointFuelLevel -> FuelLevelDisplay;
WarningTrigger: event data port WarningLamp ->
TriggerLampAndMeter.ShowWarning;
FuelLevelTrigger: event data port FuelLevel ->
TriggerLampAndMeter.ShowFuelLevel;
end TriggerLamps.imp;

thread Trigger
  features
    ActuateWarningLamp: out data port;
    PointFuelLevel: out data port;
    ShowWarning: in event data port;
    ShowFuelLevel: in event data port;
end Trigger;

thread implementation Trigger.imp
  properties
    Dispatch_Protocol => Periodic;
    SEI::Priority => 6;
    Compute_Deadline => 2 Ms;
    Compute_Execution_Time => 1 Ms .. 2 Ms;
    Period => 10 Ms;
    Allowed_Processor_Binding_Class => processor ICL_CPU.imp;
end Trigger.imp;

process COO_Other_AEs
  features
    MemoryReadWrite: requires data access;
end COO_Other_AEs;

thread OtherAEs
  features
    ReadWrite: requires data access;
end OtherAEs;

process implementation COO_Other_AEs.imp
  subcomponents
    OtherAEs: thread OtherAEs.imp {
      Compute_Deadline => 8 Ms;
    };
  connections
    ReadWriteCon: data access MemoryReadWrite -> OtherAEs.ReadWrite;
end COO_Other_AEs.imp;

thread implementation OtherAEs.imp
  properties
    Allowed_Processor_Binding_Class => processor COO7_CPU.imp;
    Dispatch_Protocol => Periodic;
    SEI::Priority => 4;
    Compute_Deadline => 8 Ms;
    Compute_Execution_Time => 2 Ms .. 4 Ms;
    Period => 10 Ms;
end OtherAEs.imp;

processor ICL_CPU
end ICL_CPU;

processor implementation ICL_CPU.imp
  subcomponents
    ICL_RAM: memory ICL_CPU_RAM.imp;
    modes none ;
end ICL_CPU.imp;
processor COO7_CPU
end COO7_CPU;

processor implementation COO7_CPU.imp
subcomponents
  COO7_RAM: memory COO7_CPU_RAM.imp;
             modes none ;
end COO7_CPU.imp;

memory COO7_CPU_RAM
end COO7_CPU_RAM;

memory implementation COO7_CPU_RAM.imp
end COO7_CPU_RAM.imp;

memory ICL_CPU_RAM
end ICL_CPU_RAM;

memory implementation ICL_CPU_RAM.imp
end ICL_CPU_RAM.imp;

bus CANbus
end CANbus;

bus implementation CANbus.imp
end CANbus.imp;

bus FuelLevelSensorWire
end FuelLevelSensorWire;

bus implementation FuelLevelSensorWire.imp
end FuelLevelSensorWire.imp;

bus LampWire
end LampWire;

bus implementation LampWire.imp
end LampWire.imp;

bus IndicatorWire
end IndicatorWire;

bus implementation IndicatorWire.imp
end IndicatorWire.imp;