MODEL-BASED DESIGN OF A ROADSIDE UNIT FOR EMERGENCY AND DISASTER MANAGEMENT

by

Nur Hilal

B.S., Informatics Engineering, Al-Baath University, Syria, 2006

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2020

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Allah the Almighty for giving me the strength, knowledge, ability and opportunity to undertake this research study and to persevere and complete it satisfactorily.

I would like to thank my supervisor Prof. Arda Yurdakul for her guidance, efforts, cooperation and support throughout my years of study and through the process of researching and writing this thesis. She consistently allowed this research to be my own work, but steered me in the right direction whenever she thought I needed it. I also want to acknowledge jury members Prof. Alper Şen from Boğaziçi university, and Assoc. Prof. Rabie Ben Atitallah from Galatasaray university for their valuable comments and feedback.

I gratefully acknowledge the contributions of Yavuz Köroğlu who provided help and shared his ideas and expertise through this thesis work.

I would like to thank Michael Mestre for his help and quick response during translation of the abstract to Turkish.

I want also to thank my fellow lab colleagues, Görker Alp Malazgirt, Emre Bilgili, Semih Erşahin, Ceyhun Onur, Mahmut Karaca, and my friend Gamze Ege Kaya for their inspiration, support and advice especially towards the end of this thesis.

Finally, I must express my very profound gratitude to my beloved parents, Walid Hilal and Fadiah Alhaswani, and to my brother M. Anas Helal for their undying support and continues encouragements at all times. Without them, I would not have made it through my masters degree.

ABSTRACT

MODEL-BASED DESIGN OF A ROADSIDE UNIT FOR EMERGENCY AND DISASTER MANAGEMENT

Every year, a massive number of deaths happen because of traffic accidents. In order to increase the traffic victim's survival rates, it is important to reduce the arrival time of trauma intervention teams to accidents' sites. Automatic incident detection provides faster incident reporting, in which it decreases the delay of arrival time of first responders.

In this thesis, we propose a system where traffic incidents can be detected, verified and reported using multiple detection mechanisms and communication technologies to provide faster response and allow for first responders to arrive as quickly as possible. Our proposed system contains a Roadside Unit (RSU), which is responsible for incident detection using two automatic incident detection algorithms depending on traffic parameters collected by Inductive Loops sensors and communication technology called VANET. The proposed RSU, also, listens and receives two incident reporting signals sent by eCall and WreckWatch solutions. This thesis provides the internal architecture of the proposed RSU using Model-Based Engineering concept, where the RSU is modeled in Architecture Analysis and Design Language (AADL). We provide an AADL model that captures the internal structure of the RSU, its components and their interactions. We provide experiments on the model's tasks execution using gem5 simulator depending on different configurations. We used gem5 simulation results for scheduling properties of the AADL model in order to present scheduling tests and latency analysis. In addition, we present scheduling simulations using AADL Inspector. Test results show that our RSU model is schedulable with low processor utilization factors, and provides incident detection and reporting in under three minutes.

ÖZET

ACİL DURUM VE AFET YÖNETİMİ İÇİN YOL KENARI ÜNİTESİNİN MODEL TABANLI TASARIMI

Her yıl, trafik kazaları yüzünden çok sayıda ölüm meydana gelmektedir. Trafik kazazedelerinin hayatta kalma oranlarını arttırmak için, travma müdahale ekiplerinin kaza alanına olabildiğince çabuk varmalarını sağlamak önemlidir. Otomatik olay tespiti, daha hızlı olay bildirme olanağı sağlar ve böylece kazanın gerçekleşme anı ile ilk müdahalenin olay yerine ulaşması arasındaki süreyi azaltır.

Bu çalışmada, çeşitli algılama mekanizmaları ve iletişim teknolojilerinden faydalanarak, trafik olaylarının algılanması, doğrulanması ve bildirilmesine olanak sağlayan bir sistem önerilmektedir. Önerilen sistem, iki otomatik olay tespit algoritması kullanan ve olay algılanmasından sorumlu olan bir RSU (”Roadside Unit” - Yol Kenarı Ünitesi) içermektedir. Algoritmalardan biri endüktif döngü sensörleriyle toplanmış trafik parametrelerine, diğeri VANET (”Vehicle Ad-hoc Networks” - Tasarsız Taşıt Ağları) adı verilen bir iletişim teknolojisine dayanmaktadır. Önerilen RSU, ayrıca eCall ile WreckWatch çözümlerince gönderilen iki olay bildirme sinyalini dinler ve alır. Bu çalışmada önerilen RSU, AADL (”Architecture Analysis and Design Language”) dili ile modellenmiştir. Modelin iç yapısı, bileşenleri ile onların etkileşimlerini kapsayan bir AADL modeli tanımlanmaktadır. Çeşitli işlemci yapıları için gem5 isimli simülatör ile modelin görev yürütümü üzerine yapılan deneyler sunulmaktadır. AADL modelinin zamanlama özelliklerini belirlemek ve nihayetinde zamanlama testleri ile gecikme analiz raporları sunmak amacıyla, gem5 benzetim sonuçları kullanılmaktadır. Ayrıca, AADL Inspector ile yapılan zamanlama benzetimi gösterilmektedir. Test sonuçları, RSU modelimizin düşük işlemci kullanım oranı ile programlanabilir olduğunu ve üç dakikadan kısa bir sürede olay tespiti ve raporlaması sağladığını göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES x
LIST OF SYMBOLS xi
LIST OF ACRONYMS/ABBREVIATIONS xii
1. INTRODUCTION 1
2. RELATED WORK 5
2.1. Incident detection methods and sensors 6
2.1.1. eCall 7
2.1.2. WreckWatch 8
2.1.3. Traffic Engineering and Road Sensors 8
2.1.4. Vehicle Ad-Hoc Networks 13
2.2. Incident Classification 17
2.3. Model-Based Engineering 18
2.3.1. Architecture Analysis and Design Language (AADL) 19
2.3.1.1. AADL Language Overview 19
2.3.1.2. AADL Tools 21
3. MODEL-BASED DESIGN OF A ROADSIDE UNIT FOR EMERGENCY AND DISASTER MANAGEMENT 23
3.1. Proposed System 23
3.2. Roadside Unit (RSU) Model 24
3.2.1. Input unit 27
3.2.2. Detection Unit 28
3.2.3. Processing Unit 30
3.2.4. Output Unit 33
3.2.5. Devices 34

3.2.6. Complete Functional Model 35
3.2.7. Platform 37
4. EXPERIMENTS AND RESULTS 40
4.1. Tasks' code design and implementation in C++ 40
4.2. Tasks' simulations on gem5 41
4.3. RSU Model's schedulability analysis in OSATE 45
4.4. RSU Model's flow latency analysis in OSATE 46
4.5. Scheduling simulations in AADL Inspector 48
5. CONCLUSION AND FUTURE WORK 51
REFERENCES 53
APPENDIX A: AADL MODEL SCRIPTS 58
A.1. Software Components 58
A.1.1. Input Unit 58
A.1.2. Detection Unit 61
A.1.3. Processing Unit 63
A.1.4. Output Unit 65
A.1.5. Data Types 67
A.2. Hardware Components 74
A.2.1. Devices 74
A.2.2. Platform 76
A.3. RSU system integration 79

LIST OF FIGURES

Figure 2.1. Explanatory figure for the changing lane parameters, inspired from [31]. 16

Figure 3.1. The proposed system. 24

Figure 3.2. Input Unit's AADL model. 27

Figure 3.3. Detection Unit's AADL model. 29

Figure 3.4. Processing Unit's AADL model. 31

Figure 3.5. Th_Verification task execution flow. 32

Figure 3.6. Output Unit's AADL model. 34

Figure 3.7. The devices used in RSU. 34

Figure 3.8. The complete functional AADL model. 36

Figure 3.9. Deployment options. 37

Figure 3.10. System integration. 38

Figure 4.1. Simulation command example of ecallpoll thread on x86 processor. 42

Figure 4.2. System implementation instantiating. 45

Figure 4.3. System variations' schedulability test results. 46

Figure 4.4. End-to-end flow in a system implementation. 47

Figure 4.5. Static scheduling simulation by AADL Inspector. 49

Figure 4.6. Dynamic scheduling simulation by AADL Inspector. 50

LIST OF TABLES

Table 2.1. Input and output variables stated in [23]. 11

Table 2.2. AADL components categories. 20

Table 4.1. The first option based on NXP RSU. 41

Table 4.2. The second option with 4 variations based on Siemens RSU. 42

Table 4.3. gem5 results for the proposed option 2_1 on X86 environment. . . . 42

Table 4.4. gem5 results for the proposed option 2_2 on X86 environment. . . . 43

Table 4.5. gem5 results for the proposed option 1 on ARM environment. . . . 43

Table 4.6. gem5 results for the proposed option 2_1 on ARM environment. . . 44

Table 4.7. gem5 results for the proposed option 2_2 on ARM environment. . . 44

Table 4.8. Minimum latency of all end-to-end flows under all platform variations. 47

Table 4.9. Maximum latency of all end-to-end flows under all platform variations. 48

LIST OF SYMBOLS

$\lvert bc \rvert$	Lane width
$\lvert ab \rvert$	Distance between vehicle A and vehicle B
Θ	Angel of which the vehicle change lanes

LIST OF ACRONYMS/ABBREVIATIONS

AADL	Architecture and Analyses Design Language
AID	Automatic Incident Detection
BAnnex	Behavior Annex standard for AADL
CCH	Common Control Channel
CLD	Changing Lane Distance
CLS	Changing Lane Speed
CPS	Cyber Physical System
DR	Detection Rate
DSRC	Dedicated Short Range Communication
Eannex	Error Model Annex standard for AADL
eCall	Emergency Call
EMYNOS	Next Generation Emergency Communications
ESA	European Space Agency
EU	European Union
EV	Emergency Vehicle
FAR	False Alarm Rate
FD	Fundamental Diagram
FIS	Fuzzy Inference System
I2I	Infrastructure to Infrastructure
IDP	Intelligent Data Processing
ILD	Inductive Loop Detectors
IoT	Internet of Things
ITMS	Intelligent Traffic Management Systems
ITS	Intelligent Transportation Systems
IVS	In-Vehicle Systems
MARTE	Modeling and Analysis of Real Time and Embedded systems
MBE	Model-Based Engineering
MF	Membership Function

ML	Machine Learning
MSD	Minimum Set of Data
MTTD	Mean Time To Detect
OBU	On-Board Units
OMG	Object Management Group
OSATE	Open Source AADL Tool Environment
PSAP	Public Safety Answering Point
PVD	Probe Vehicles Data
RFID	Radio Frequency Identification
RSU	Roadside Unit
SAE	Society of Automotive Engineers
SCH	Service Channels
SDLC	System Development Life Cycle
SysML	Systems Modelling Language
UML	Unified Modelling Language
V2I	Vehicle to Infrastructure
VANET	Vehicl Ad-hoc Network
VIP	Video Image Processor
WAVE	Wireless Access in a Vehicular Environment
WHO	World Health Organization

1. INTRODUCTION

Road traffic accidents are one of the leading causes of death around the world. According to World Health Organization (WHO), on 2018th global status report on road safety [1], 1.35 million deaths happen each year because of road accidents. It is number one cause of death for children and young adults of 5 - 29 years old. In order to decrease casualties number, drastic measures need to be applied, such as better incident detection and faster emergency responders dispatch. To be able to increase the survival rates of trauma victims in road accidents, they need to get rescued, transferred and undergo the appropriate medical procedures within one hour since the accident has taken place. This specified time between a trauma occurrence and a medical intervention of a special trauma team is known as the Golden Hour [2]. Our major interest is that the accident detection and reporting should not take more than 5 minutes, and the arrival of the first responders should not exceed 15 minutes, as illustrated in [3].

With the upcoming evolution of smart cities comes a new emerging concept of intelligent traffic management systems, where emergency management issue is one of the hottest topics in the literature. Intelligent Transportation Systems (ITS) are advanced applications which intend to offer inventive services that help the users to drive safely and be better informed of upcoming traffic issues to make safer and more coordinated transport networks [4]. Many research papers have considered how the architecture of ITS should be to serve as many smart traffic services and applications as possible. In this thesis, we focused on the emergency management section. Niar *et al.* (2014) proposed a multi-layer reconfigurable disaster and emergency management architecture in ITS [5]. Their architecture consists of three layers: Sensor Processing layer, Intelligent Data Processing layer, and Cloud Processing layer. The Sensor Processing evolves around data collection from various sensors on the road or in vehicles, or even from smart-phones. The Intelligent Data Processing layer consists of small yet computationally powerful units that are responsible for processing the incoming data

from the sensors and provides multiple services such as traffic flow control, local routing, and incident management. The Cloud processing layer serves the global decision making, where it takes the data from the previous layer, and provides global traffic management and surveillance systems. The aim of the study in this paper is to analyze the collected data from the traffic network to achieve a plan for road evacuation or dispatching Emergency Vehicles (EV) like fire trucks, car towers, and ambulances to attend to the incident site and help the victims. Looking at this architecture, we chose to take a deeper look into those units in the Intelligent Data Processing layer. Such units are now called Road Side Units (RSU), where they receive multiple and different inputs, make initial processing and provide services within its range.

There are few commercial RSUs on the market [6] [7], each provides different services and connectivity to its consumers. However, none has claimed to provide incident detection or emergency management. In our study, we focus on incident detection using multiple detection algorithms and emergency signals that depends on different types of sensor technologies. In addition, our study proposes the help of the cloud layer to provide incident severity classification, emergency routing plans, and incident clearance confirmation.

A well-designed incident management system has to start with a detection mechanism to detect the existence of an emergency or a disaster once it happens, followed by verification of the incident to minimize false alarms. Then, this system has to quickly classify the severity level of the incident before it can decide on an appropriate response for this particular incident. One of the prime detection mechanisms is to use Emergency Call (eCall). eCall is a European initiative with the aim of bringing rapid emergency assistance when road accidents happen [8]. eCall depends on In-Vehicle Systems (IVS) that is embedded in modern vehicles where it detects accidents based on acceleration and crash sensors in the vehicles, then automatically contacts emergency parties for rescue. Other detection mechanisms are based on road sensors, such as Inductive Loop Detectors (ILD), magnetic sensors, Video Image Processor (VIP), ultrasonic sensors, infrared sensors and radars [9]. Smartphones applications can also

be used to produce real-time data for incidents detection [10]. Once an incident is detected by one of the mechanisms, it must be confirmed to reduce the possibility of false alarms. Then, an incident severity level classification should be performed in order to decide on the appropriate response. Sending a specific EV and choosing the right routing scenario is very important in saving time and resources. The classification of the incident's severity level depends on the collected data from the incident site that relates to the size of the incident, how many vehicles involved, how many victims, place of the accident, and whether it contains hazards of any type. Such sophisticated computation using machine learning techniques can be exhausting for the local RSU to provide, hence, proposing cloud computing to mitigate execution overhead. One of the key points for EV fast arrival is the routing algorithms that include best route planning and clearing the road to/from the incident site to the nearest medical facility if needed. Routing algorithms depend heavily on controlling the traffic light signals to prioritize the EV. To guarantee the best route and traffic light signal management, this process is designed to be done in the cloud layer, where the cloud can also delegate routing instructions to all RSUs implanted on the route from/to accident location. In addition, we suggest using VANET messages to broadcast to vehicles to clear a specified lane for the EV to pass without delay.

Creating such complex systems takes a significant amount of time and effort. Designing these systems must be done in a cautious and a professional way where it could be carefully tested and monitored before it can be deployed. The best way to do this is by using Model-Based Engineering (MBE). MBE is a methodology of system engineering that focuses on visual representations of the system aspects using modeling principles throughout the System Development Life Cycle (SDLC) [11]. Models are created to analyze the system and its parts, and provide some abstract perspective in order to find out whether the system is correct or not, in other words, if the system fulfill its requirements. Using MBE provides early identification of requirements' issues, higher system design integrity and early detection of system design errors. Hence, MBE reduces time consumption and costly modifications before reaching the integration and testing level. There are many modelling languages that differ in characteristics and

syntax to define and describe these complex systems' architectures and behaviors [12]. In this thesis, we are using Architecture Analysis and Design Language (AADL) [13] to model our RSU system and provide some inside look into its architecture and behavior. To be able to work with AADL, we used a stand-alone software called OSATE [14], where we developed our system models and specifications and we were able to test its schedulability and latency overhead using analysis tools embedded in this software. We also used AADL Inspector software to provide scheduling simulations for our model.

In this thesis:

- We propose an intelligent emergency management system that includes a Road-side Unit (RSU) and a cloud computing approach.
- We provide a novel approach of an RSU architecture that provides incident detection using multiple technologies and communication services.
- We propose a verification process to confirm detected incidents to minimize false alarms.

In this thesis, we start with the related work in Chapter 2, where we discuss some commercial RSUs and their services, and provide some insights on other research papers that work around intelligent traffic and emergency management systems. In addition, we provide the background information and studies, on which this thesis utilizes, in detail. In Chapter 3, we explain our methodology and system proposal with insightful details about the internal architecture of our RSU. We walk through the simulation processes and experimental work in Chapter 4. In the final chapter, we conclude our study and discuss future work.

2. RELATED WORK

In the literature, there are numerous studies discuss Roadside Units (RSUs) and intelligent traffic management systems (ITMS). To the best of our knowledge, none of these studies uses multiple incident detection techniques and different communications protocols to provide traffic incident detection and reporting services. The rest of this chapter will provide an overview about several research papers that studies roadside units services, and a couple of on the market commercial roadside units.

One of the studies involving roadside units services is presented in [15]. They discuss a concept of road side servers that work in different road systems related to telematics services. In [15], two models are proposed in signalized intersections and smart highways. These models provide driver's assistance and vehicular safety for vehicles that are located within the service range of these roadside servers. Their work provides advisory safety instructions and vehicle incident prevention techniques.

Another study discusses using drones as multipurpose mobile RSUs [16]. They propose to combine multiple connection solutions into swarm of autonomous drones to assist first responders in terms of connectivity in blind areas, in an attempt to mitigate the large cost of stationary RSU installations. However, their work is mainly about connections and network security, it does not provide incident detection and verification.

In [17], the authors propose a model of intelligent traffic control system for ambulance clearance, stolen vehicle detection and traffic congestion control. They propose using Radio Frequency Identification (RFID) technology, ZigBee modules and automatic traffic signal control systems. Every vehicle must be equipped with an RFID tag, where it can be detected by an RFID reader implanted on the road. This RFID reader counts the number of vehicles passing through the road in a specific time interval, and based on the traffic volume, they set the green light duration through communications

with the automatic signal control system. Also, the RFID reader compares the RFID tags of the passing vehicles with a list of stolen tags. If one is a match, a procedure will take place to stop the car and inform authorities. Every emergency vehicle is equipped with ZigBee transmitter module. Hence, when it approaches a traffic junction, it communicates with the ZigBee receiver module in the automatic signal system to turn on the green light and passes through.

In addition to the previous studies, there are commercial RSU devices available on the market. One of these products belong to Siemens [6]. Siemens provides an RSU mainly used for connectivity and crash prevention. It provides wireless communication as vehicle-to-infrastructure (V2I) connectivity, local Wi-Fi hotspot for travel applications and smart devices, in addition to optional backhaul for LTE cellular network. Another product is manufactured by NXP [7]. NXP provides an intelligent RSU that acts as a smart traffic light, where this device uses multiple sensing techniques such as cameras, radars, and V2I communications. Its main purpose is to optimize traffic flow through traffic light control. In addition, it provides communication solutions such as WiFi and cellular hotspot connectivity.

As we sailed through the previous studies that provide partial solutions for traffic flow control, we felt compelled to create a solution that provides multi-sensory approach for incident detection and reporting. Once an accident happens, it is a time-critical issue to provide incident detection and response to help increase trauma victims' survival rates.

In the following sections of this chapter, we will discuss the literature behind our work regarding incident detection methods and systems' modelling.

2.1. Incident detection methods and sensors

Incident detection is one of the leading topics in Intelligent Transportation Systems (ITS). There are many research and projects moving forward into finding the best

and fastest solutions to mitigate traffic incidents and decrease its casualties. Incident detection is defined as the process of distinguishing the time and location of the incident and its possible cause and type. In this chapter we are going to talk about all the technologies used in incident detection purpose and that we are using in our proposed system from a literature perspective.

2.1.1. eCall

One of the most important incident detection methods is eCall [18]. eCall is an emergency system that provides an automatic incident detection of road accidents and initiates a telecommunication service with the related Public Safety Answering Point (PSAP). eCall had first started as an initiative in the European Union (EU). However by mid 2014, the decision to deploy the eCall service was announced by The European Parliament and the European Council. eCall system consists of two parts, an In-Vehicle System (IVS) and an eCall data modem installed at all PSAPs. In the event of a traffic accident, the IVS unit detects the accident using some built-in crash sensors within the vehicle, and establish a 112-voice call automatically (or manually by the driver) to the nearest PSAP center through cellular networks. Even if the driver is unable to talk, the IVS will send a Minimum Set of Data (MSD) to the PSAP center to notify authorities with the accident information such as the vehicle location, vehicle identification, number of passengers and other related information. IVS units are now a mandatory equipment for all new car models since the end of March 2018, while the infrastructure of eCall modems at PSAPs are already deployed in all European states by the end of 2017 [19]. Furthermore, a new project called Next Generation Emergency Communications (EMYNOS) is initiated to extend eCall communication technology to include mobile devices, photos and video streaming. Moreover, it provides the ability to prevent hoax calls, offer special support for the disabled and obtain accurate user/device position [20].

However, eCall's IVS units are only implanted in the new modern cars, and adding IVS to old cars might be expensive to a big portion of the population. In the light of

these limitations, we wanted to add another method of incident detection that might be able to cover the lack of pre-installed detection mechanism in old vehicles. In the next section, we talk about incident detection using smartphones.

2.1.2. WreckWatch

In [10], the authors propose a different approach on incident detection by providing a study on detecting traffic accidents using smartphones. The availability and low cost of smartphones against other means of incident detection and reporting mechanisms makes them a favorable alternative solution. In addition to their wide-spread usage among users, smartphones travel with their owners which provide incident detection whether or not the vehicle is equipped with an IVS unit. The authors of [10] propose a smartphone-based prototype of a client/server application called "WreckWatch". WreckWatch consists of two parts, the WreckWatch client and the WreckWatch server. The WreckWatch client is an application installed on the user's smartphone, where it exploits the built-in sensors such as GPS and accelerometer to detect a traffic accident when it happens. It also uses other sensors like the microphone to detect accident sounds, which is one of the main mechanisms to prevent false alarms such as dropping the phone by mistake. When an accident is detected, the WreckWatch client will relay the accident information to the WreckWatch server through HTTP post operations. The WreckWatch server provides data gathering and communication services for first responders, family and friends.

2.1.3. Traffic Engineering and Road Sensors

Incident detection can be achieved using traffic engineering. Traffic engineering, by definition, is the science that measures traffic metrics to design and facilitate traffic flow and transportation systems. Traffic engineering sensors are widely used, and they are many in types and performance levels. To name a few: ultrasonic sensors, radars, inductive loops, cameras and infrared sensors. Road sensors are mainly used in two areas: roads intersections and highways, where they collect different traffic metrics such

as volume, density, speed and occupancy [9]. Using these metrics improves traffic flow management and helps in creating better traffic systems. Due to their wide usage and availability, traffic sensors have become a very important factor in incident detection methods.

Incident detection algorithms can be generally divided into two categories: manual and automatic. Manual incident detection involves a human presence to be able to recognize and notify of an incident occurrence. The automatic incident detection algorithms (AID) use real time traffic data to automatically trigger an incident alarm and find its size and location. Most of the current research on incident detection is about AID algorithms. An AID algorithm performance is mainly measured by three variables: Detection Rate (DR), Mean Time To Detect (MTTD), and False Alarm Rate (FAR) [9]. Detection rate is a percentage of the number of detected incidents to the number of incidents that had occurred. MTTD is the average time an algorithm takes to detect incidents, while the FAR is the ratio of the false alarms to the total number of alarms issued. In addition, any algorithm depends on the accuracy of the measured data, a well-chosen sensor type in the right area is key to provide better performance. Road sensors are either embedded in the pavement, off the road or above the traffic flow. Therefore, they are affected by environmental conditions such as fog or rain. Along the comparisons made in [9], between multiple sensor technologies, we find inductive loop sensors withstand most weather conditions due to their instalment under the pavement.

An inductive loop sensor is basically a single or multiple wire loops embedded under the pavement and connected to a controller cabinet using extension cable [21]. These main parts make what is called an Inductive Loop Detector (ILD) system. ILDs installation methods and techniques vary between traffic departments. Usually, the ILDs are installed prior to the pavement installation. However, if the pavement is already in place, the wire loops are installed by performing a saw-cut in the pavement in the shape of the wire loop to embed the wires under the lane surface area and connecting it to the control cabinet nearby, that is usually set on the curb. The wire

loops are protected with conduits and the saw-cut is covered with sealant [22]. The controller powers the wire loops which creates a magnetic field in the loop area that resonates at a specific frequency. The controller registers that constant frequency when no vehicle is present and establishes it as a comparison factor. When a vehicle travels over the loop area or stops at it, it causes the magnetic field to change resonance to a higher frequency. The controller senses the presence of the vehicle and forces a relay -that is normally open- to close. Once the vehicle leaves the loop area, the relay reopens and the frequency returns to its established level.

ILDs are used in traffic engineering to provide the main traffic variables such as lane occupancy data and vehicle count, in addition to speed data generation if two loop sensors were used in tandem. ILDs were used in the early incident detection algorithms because of their availability and low cost. One of the research papers that proposed incident detection algorithm based on inductive loop detectors is presented in [23].

AID algorithms differ in their approaches, such as pattern recognition, statistical theory and artificial intelligence [24]. In [23], fuzzy logic approach is used for incident detection based on inductive loop detectors. Fuzzy logic comes from fuzzy set theory, where a fuzzy set is a set that its elements have membership degrees. Fuzzy logic depends on finding an approximate rather than a precise pattern [25], which makes it a promising approach because of its ability to deal with the uncertainty of traffic conditions and assessments. The authors of [23] propose an approach of two Fuzzy Inference Systems (FIS) which study traffic data collected by two loop detector stations. One of the two FIS is placed upstream (FIS1) of the incident location and the second is at the downstream (FIS2). The proposed system compares calibration data of normal traffic with current collected data to find differences, these differences are computed on the Fundamental Diagram (FD) of traffic flow. When these differences in the FD between normal traffic conditions and observed traffic conditions of the same time of the day are “low”, then the system is in “normal state”, or else it is in “abnormal state”.

The proposed system in [23] is based on the collected traffic data such as density, speed and volume that comes from the installed ILD stations. The input variables, shown in Table 2.1, are considered by both FIS1 and FIS2 in absolute or relative matter (difference between usual and current conditions). The output variables STATE1 and STATE2 serve as the state of an upstream and downstream, respectively, of a road section according to traffic conditions.

Table 2.1. Input and output variables stated in [23].

Variable	**Type**	**Description**
DFR-K L1	Input variable for FIS1	Distance between normal and current traffic condition in Flow/Density plane for lane 1
DFR-SMS L1	Input variable for FIS1	Distance between normal and current traffic condition in Flow/Space Mean Speed plane for lane 1
DFR-K L2	Input variable for FIS1	Distance between normal and current traffic condition in Flow/Density plane for lane 2
DFR-SMS L2	Input variable for FIS1	Distance between normal and current traffic condition in Flow/Space Mean Speed plane for lane 2
STATE1	Output variable of FIS1	State of the system
K L1	Input variable for FIS2	Observed Density in lane 1
SMS L1	Input variable for FIS 2	Observed Spatial Mean Speed in lane 1
K L2	Input variable for FIS 2	Observed Density in lane 2
SMS L2	Input variable for FIS 2	Observed Spatial Mean Speed in lane 2
STATE2	Output variable of FIS 2	State of the system

For simulation purposes, the proposed algorithm in [23] considered a simple straight road segment. The road has one direction and two lanes with the length of 10 km. To record traffic flow data, virtual detector stations were embedded into the road at 100-meter distance from one another. Each detector station has two loops, one for each lane, where each loop represents the upstream of the next section of the road and downstream for the previous section of the road.

What we understand from this algorithm is that their proposed system depends on collecting traffic data for an interval of one minute, then compares some of the data on the fundamental diagram to find the differences between current and usual

conditions for the FIS1, where they compare the other collected data with calibration data in absolute matter for the FIS2. Both of comparisons data are then fuzzified by using membership function (MF) to find their membership degrees. According to their membership degrees we can find if the input variable belongs to one fuzzy set. According to fuzzy logic, they created some fuzzy rules in order for the inference system to produce a conclusion. In this algorithm, they used Mamdani's product-sum inference system [26], where its fuzzy rules are a collection of If-Then statements that define how the FIS should make a decision based on classifying inputs or controlling an output. An example of these statements from this algorithm is written below:

$$[(\text{DFR-K L1 is Small}) \wedge (\text{DFR-K L2 is Medium}) \wedge (\text{DFR-SMS L1 is Small}) \wedge \\ (\text{DFR-SMS L2 is Small})] \implies (\text{STATE1 is Abnormal})\ [23]$$

$$[(\text{SMS L1 is Large}) \wedge (\text{SMS L2 is Medium}) \wedge (\text{K L1 is Medium}) \wedge (\text{K L2 is Small})] \\ \implies (\text{STATE2 is Normal})\ [23]$$

Each statement depends on the four input variables they used to produce the output. Of course, there are more fuzzy rules, about 81 rules per inference engine, embedded in the inference system other than these two quoted above. For the inference system to work without errors, the fuzzy rules should cover most, if not all, possible combinations of input variables. After arriving to conclusions by the inference engine based on all fuzzy rules created, if one of the output variables produce an abnormal state for a certain amount of time (i.e. 2 consecutive minutes), the algorithm then prompts an incident alarm [23].

This algorithm's performance is discussed in detail in [23]. They proposed three options for simulations, where the loop detectors were separated by 100m, 500m, and 1000m in between. The algorithm's DR is as high as over 80%, except for low traffic volume conditions (2000 vehicle/hour) with 1000m distance between loops. The algorithm's FAR is less than 1% in all tested cases, and the MTTD increases as the distance between loops increases from a minimum of 1.10 to a little less than five minutes.

2.1.4. Vehicle Ad-Hoc Networks

In addition to traffic road sensors, many other approaches are being studied and developed with the aim of providing better transportation and traffic systems. Traffic engineering system is evolving with the concept of Internet of Things (IoT) being introduced in the recent studies. With IoT, we will soon see smart cities with smart homes, autonomous vehicles and Intelligent Transportation Systems (ITS). ITS is a set of services and applications provided to vehicles and passengers to improve mobility, comfort, quality and safety along the roads [27]. In order for ITS to distribute such services, it needs to depend on reliable communications and networks to reach all consumers in a short amount of time, hence the surface of Vehicular Ad-Hoc NETworks (VANET). VANET is a wireless network connects moving or stationary vehicles together to exchange information and safety messages. VANET also covers the connection with roadside smart devices providing important traffic flow information, routing and incident prevention. The previous connections are known as Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) respectively, in addition to Infrastructure-to-Infrastructure (I2I) communications [27]. The stationary infrastructure plays a key role in providing consistent and full utilization of VANET and its large variety of safety, non-safety and entertainment applications. In order to be connected to VANETs, vehicles need to be equipped with special wireless communication devices known as On Board Units (OBUs). An OBU device is responsible for connecting the vehicle to other vehicles and the infrastructure (basically V2V and V2I communications) to exchange data and use the available services. As the wheel of development is running fast for IoT and intelligent traffic, there has been many standards developed to provide quick and reliable access to the vehicular network, some of these standards are: Cellular Networks, WAVE, DSRC, and WiMAX.

Dedicated Short Range Communication (DSRC) is a medium range wireless communication service based on the 802.11p standards, which are derived from the 802.11a standards [28]. DSRC is mainly used for V2V and V2I communication and data exchange due to its low communication latency and high data transfer capability. These

communications may include road conditions, traffic information, safety messages, accident information, automated toll payment and so on. However, exchanging these messages is subject to 802.11p specifications, where safety and control messages are transferred during Common Control Channel (CCH) interval. The rest of the non-safety and other services messages are transferred through one of the available Service Channels (SCH) [28]. Wireless Access in a Vehicular Environment (WAVE) is another standard that combined IEEE1609 with DSRC 802.11p protocol stack to provide communication model, security mechanisms and physical access for wireless connection to basic devices such as OBU, RSU, and WAVE interface in the vehicular environment. According to the IEEE1609 standards, for messages exchange on the VANET connections, the channel time is divided into 100 ms synchronization intervals. These intervals consist of a CCH interval where all vehicles' OBUs tune into the CCH frequency for receiving control messages, and an SCH interval where the OBUs optionally switch to in order to get a service or deliver a message.

Since the vehicles are basically talking to each other, and to the Infrastructure, this brings the next question: what are the standard formats for these messages? Fortunately, the SAE provides a message set with its data frames and data elements that is created for DSRC/WAVE based applications [29]. SAE J2735 DSRC message list contains different messages types and frames that provides a variety of options for VANET applications to exploit. The SAE J2735 listed in [29] is just a brief summary of some of the messages, while the complete list is available for purchase on SAE standards website [30]. The two most important DSRC messages are probeDataManagemet and probeVehicleData, where the first is the control message sent by the RSU to the vehicles requesting specific probe data while the latter is the reply message sent by the OBU to the RSU that contains collected probe data that was requested previously.

Now that the ITS is also providing us with traffic data coming from moving vehicles, many researchers proposed incident detection based on VANET communications and messages. One of these incident detection algorithms is presented in [31]. They propose AID techniques that take collected traffic data from passing vehicles to an-

alyze traffic flow and find anomalies related with traffic incidents. They depend on RSU infrastructure installed with 1 km between one RSU and another, and unlike the ILD, the RSU does not need to connect with other RSUs to collect and analyze traffic data. Based on the previous thought, they assume that with enough vehicles equipped with OBUs to exchange specific traffic data with the RSU in their range, the RSU will be able to spot anomalies in the traffic flow and recognize the incident as fast as possible. For collecting enough traffic variables, they calculated the probabilistic data for the vehicles to be able to connect and share their data successfully in the shortest amount of time while they are travelling through the RSU range. They found that for the traffic data aggregation to be meaningful, a minimum number of 24 vehicles have to submit their related traffic data will suffice. Depending on the traffic flow, it might take a few minutes to collect meaningful amount of data. They calculated that in a sparse traffic flow of 720 vehicles/hour, it will take 2 minutes to aggregate the right amount of data for incident detection. The proposed AID algorithms in [31] depend on the lane changing information that the passing vehicles share with the RSU. The proposed algorithms consider that vehicles' speed and travelled distance when changing lanes are significantly impacted in the case of accident existence. The idea behind their algorithms for the RSU is to aggregate all traffic data related with passing vehicles changing lanes, and constantly monitor the average values of these traffic variables. Then, the RSU will be able to establish belief measurements of the existence of a traffic incident, where an incident alarm will be issued if these belief measurements exceed a pre-established threshold. The authors in [31] propose two algorithms that depend on two different traffic parameters to analyze and recognize the anomalies related to a traffic incident; these two algorithms are named: Changing Lane Distance (CLD) Method, and Changing Lane Speed (CLS) Method. The CLD method calculates the required distance for a vehicle to change lanes. Usually, changing lanes within a short distance is related to traffic incidents due to the lane being blocked or congested. The CLD method depends on the average distance and average time that takes for a vehicle to change lanes to specify whether an incident exists or not. The following figure shows the required parameters that are associated with changing lanes. Figure 2.1 considers two vehicles A and B. Assuming B is involved in a traffic accident, A is changing lanes

to pass Vehicle B. According to previous notation, the distance between A and B right before A changes lanes should be shorter than a specific threshold to increase the belief of a traffic accident.

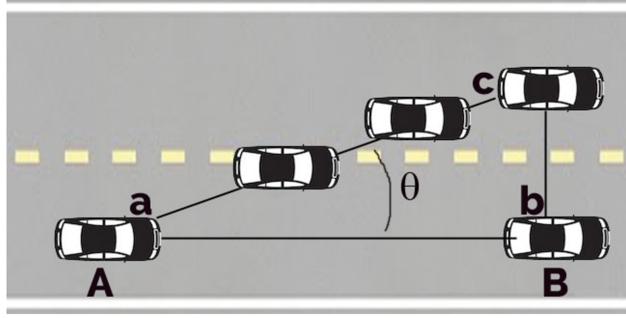

Figure 2.1. Explanatory figure for the changing lane parameters, inspired from [31].

From the figure we note that when A changed the lane, it created an imaginary right triangle, where we can calculate the distance between A and B depending on the distance between two lanes $|bc|$ (which is typically 3.5 meters) and the angle θ of lane changing that vehicle A shares with the RSU. this calculation is done using the following formula of right triangle:

$$tan\theta = \frac{|bc|}{|ab|} \tag{2.1}$$

The CLD algorithm starts by filtering the received data from passing vehicles. Then, it calculates the distance and time needed by all vehicles (that shared their information with the RSU) to change their lanes. Then, it calculates the average distance and time of all the previous values, where it is compared to a predefined critical region. If the average distance and time fall into that critical region, then a belief value is increased. This algorithm is repeated periodically by the RSU. If the belief value exceeds a specific threshold, the algorithm will prompt an incident alert. In a similar manner, the CLS method processes the received data to find anomalies, but it depends on the vehicles' speed rather than the distance. It is noted that when a traffic accident

is present, the arriving vehicles tend to slow down during changing lanes for safety concerns. In non-accident cases, the average vehicles' speed changes in an average time that is shorter than the average time in accident cases. With that notation, they specify the critical region of average speed and average time to recognize the possibility of a traffic incident existence. As the first algorithm, a belief will be increased, and when the belief exceeds a predefined threshold, then an incident alert will be issued. Upon experiments and simulations, performance comparisons between these algorithms and other AID algorithms are presented in [31]. The performance comparisons show better results in incident detection rates, especially in sparse traffic cases when little information is available.

2.2. Incident Classification

So far, we explored many technologies and methods for incident detection, because it is a very time-sensitive issue for an incident to be discovered and reported as fast as possible to save lives. However, another concept is as important as incident detection, incident classification. Incident classification is the process of categorizing an incident condition to a severity level. Incident classification is a necessary process in which it determines what types of emergency vehicles are needed and how many, in addition to prioritization decisions. Normally, an incident is classified manually by an on-site officer or traffic operators where they might need to check CCTV cameras to specify the severity of the incident. Manual classification consume time between taking decisions and notifying related authorities, which motivate us to look for an automatic approach. Research by Nguyen, Cai and Chen [32] discusses automatic incident classification using Machine Learning (ML) techniques. They use real official data collected and logged by a transport management center in Sydney, Australia. They used four types of machine learning methods: Naïve Bayes, k-nearest neighbour, Support vector machines and decision tree. They applied these methods on training data set that covers 4 years period from 2011 to 2014 of incidents logs. An incident severity level depends on multiple factors such as location, time of day, number of affected lanes, whether there are injuries and so on. They applied the four methods on the train-

ing data and compared the results, among the four, they noted that decision tree has achieved better performance than the rest of ML methods.

2.3. Model-Based Engineering

As the embedded systems is developing over the years, they become more and more reliant on commercial hardware and packed with software applications capabilities. In addition, Cyber Physical Systems (CPS) are rapidly taking over the development wheel to produce automated devices that provide more safety, quality and comfort to users. However, the integration between commercial hardware and a variety of software applications is a growing challenge for embedded system developers. The best way to overcome this challenge is to use a design practice that simplifies and presents the system in a level of abstraction to provide easier understanding for the system requirements and analysis. This design practise is better known as Model-Based Engineering (MBE), where models are higher abstraction of software and hardware components that support engineering activities such as system analysis, verification and implementation.

Using MBE allows developers to start with most basic system architecture they are provided with and have some predictive analysis early in the development life cycle. As the system design is refined throughout the life cycle, the model becomes more detailed and provides finer granularity that allows for a greater insightful analysis of the system's qualities and capabilities [13]. Some argue that modeling takes longer time in the design stages, however, using the right modeling tools will take down the time consuming of the overall development cycle a notch, because it detects most of the faults as early as it can be addressed and fixed. However, not all modeling approaches are suitable for all systems. Choosing the right modeling language and analysis tools is a very important task. With this in mind, we studied multiple modeling languages to figure out which is the best for our system. Recently, several modelling languages have been standardized. Unfortunately, none of them is considered to provide the full scope required to cover all aspects of systems design effectively.

One of the modelling languages is Systems Modelling Language (SysML) [33]. SysML is a Unified Modelling Language (UML) profile, which is oriented to system engineering for analysis, design, and verification of complex systems. These systems may include software, hardware, procedures, information and facilities. Another UML profile language is Modeling and Analysis of Real Time and Embedded systems (MARTE) [34]. MARTE provides a generic canvas to describe and analyze systems. Both languages mentioned above are adopted by Object Management Group (OMG) and are extensions of the original UML language.

Another modelling language, we have already mentioned before, is AADL, which is standardized by the Society of Automotive Engineers (SAE) [13]. AADL has well defined semantics for representing the architecture of software-intensive large-scale embedded systems and systems of systems, such as autonomous systems, motorized vehicles, aircraft, spacecraft and medical devices.

2.3.1. Architecture Analysis and Design Language (AADL)

AADL by definition is an architecture modeling language that is mainly targeted for real-time safety-critical systems. AADL is a component-based modelling language provides both textual and graphical interfaces, where an AADL model contains component types, interfaces, implementations, subcomponents, data flows and other properties [35]. The language provides hierarchical architecture where the top level component is the root system. AADL defines a variety of component structures, each represents a different entity.

2.3.1.1. AADL Language Overview. Components in AADL are divided into three main categories: software, hardware and hybrid (generic and composite) components [35]. Process, thread, subprogram, thread groups, and data represent the software components of the designed system architecture. While processor and memory are execution platform components represented in terms of hardware, device and bus components are represented as the physical system. The hybrid components are basic

Table 2.2. AADL components categories.

Category	**Description**	**Type**
Data	Represents the data used in the source code and data types	Software Components
Thread	Represents a schedulable concurrent execution unit	
Process	A protected address space that contains the threads and data subcomponents to execute the software.	
Device	A physical component that interact with the outside world, like a sensor or actuator.	Hardware Components
Processor	Represents an execution entity that executes threads and virtual processors	
Memory	Represents an entity that stores data and code.	
Bus	Represents a physical connection between units or entities	
System	Represents a unit that encapsulates the necessary subcomponents required to implement the system	Composite

constructs that do not represent anything concrete. They are used either to encapsulate and combine multiple components into one shell as a system component, or to specify an unknown component to be refined later on in the design process. AADL language is extensible to support annotation of models with user-defined and analysis-specific properties and with specialized annex sublanguages. For example, the Behavior Annex standard for AADL (BAnnex) extends the core of AADL to denote the components interaction behavior to address the safety-criticality aspects of the system. The Error Model Annex standard for AADL (EAnnex) extends the core of AADL to provide fault-tolerance and reliability aspects of the system architecture through specifying fault behavior and error propagation. Other annexes are under development [13]. Table 2.2 lists the component categories that we used in designing AADL models and their definitions. Other categories of AADL can be found in [13] [35].

An AADL model consists of component types and implementations, where the component type captures the component's external interface and how it communicates with other components. The component implementation basically describes the component internal architecture and properties, where it specifies how the interface and the subcomponents are connected to provide the services. A model can have a

type and multiple implementations, where this great feature allows for separating the component's specification from its implementation that allows the developer to define different architectures and test for best performance, and eventually choose the right one. Any AADL component type is characterized by a component category, unique identifier, features, flows, and properties. The features, basically, represent the ports in which they provide an input or an output to be connected later to another component. Ports are of three kinds: event, data, and event data ports. From their names, the ports carry data or/and events inside or outside the component where they are transferred by a connection defined in a higher level component. Flows are defined to specify how the data/events travel from inputs to outputs, where they are refined later in an implementation. Properties provide characteristics to the model and present more specifications. The component implementation, as stated earlier, defines the internal architecture by specifying subcomponents and connections, refining flows and adding more properties. A subcomponent can be any component category that is lower in hierarchy level. For example, a process can have threads as subcomponents, where a thread can have subprograms as subcomponents. A system component can have subcomponents of all categories. When a component implementation specify subcomponents, it is important to capture how these components are connected to each other and to the predefined ports, which is declared in the connection section of a component implementation. Flows provide detailed flows of the data from one port to another passing through the subcomponents. In a similar approach to inheritance in object-oriented programming, component's types and implementations can be extended to provide reusability and flexibility. A component cannot extend more than one component, and if a property was defined in the parent component, it will be overridden by the extending component. A complete model in AADL must have a system component that connects all the defined hardware and software together to define their integration and specify their interactions through different connections and properties.

2.3.1.2. AADL Tools. After the language was standardized, software companies and universities developed many tools for AADL. The most famous one is OSATE [14]. OSATE (Open Source AADL Tool Environment) is developed by CMU-SEI (Software

Engineering Institute at Carnegie Mellon University). OSATE supports the latest version of the language and it is constantly growing and developing. This tool provides textual and graphical editor in addition to a variety of analysis tools such as latency, scheduling and safety tests.

After constructing a well-designed model using AADL language, it needs to be verified as a correct model that achieves the goal it was designed for. The verification and validation of real-time embedded systems designed by architectural modelling languages like AADL is a research challenge. Model verification comes in different approaches: analysis reports, simulation, model transformation and code generation. There have been many studies on model verification. Some of them consider mapping the model into another language that can be verified using its tools, others take AADL models as they are and provide testing and verification capabilities such as COMPASS Toolset [36]. COMPASS is an integrated toolchain for verification and validation of AADL models using advanced tools and model-checking techniques. Unfortunately, COMPASS tool is only available to download for European Space Agency (ESA) members. Hence, it is not available for academic use [36]. Another application used for AADL model verification is Ocarina [37]. Ocarina is a command-line tool that can be used to validate AADL models using semantic checks, code generation and so on. OSATE includes a bridge to Ocarina to allow code generation of AADL models. However, there is another application that contains Ocarina in addition to Cheddar [38], the famous scheduling software. This application is called AADL Inspector [39], from Ellidiss technologies. AADL Inspector supports the AADL core language and its Behavior annex but does not support the graphical models. It provides many different verification tests such as static analysis and Ocarina semantic checks as well as system execution simulation.

3. MODEL-BASED DESIGN OF A ROADSIDE UNIT FOR EMERGENCY AND DISASTER MANAGEMENT

In this thesis, we propose an intelligent emergency management system composed of an RSU and cloud-computing services. Our aim behind this system proposal is to provide incident detection and reporting as fast as possible. We believe that by depending on the most recent and developing technologies and algorithms from different resources, our system will be fault-tolerant by having alternative solutions to overcome shortcomings in each exploited technology. In this chapter we will describe the whole system in general, explore the internal architecture of our RSU, how the functionality is modelled, what are the necessary devices to use, and how all of the system is deployed onto multiple different platform options using AADL modeling language.

3.1. Proposed System

In this thesis, we propose a state-of-the-art architectural model of an RSU, which depends on the concept of the Intelligent Data Processing (IDP) layer proposed in [5]. Our model integrates existing partial solutions and provides incident verification approach to minimize false alarms in addition to communication capabilities to pass routing messages to EVs and evacuation broadcasts to passing vehicles to clear a specific lane ahead of the EVs. While the proposed RSU is the device that detects and report traffic incidents, we propose to use cloud computing to perform incident classification using machine learning techniques. Incident classification is an important step before a proper response can be decided upon receiving the incident alarm. Using machine learning techniques helps the system to learn from its own incidents' data to classify incidents into severity levels depending on data coming from all RSU devices, in which it provides rich and meaningful data to be trained. After the incident has been detected by the RSU, the cloud is responsible for classifying and reporting the incoming information to the related authorities, and creating routing plans for EVs to reach the incident site without delay. The cloud will relay the routing instructions to

all RSUs along the planned route to guide the EVs to its destination and provide lane evacuation technique so the EVs will not suffer through traffic jams. Our proposed system is shown in Figure 3.1

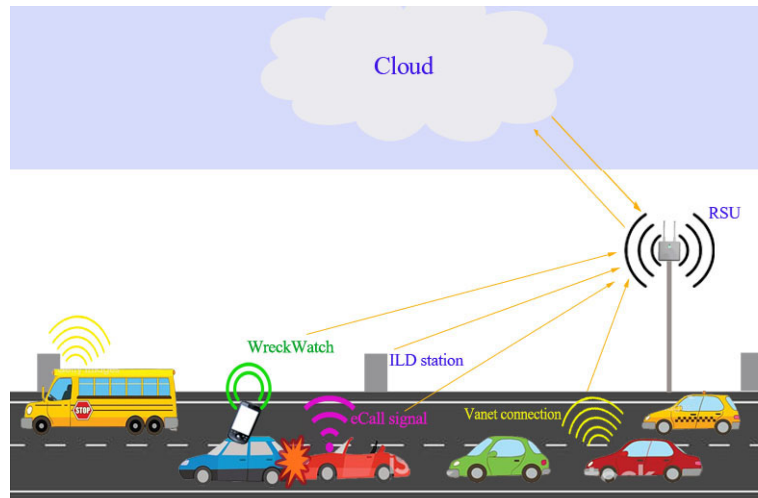

Figure 3.1. The proposed system.

To explain our RSU model in a sophisticated representation, we chose AADL as the modelling language to present the RSU architecture and provide multiple and different deployment implementations. We used OSATE to model our RSU and produce two analysis tests: schedulability and flow latency analysis. In addition, we used AADL Inspector software to provide static and dynamic schedluing simulations on our RSU model.

3.2. Roadside Unit (RSU) Model

In this study we propose an RSU that uses four detection techniques. Two of the detection techniques are algorithms which depend on real-time traffic variables. These traffic variables are measured through two different technologies to provide better detection rate and less false alarm rate in our RSU . The first one is the ILD road sensor technology, where we propose to use the algorithm we mentioned in Section 2.1.3 [23]. We consider a 1 km road of two lanes and divided into 10 sections with an ILD at each section per lane. These ILDs will be connected to our RSU through RS485 cables [40], where they send their traffic variables every one minute time interval. After receiving

these data, the RSU will use the algorithm to detect if there is an incident or not. The second detection algorithm depends on VANET communication technology. The algorithm collects data from passing vehicles and process them to find anomalies the normal traffic would not have. This algorithm is mentioned in Section 2.1.4 [31].

The other two detection mechanisms are eCall [19] and WreckWatch [10]. Both of these mechanisms use direct sensing to detect an accident, where the algorithm depends on car sensors and smartphones sensors directly, not like collecting traffic parameters and process them for finding anomalies. Our RSU will play the server side that listens to accident signal coming from these channels. For eCall, as stated earlier, every modern vehicle will be equipped with an IVS device to provide instant incident detection and to communicate with the nearest PSAP point to report the accident. We propose that the PSAP data modem is integrated in the RSU where the IVS will send the eCall MSD directly to the RSU and the RSU will handle the data and pass it automatically to the cloud. We believe that with adding this technique, we provide faster response through automatic reporting and communication. WreckWatch, as explained before, is a client/server application, where the client is installed on the smartphone and is responsible for the incident detection based on multiple direct sensors on the phone. We propose to include the server side on our RSU, where it listens for and handles the incoming WreckWatch data and passes them automatically to the cloud. After an incident is detected on the RSU, we propose a verification method to screen false alarms. This method depends on confirming the detected incident by receiving a second incident alarm from a different detection mechanism that matches the same information. We assume that an RSU is installed every 1 km in urban areas and highways. Our RSU does not need to communicate with another RSU to complete its job, while it communicates with the cloud, EVs and passing vehicles.

To put it in summary, our RSU does four basic jobs:

- Listening
  - Traffic flow variables through physical connections to ILDs
  - Vehicles' traffic variables through VANET
  - eCall signal and data through cellular networks
  - WreckWatch signal and data through internet connection
  - Cleared incidents data through cloud connection
- Thinking
  - Automatic incident detection using ILDs traffic variables
  - Automatic incident detection using traffic data from VANET
- Processing
  - Processing eCall and WreckWatch data
  - Recording and comparing new incidents with possible detected incidents data
  - Confirming possible incidents
  - Clearing confirmed incidents
- Communicating
  - Sending confirmed incidents to cloud
  - Broadcasting lane evacuation messages to passing vehicles
  - Routing messages to EVs

In AADL, we have decomposed our RSU software architecture into four units (processes): Input, Detection, Processing, and Output units. The Input unit receives the incoming data and sends it to its related parties, while the Detection unit is responsible for incident detection depending on current traffic variables. The Processing unit takes all the incoming incident alarms and apply the verification technique to confirm them. In the meantime, the Output conveys messages to EVs and passing vehicles in addition to sending confirmed incident information to the cloud for classification and routing plans.

3.2.1. Input unit

The input unit is a process that consists of all the tasks (threads) that are responsible for listening, receiving and filtering the incoming data from different resources. As we stated earlier, we are exploiting multiple detection technologies into our system, and therefore we need to provide a specific task for each communication technology for receiving their data and distributing them to the related processing parties. The input unit, as shown in Figure 3.2, contains 4 threads: ild_Listener, VanetFilter, Wireless-Filter, and eCallPoll.

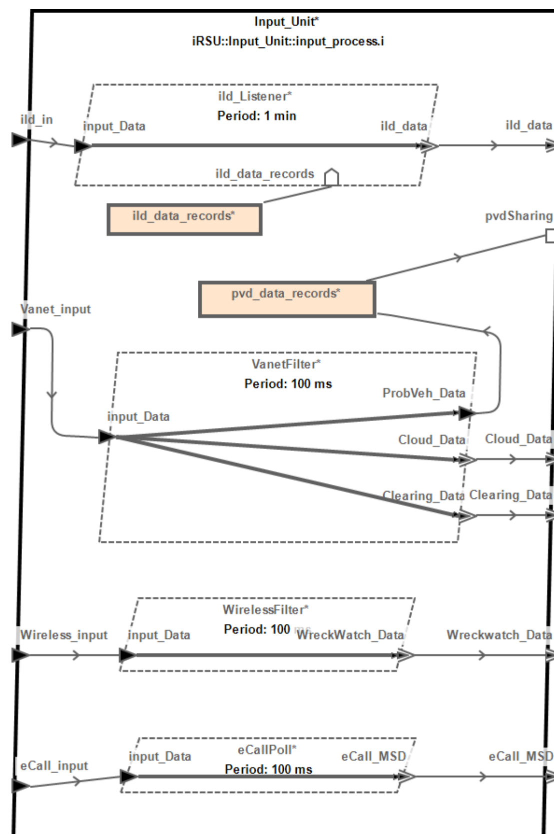

Figure 3.2. Input Unit's AADL model.

The ild_Listener is a periodic thread that listens to incoming data from all the ILD devices connected to our RSU. Since the loop detector devices need a whole minute to produce meaningful data, the period of this task should also be one minute, to avoid unnecessary task switching delay. The ild_Listener aggregates all the incoming data into one data component *ild_data_records*. Once it is finished, it issues an event to start

the related detection algorithm and passes the aggregated data.

The VanetFilter is a periodic thread that receives and filter incoming cloud data messages and probe vehicles data (PVD) messages through VANET's connection and select only the related data. The thread first has to differentiate between a cloud message and a vehicle message by its header. The cloud message is either a routing message or a cleared incident notification. An event is issued to dispatch the routing thread or the verification thread, respectively, and passes it the message. If the incoming data is a PVD message from a connected vehicle, it will be recorded into the data component *pvd_data_records* until it accumulates 24 records. When new data comes, it replaces the oldest data in the data component. We specified the period for this thread as 100 ms.

The eCallPoll thread is also a periodic thread. It listens to incoming eCall signals. Once an eCall signal arrives, the thread receives the data, issues an event and sends the data to the eCall handler task in the processing unit.

The WirelessFilter is also a periodic thread that listens to incoming WreckWatch data coming through the wireless connection. When a WreckWatch message is received, the wirelessFilter issues an event to dispatch the WreckWatch handler, in the processing unit, and passes the message to it. The period both for the WirelessFilter and eCallPoll is specified at 100 ms.

3.2.2. Detection Unit

The Detection unit is the process that is responsible for traffic incident detection depending on two different AID algorithms. The detection unit contains three threads: ILD_th, Van_th, and Incident_th as shown in Figure 3.3.

The Van_th thread is the task that applies the incident detection algorithm depending on traffic parameters from VANET [31]. This thread is a periodic thread that

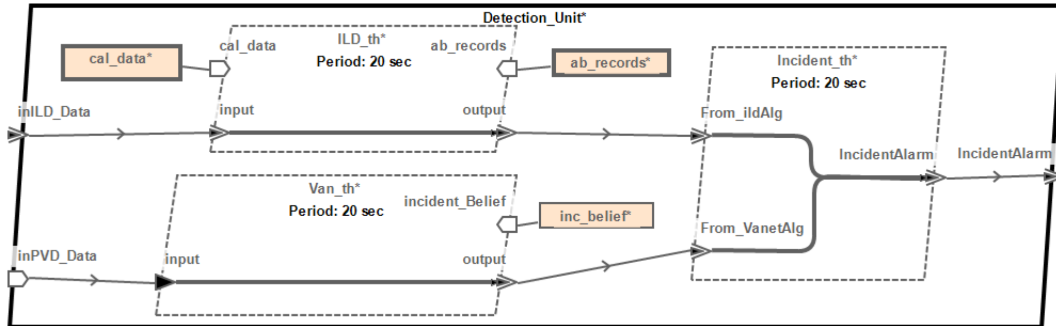

Figure 3.3. Detection Unit's AADL model.

is dispatched every 20 seconds. As we see from Figure 3.3, the Van_th thread has an access connection (inPVD_Data) to the *pvd_data_records* data component that is filled by the VanetFilter thread in the input unit. Once the Van_th thread is dispatched, it reads the accumulated data in the *pvd_data_records* data component, if the data is enough and meaningful, it executes the detection algorithm. The detection algorithm takes the traffic data and spots anomalies based on distance and vehicle speed calculations and increase the incident belief value. As we see in Figure 3.3, the Van_th uses a data component *inc_belief* that holds the incident belief value in memory where it can change this value according to the algorithm's output on every dispatch. If the incident belief exceeds a predefined threshold (in our implementation we specified that as 2), the thread issues an event and sends the incident data to the incident_th task. As we mentioned above, we specified a period of 20 seconds for this thread, because with high traffic flow volume and density, collecting data from 24 vehicles might finish a lot faster than 1 or 2 minutes that was mentioned in the algorithm paper itself [31].

The ILD_th thread applies the incident detection algorithm based on inductive loop detectors (ILD) that we explained in Section 2.1.3 [23]. It analyses the incoming data and detects anomalies among the current traffic variables. This thread uses two data components *ab_records, and cal_data*. *Cal_data* is the calibration data that was measured and established previously when no incident is present. These data are specific to a time and a day of the week, in which the algorithm uses to compare with the current traffic variables to detect anomalies. Once an anomaly is detected, the lane and road section data are recorded in the *Ab_records* data component. *Ab_records*

contains a list of all the road sections and lanes that had presented an anomaly in the previous execution only. So, if an anomaly presented itself in the current execution and matched a previous one, it means there is an incident to report. The thread issues an incident alarm event and passes the incident info to the incident_th task. The ILD_th is sporadic, and depends on collected data that are coming from the ild_listener thread in the input unit. We also specified a 20 seconds period for this thread even though the data cannot be ready before a full minute interval. The reason is explained in the following paragraph.

The Incident_th thread is basically the collector of incident alarm information from the preceding two threads. It records the information in a unified representation and passes it with an event to the verification task in the processing unit. The Incident_th thread is also sporadic and follows the same period, because the scheduler specifies that with task presidencies - in which ILD_th and Van_th precedes Incident_th - all tasks should have the same period. Because we specified Van_th period to be 20 seconds, so the Incident_th should also be 20 seconds, and therefore the ILD_th period has to be 20 seconds, too. Another reason is that if we had specified each of them to have one minute interval, it will add more latency on the execution time and data flow, and slow down the system response time.

3.2.3. Processing Unit

The processing unit is a process that wakes up when an incident alarm is issued. It deals with all the incident detection methods and technologies, and verifies the detected incidents before reporting them. The processing unit, as shown below, consists of three tasks: Th_eCallHandler, Th_WreckWatchServer, and Th_Verification.

Th_eCallHandler thread is dispatched when the eCallPoll thread in the input unit receives an eCall signal and issues an event. It is responsible for handling the MSD data coming from the input unit and transfer it in a unified form to the Th_Verification task. This thread is sporadic and its specified period is 30 ms. Th_WreckWatchServer

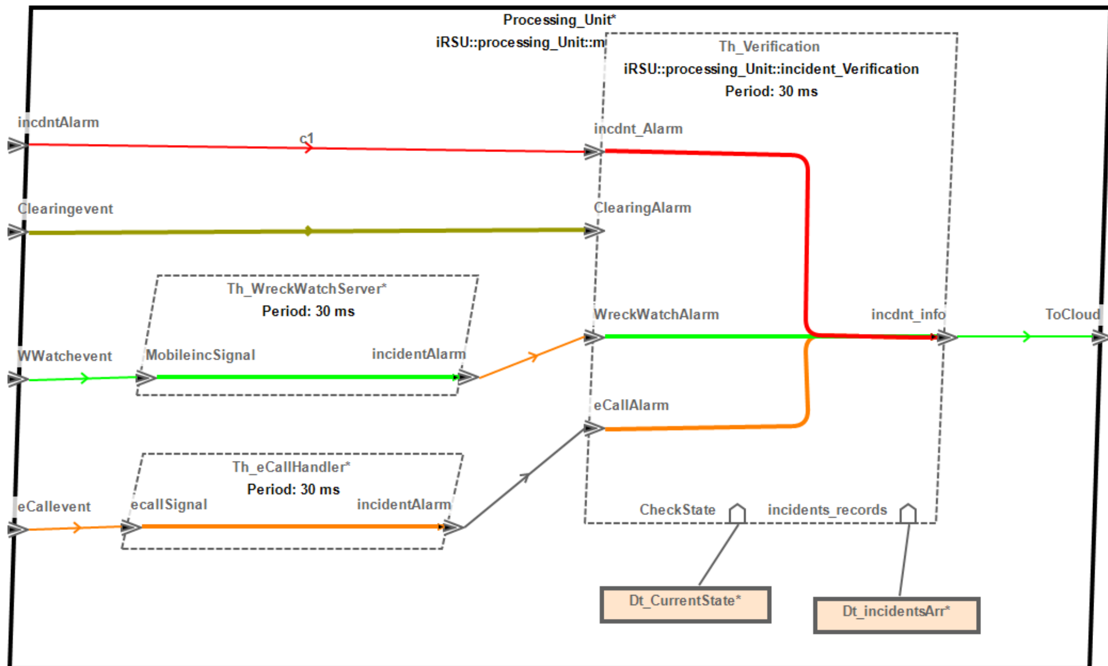

Figure 3.4. Processing Unit's AADL model.

thread applies the server side application that is mentioned in Section 2.1.2 [10]. It handles the coming data from the WirelessFilter that is related to the WreckWatch application and sends the data in a unified form to the Th_Verification thread. This thread is also sporadic since it only works when a WreckWatch signal arrives, and its period is as the WirelessFilter, 30 ms. Th_Verification task is sporadic and is dispatched every time an incident alarm event is issued by either Incident_th, Th_eCallHandler, Th_WrechWatchHandler, or by a clearing event issued by the VanetFilter in the Input unit. Since this thread is dispatched by four different tasks, we specified its period as 30 ms to be able to respond to all incoming data in the worst case scenario. The verification task job is to confirm a possible traffic incident by receiving another incident alarm from another detection technology. This thread depends on recording incoming incident alarms and change the system state according to current situation. When no incidents are detected, the system is in normal state. When an incident alarm is issued the system switch to investigation state, only if the incident alarm is coming from the detection unit. When all recorded incidents are confirmed, the system state switches to clearing and awaits for a confirmation from the cloud that the incidents have been cleared to return to the normal state.

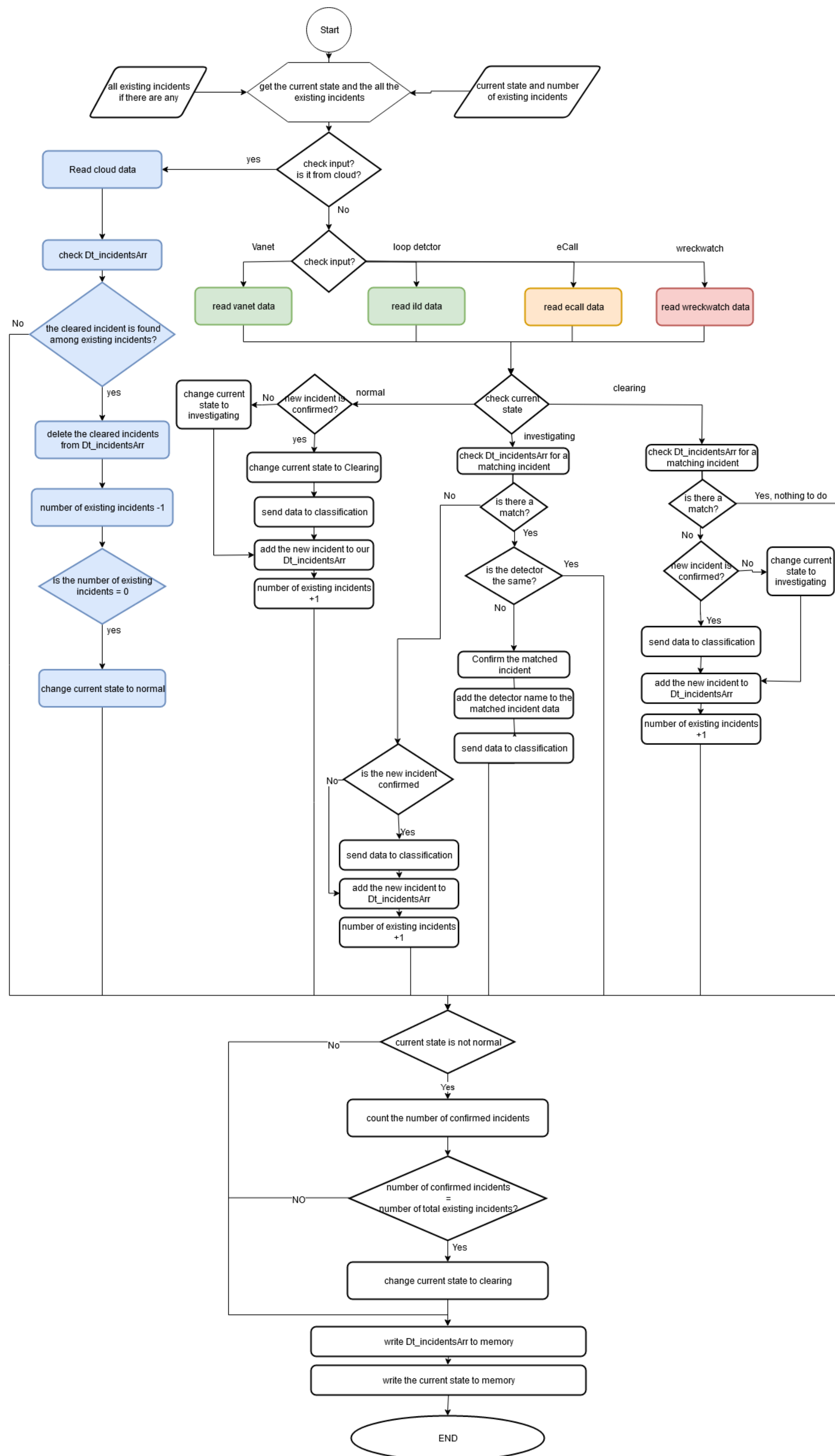

Figure 3.5. Th_Verification task execution flow.

The system state is recorded in the data component *Dt_currentState* which contains the current state and the number of incidents on the list. We consider that the traffic incidents coming from both eCall and WreckWatch channels are confirmed and do not need to be verified, because they depend on direct sensors to detect a crash. But, the traffic incidents detected by AID algorithms that depend on traffic variables (which are error prone and have some ambiguity) need to be verified. Even if the detection algorithm provides false alarm prevention techniques, it is important to provide a verification step to insure not to exhaust the authorities with frequent false alarms. So, when an eCall or WreckWatch signal arrives, they are considered confirmed, they are recorded in the data component *Dt_incidentArr*, and then reported to the cloud. However, when one of the incident detection algorithms in the Detection unit sends a possible incident alarm, the verification thread first check if there is a match in the previously recorded incidents, if there is a match and it was detected by another detection technology, then the incident is confirmed and reported. If not, then the incident will be added to the list and awaits confirmation. Figure 3.5 shows a flow chart of the verification task execution steps.

3.2.4. Output Unit

Output unit is the process that is responsible for conveying routing messages and confirmed incident reporting to the related parties. This unit consists of two tasks: Routing_Th and ToCloud_Th. The Routing task receives instructions from the cloud to either broadcast a message for the passing vehicles to evacuate a certain lane for a coming EV, or to relay the routing messages to the EV passing through the RSU communication range. ToCloud_Th thread is responsible for reporting the confirmed incident data to the cloud for incident classification. Both of these threads are sporadic and have a period of 30 ms.

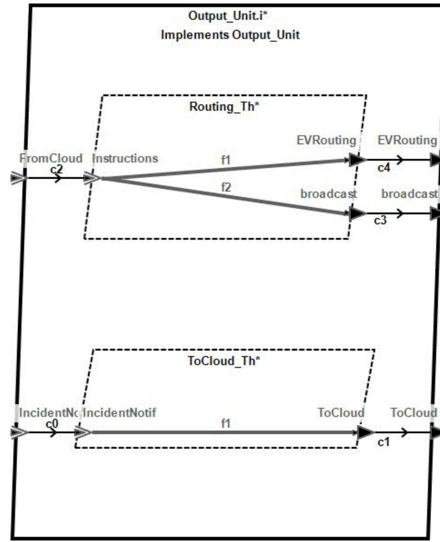

Figure 3.6. Output Unit's AADL model.

3.2.5. Devices

In order to get the data we need to process, the RSU depends on multiple communication devices and some road sensors. All the devices are shown in Figure 3.7.

Since the RSU will use the incident detection based on ILDs, we propose to use an interface that is responsible for connecting the ILDs through RS485 connection, and transfer the data to the RSU on the UART bus [41]. In addition, this interface will need a bridge [42] for the UART bus to connect with the PCIe 3.0 bus we are using as a system bus.

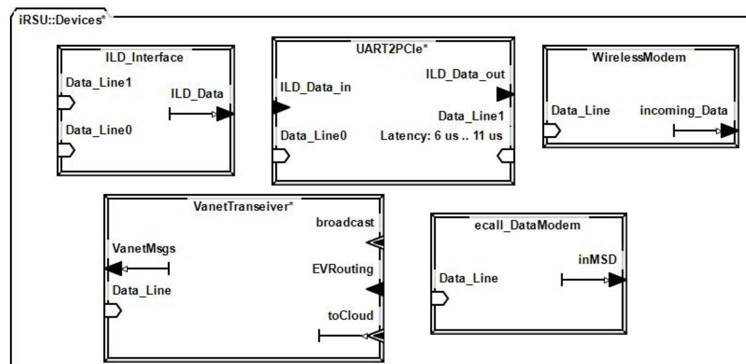

Figure 3.7. The devices used in RSU.

One of the communication devices is the VanetTransceiver. The VanetTransceiver is a device that is responsible for sending and receiving VANET messages from/to the passing vehicles and the cloud through V2I and I2I communication standards [27] [28]. The second communication device we are using is the Wireless modem (Wi-Fi), where it plays a role as a hotspot for smartphones, as provided in Siemens RSU device [6]. It is responsible for getting the WreckWatch signal and data when a smartphone owner, whose smartphone is successfully connected to our RSU, is involved in an incident.

3.2.6. Complete Functional Model

So far, we have described the software components of our RSU separately in addition to the devices that facilitate data transfer from/to the RSU. Figure 3.8 shows how these components are incorporated together in one system, the RSU. In the figure we see how the four processes are connected together, and how the data flows from the source to the destination. In AADL, it is important to specify how the data are flowing through the system by defining a flow source where the data starts, a flow sink where it ends, and a flow path where it is passing. In the top level system, we define an end-to-end data flow, by specifying each connection from the flow source through the flow path all the way to the flow sink. Defining an end-to-end data flow provides an insightful look on how long it takes the system to respond, or in other words, what is the maximum latency the system will present.

Now that we specified how the RSU works regarding functionality, next we specify the execution platform that will make these tasks productive. The execution platform consists of three components: processor, memory and bus. The right choice of the CPU, memory and buses and the definition of their interconnection will affect the execution time, scheduling and performance. In the spirit of experiment and research, we opted to provide five variations of execution platform inspired by the commercial RSUs mentioned in the related work chapter [7] [6]. To be able to make as many variations as we wanted, we extended (inherited) the functional model to each one of the deployment options as shown Figure 3.9.

Figure 3.8. The complete functional AADL model.

3.2.7. Platform

As we see in Figure 3.9, iRSU_Sys is the top level system and it is the system component type, where it defines the system interface. The iRSU_Sys.UrbanMinimal is the system component implementation where it captures the functionality of the proposed system through defining software processes and hardware devices and their interactions and interoperability. Each deployment option couples a specific execution platform with the functional model. Each deployment option is defined through a system implementation component that extends the iRSU_Sys.UrbanMinimal, adds execution components such as processor, buses and memory, and associates the software components and their connections to each execution component through binding statements.

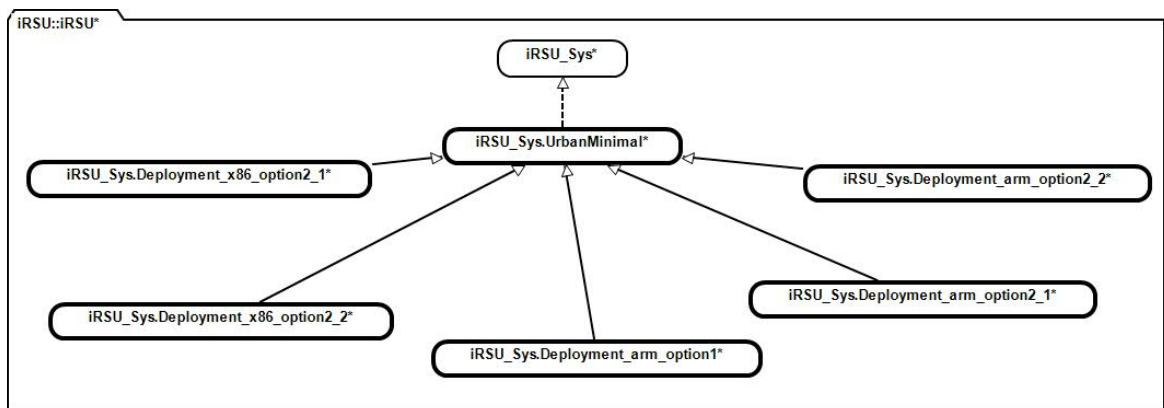

Figure 3.9. Deployment options.

Before we explore each deployment variation, we need to specify how the execution platform is connected to and associated with the software and hardware components. Since we are using only a single core processor, the connections between the functional model and the execution platform is almost the same in all the variations, except for the iRSU_Sys.Deployment_arm_option1, where we added an L2 Cache memory to the system, but this component is only connected to the processor through the main bus, which does not affect the overall interoperability between components. In Figure 3.10 we see how all the components inherited from the functional model are connected to the execution platform.

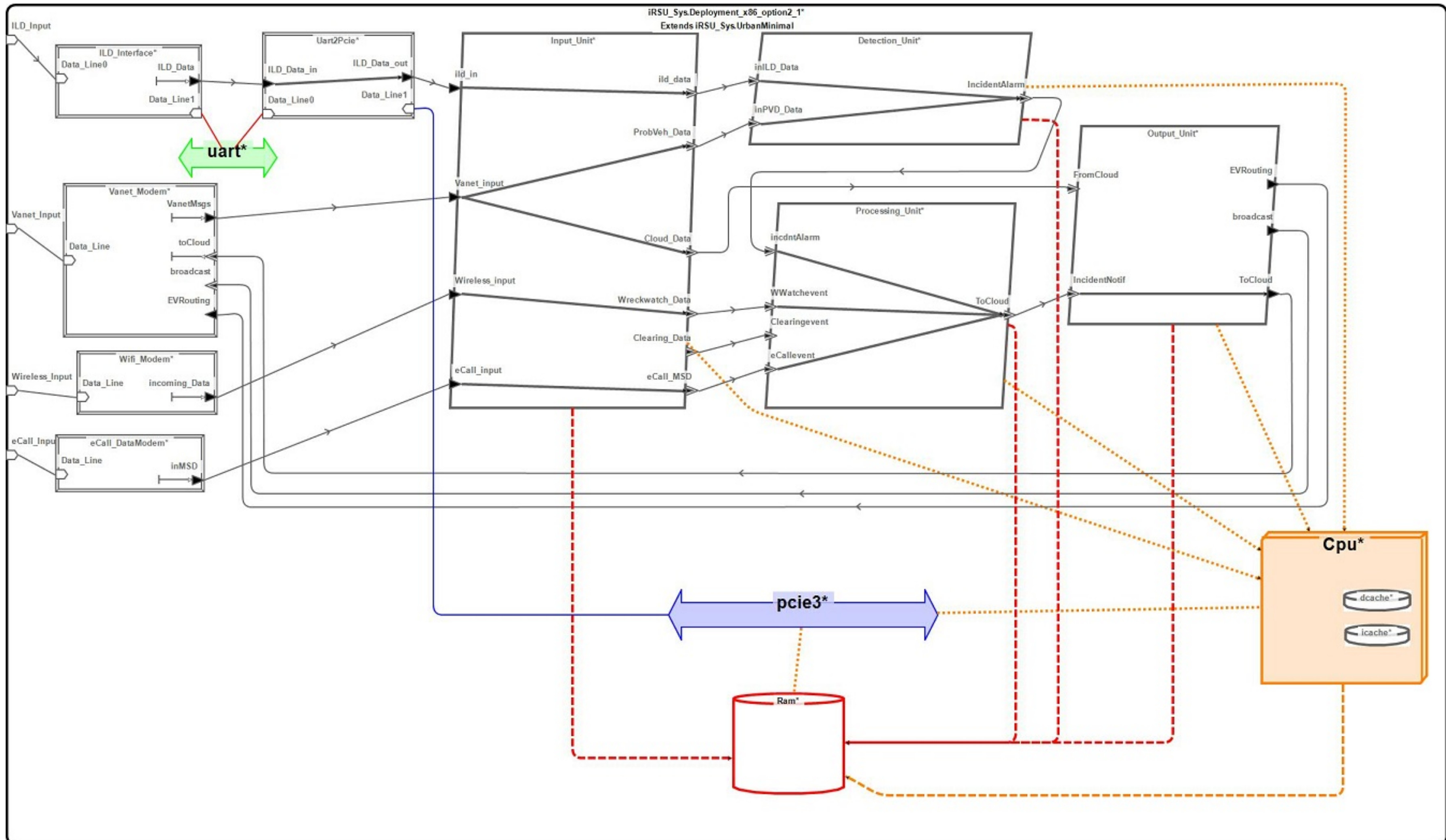

Figure 3.10. System integration.

As shown in Figure 3.10, we have a single core CPU with L1 Cache, 1 Memory component, PCIe 3.0 bus, and UART bus. The PCIe 3.0 is the bus that connects all the peripherals with the CPU and the memory. The UART bus connects the devices responsible for reading ILDs data. The CPU is connected to all software components, the memory, and the PCIe 3.0 bus. The memory also connects to the software components, and exchanges the same connection with the CPU and PCIe 3.0.

The reason we have so many variations, although they are all connected in the same way, is the CPU type, speed and internal architecture that plays a significant role in task execution times. We will see in the next chapter how these characteristics affect the overall performance of the system.

In AADL, we cannot capture the internal architecture of the CPU, we only define specific performance and scheduling properties. However, the difference is shown in the execution time of each task. More details will be provided in Chapter 5, as we explain all of the execution platform options, where we provide our experiments using gem5 and AADL testing.

4. EXPERIMENTS AND RESULTS

In Chapter 4, we explained the proposed RSU internal structure, its subcomponents and their interactions. The next step is the model validation and verification using different simulation and testing tools. In this chapter, we will explain how we implemented the RSU tasks, found their execution characteristics, and applied scheduling and latency tests in addition to simulation reports. Our experiments is divided into five stages:

(1) Tasks' code design and implementation in C++.
(2) Tasks simulations on gem5.
(3) RSU Model's schedulability analysis in OSATE.
(4) RSU Model's flow latency analysis in OSATE.
(5) Scheduling simulations in AADL Inspector

4.1. Tasks' code design and implementation in C++

In order to apply model verification in AADL, we have to specify each task's execution time and deadline. However, since we do not have the actual codes of ILD based algorithm [23], VANET based algorithm [31], the server side of eCall [19] and WreckWatch [10] , we tried to imitate the algorithms and applications depending on the explanation provided in their papers. Therefore, we had to go into this stage, where we implemented the detection algorithms and solutions to the best of our capabilities. For each task, we coded a small program, using C++ language, that implements the proposed functionality in this task provided with an assumed input data to confirm its workflow.

In ILD based detection algorithm, as explained earlier in Section 2.1.3, they depended on Fuzzy Logic concept and used an inference system with embedded fuzzy rules to provide an output for the current traffic variables [23]. However, they did not

provide all the fuzzy rules they used, probably because they were too many. Since they used two inference engines, each with four input variables, including three fuzzy sets per variable, it resulted in 81 fuzzy rules for each inference engine's output variable. For us to be able to implement this algorithm, we assumed the 162 fuzzy rules based on common knowledge in traffic engineering, and applied the inference system in C++. For the VANET based detection algorithm, they provided a pseudo code for their methods [31], which was good enough for us to interpret the algorithm code. For eCall and WreckWatch servers, there were not enough details about how exactly will server react to their signals, but since our RSU plays as a relay point for these signals, there was not much to code other than forward the incoming incident data and use it for confirming detected incidents. The rest of the tasks was implemented according to their proposed functionality we explained in Chapter 4. The code for each task is available on the attched CD.

4.2. Tasks' simulations on gem5

After we implemented all the tasks, we wanted to capture their execution time over a specific execution platform. One of the simulation software to use for this purpose is gem5 [43]. gem5 is a modular platform that works on Linux and provides system-level architecture and processor microarchitectures.

Inspired by RSUs from Siemens [6] and NXP [7], as mentioned in Chapter 2, we propose the execution platforms, that are presented in Table 4.1 and Table 4.2, based on CPU microarchitecture, CPU speed, and memory options.

Table 4.1. The first option based on NXP RSU.

OPTION 1: NXP								
Environment	CPU	CPU Microarchitecture	CPU clock	Memory	Memory Size	iCache	dCache	L2 Cache
ARM	DerivO3	out-of-order	1 GHz	DDR4	1 GB	64 KB	32 KB	1 MB

Table 4.2. The second option with 4 variations based on Siemens RSU.

OPTION 2: Siemens								
Environment	CPU	CPU Microarchitecture	CPU clock	Memory	Memory Size	iCache	dCache	L2 Cache
X86	Minor	in-order	800 MHz	DDR4	1 GB	64 KB	32 KB	none
	DerivO3	out-of-order	800 MHz	DDR4	1 GB	64 KB	32 KB	
ARM	Minor	in-order	800 MHz	DDR4	1 GB	64 KB	32 KB	
	DerivO3	out-of-order	800 MHz	DDR4	1 GB	64 KB	32 KB	

In order to simulate our task implementation on gem5, first, we compiled our code files into X86 and ARM environment binaries. Then, we applied se.py configuration script under the proposed configuration options in Table 4.1 and Table 4.2. An example of a simulation command line in gem5 is presented in Figure 4.1.

```
nur@DESKTOP-OB8NGKC:~/gem5$ sudo build/X86/gem5.opt --outdir="irsu/input_unit/ecall_poll/x86/option2_1_m5out" configs/example/se.py
--cpu-type="MinorCPU" --sys-clock="800MHz" --mem-type="DDR4_2400_16x4" --mem-size="1GB" --caches --l1d_size="32kB" --l1i_size="64kB"
  --cmd="irsu/input_unit/ecall_poll/x86/ecall_poll"
```

Figure 4.1. Simulation command example of ecallpoll thread on x86 processor.

After applying the configuration options, gem5 presented us with simulation results for each task in the m5_out/stats.txt file that was generated after each simulation. We present all configuration options results in Tables 4.3 - 4.7.

Table 4.3. gem5 results for the proposed option 2_1 on X86 environment.

Option 2_1 : **Siemens**	**CPU specifications**	**Memory specifications**			
	X86 architecture	RAM	Caches		
	MinorCPU (in-order)	DDR4_2400_16x4	dcache	icache	L2 cache
	CPU clock: 800 MHz	Size: 1 GB	32kB	64kB	none
	Stats				
#	**Unit**	**Tasks**	**Simulated Exec Seconds**	**Simulated ticks**	**Notes**
1	Input Unit	VanetFilter	3652 Ms	3,652,407,500	measurements are in Microseconds (Ms)
2		ild_listener	3946 Ms	3,946,495,000	
3		WirelessFilter	3627 Ms	3,627,221,000	
4		eCall Poll	3756 Ms	3,755,833,500	
5	Detection Unit	IDL_Th	4730 Ms	4,729,649,000	
6		Van_Th	4543 Ms	4,542,773,000	
7		Incident_Th	3821 Ms	3,820,887,500	
8	Processing Unit	Th_WrechWatchServer	3778 Ms	3,778,065,000	
9		Th_eCallHandler	3786 Ms	3,786,741,000	
10		Th_Verification	4704 Ms	4,704,134,000	
11	Output Unit	Routing_Th	3767 Ms	3,767,040,000	
12		ToCloud_Th	3756 Ms	3,756,407,500	

As we see from these results, the X86 in-order CPU at 800 MHz speed produced execution times between 3627 and 4730 microseconds (3.5 - 5 milliseconds). In Table 4.4, we present the gem5 results for option 2_2 for X86 environment. In Table 4.4,

Table 4.4. gem5 results for the proposed option 2_2 on X86 environment.

Option 2_2 : **Siemens**	**CPU specifications**	**Memory specifications**			
	X86 architecture	RAM	Caches		
	DerivO3COU (out-of-order)	DDR4_2400_16x4	dcache	icache	L2 cache
	CPU clock: 800 MHz	Size: 1 GB	32kB	64kB	none
	Stats				
#	**Unit**	**Tasks**	**Simulated Exec Seconds**	**Simulated ticks**	**Notes**
1	Input Unit	VanetFilter	1452 Ms	1,452,124,000	measurements are in Microseconds (Ms)
2		ild_listener	1605 Ms	1,604,809,000	
3		WirelessFilter	1464 Ms	1,463,629,000	
4		eCall Poll	1501 Ms	1,501,311,500	
5	Detection Unit	IDL_Th	1939 Ms	1,938,541,500	
6		Van_Th	1808 Ms	1,808,173,000	
7		Incident_Th	1546 Ms	1,546,278,000	
8	Processing Unit	Th_WrechWatchServer	1533 Ms	1,533,461,500	
9		Th_eCallHandler	1530 Ms	1,529,528,000	
10		Th_Verification	1851 Ms	1,850,846,500	
11	Output Unit	Routing_Th	1528 Ms	1,528,221,500	
12		ToCloud_Th	1515 Ms	1,515,041,500	

we see the execution times are almost half of the ones in Table 4.3, due to CPU micro-architecture which changed from in-order to out-of-order CPU, and produced execution times between 1452 and 1939 microseconds (1 - 2 milliseconds). However, the ARM environment provide better results as we see in Table 4.5:

Table 4.5. gem5 results for the proposed option 1 on ARM environment.

Option 1 : **NXP**	**CPU specifications**	**Memory specifications**			
	ARM architecture	RAM	Caches		
	DerivO3COU (out-of-order)	DDR4_2400_16x4	dcache	icache	L2 cache
	CPU clock: 1 GHz	Size: 1 GB	32kB	64kB	1 MB
	Stats				
#	**Unit**	**Tasks**	**Simulated Exec Seconds**	**Simulated ticks**	**Notes**
1	Input Unit	VanetFilter	69 Ms	68,916,000	measurements are in Microseconds (Ms)
2		ild_listener	111 Ms	110,850,500	
3		WirelessFilter	69 Ms	68,064,000	
4		eCall Poll	86 Ms	85,139,000	
5	Detection Unit	IDL_Th	270 Ms	269,100,000	
6		Van_Th	221 Ms	220,030,000	
7		Incident_Th	106 Ms	105,313,000	
8	Processing Unit	Th_WrechWatchServer	96 Ms	95,319,000	
9		Th_eCallHandler	97 Ms	96,026,500	
10		Th_Verification	254 Ms	253,311,500	
11	Output Unit	Routing_Th	90 Ms	89,163,500	
12		ToCloud_Th	96 Ms	95,788,500	

As we see in Table 4.5, CPU speed is higher at 1 GHz with an L2 Cache of 1 MB, which produced execution times between 69 microseconds and 270 microseconds.

Table 4.6. gem5 results for the proposed option 2_1 on ARM environment.

Option 2_1 : Siemens	**CPU specifications**	**Memory specifications**			
	ARM architecture	RAM	Caches		
	MinorCPU (in-order)	DDR4_2400_16x4	dcache	icache	L2 cache
	CPU clock: 800 MHz	Size: 1 GB	32kB	64kB	none
	Stats				
#	**Unit**	**Tasks**	**Simulated Exec Seconds**	**Simulated ticks**	**Notes**
1	Input Unit	VanetFilter	121 Ms	120,464,500	measurements are in Microseconds (Ms)
2		ild_listener	231 Ms	230,994,500	
3		WirelessFilter	116 Ms	115,401,000	
4		eCall Poll	153 Ms	152,321,000	
5	Detection Unit	IDL_Th	784 Ms	783,609,000	
6		Van_Th	527 Ms	526,837,500	
7		Incident_Th	201 Ms	200,937,500	
8	Processing Unit	Th_WrechWatchServer	168 Ms	167,627,500	
9		Th_eCallHandler	173 Ms	172,747,500	
10		Th_Verification	566 Ms	565,167,500	
11	Output Unit	Routing_Th	163 Ms	162,932,500	
12		ToCloud_Th	168 Ms	167,046,000	

From Table 4.6 we see an in-order CPU where its speed is back at 800 GHz without the L2 Cache, which produced execution times variations between 116 microseconds and 201 microseconds. The final option is similar to the one in Table 4.6, but with out-of-order micro-architecture.

Table 4.7. gem5 results for the proposed option 2_2 on ARM environment.

Option 2_2 : Siemens	**CPU specifications**	**Memory specifications**			
	ARM architecture	RAM	Caches		
	DerivO3COU (out-of-order)	DDR4_2400_16x4	dcache	icache	L2 cache
	CPU clock: 800 MHz	Size: 1 GB	32kB	64kB	none
	Stats				
#	**Unit**	**Tasks**	**Simulated Exec Seconds**	**Simulated ticks**	**Notes**
1	Input Unit	VanetFilter	61 Ms	60,343,500	measurements are in Microseconds (Ms)
2		ild_listener	115 Ms	114,862,000	
3		WirelessFilter	60 Ms	59,099,500	
4		eCall Poll	76 Ms	75,839,500	
5	Detection Unit	IDL_Th	356 Ms	355,522,000	
6		Van_Th	223 Ms	222,569,500	
7		Incident_Th	101 Ms	100,473,000	
8	Processing Unit	Th_WrechWatchServer	87 Ms	86,504,000	
9		Th_eCallHandler	88 Ms	87,631,000	
10		Th_Verification	264 Ms	263,272,500	
11	Output Unit	Routing_Th	82 Ms	81,479,000	
12		ToCloud_Th	88 Ms	87,255,000	

4.3. RSU Model's schedulability analysis in OSATE

In addition to systems' modeling, OSATE also provides analyses tests and reports. There are multiple tests to be used such as safety analysis, timing analysis, flow latency and more. Every analysis can be done at a certain level where the right properties are set, the more detailed the model, the more analyses tests could be done. In our case, we are at an abstraction level where we can produce schedulability tests and data flow latency.

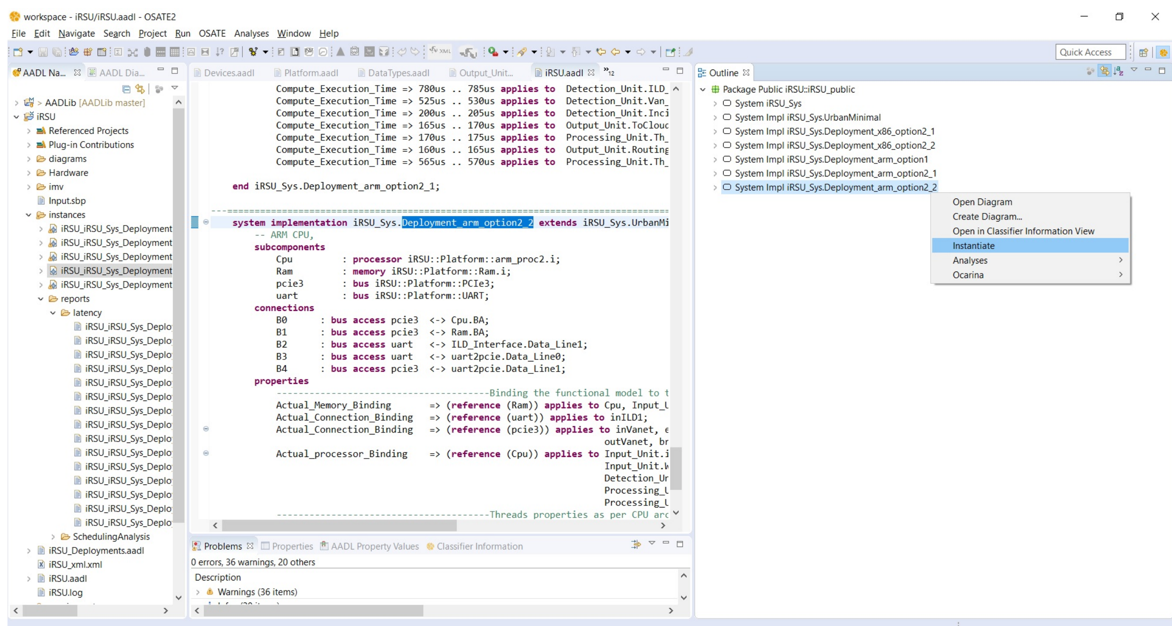

Figure 4.2. System implementation instantiating.

To be able to do any schedulability test, a system implementation should be instantiated first, as shown in Figure 4.2. From the Navigator pane, we can choose the instance that we created and apply the desired test, in this case the schedulability test. For a schedulability test, a scheduling protocol has to be set under the processor component properties. In our model, we used Rate Monotonic Scheduling (RMS). We used the schedulability test on all of the system variations, as shown Figure 4.3, all the variations are schedulable with different utilization percentage. As we see from Figure 4.3, the highest utilization factor are for the X86 environment processors, because of the execution times that was simulated on gem5. Since the period for most

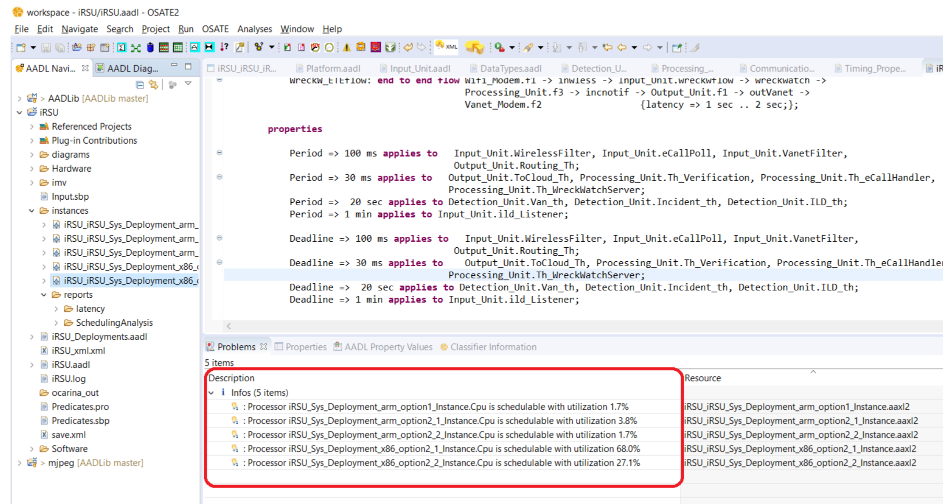

Figure 4.3. System variations' schedulability test results.

of the tasks are at 100 ms, and in option 2_1 the execution times are a little less than 5 ms per task, therefore, we see it takes over half of the processor time. However, the ARM environment provides faster execution times, while the periods stay the same, hence the much less the utilisation's percentages are. This is helpful in making the system lightweight in case it is considered to be included as an added functionality to an existing RSU.

4.4. RSU Model's flow latency analysis in OSATE

The flow latency analysis provides a deeper look on how long it takes the incoming data to be received, processed, and transferred as an output of the system. It captures the system response time. To be able to do this analysis, a flow source, flow path and flow sink should be defined through out the system's components. In the final step, an end-to-end flow should be specified and implemented in the system component, where it details step-by-step from where to where the data is moving, where it starts and where it ends. In addition, for each end-to-end flow implementation, expected latency should be specified under the property latency. In Figure 4.4 we show the four end-to-end flows of our model.

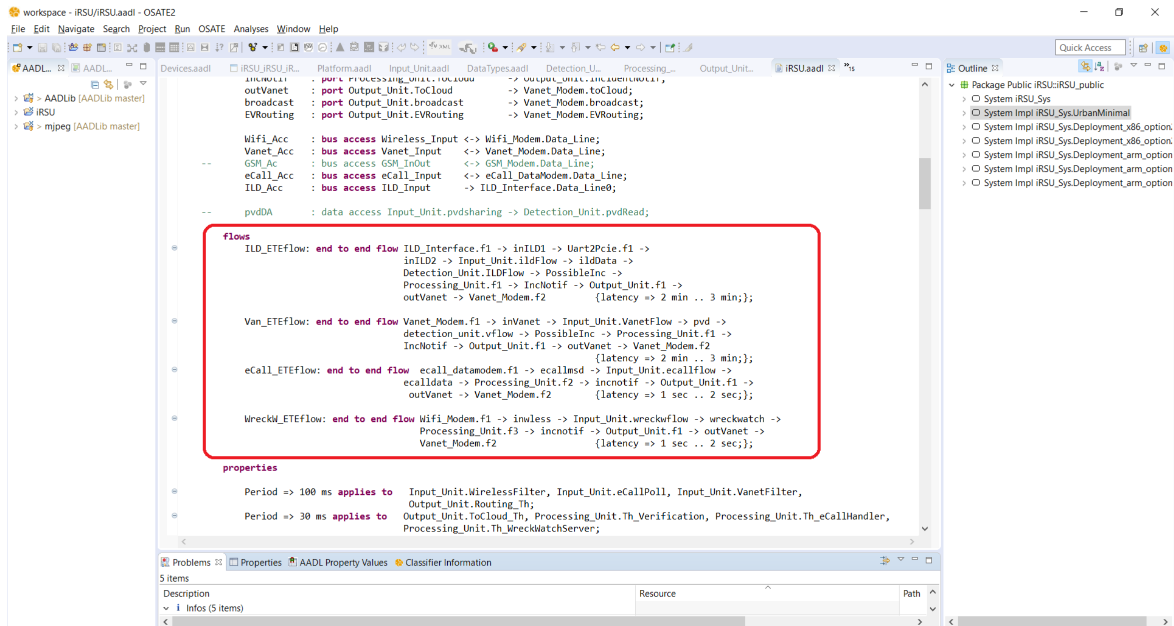

Figure 4.4. End-to-end flow in a system implementation.

Flow latency analysis depends on several factors: tasks' periods, deadlines, execution times, dispatch protocol, scheduling policy, transferred data sizes and bus transfer rate. A flow latency analysis accumulates all the latency contributors' values starting from the flow source, going through the buses and to the flow sink, where the data are finally transferred. We applied the flow latency analysis on all of the system variations, and since each variation has four end-to-end flow, 20 results appeared in the latency reports. Therefore, we are going to show a comparison between the five variations on each end-to-end flow in Table 4.8 and Table 4.9.

Table 4.8. Minimum latency of all end-to-end flows under all platform variations.

Deployment Variations	**ILD end-to-end flow**	**Vanet end-to-end flow**	**eCall end-to-end flow**	**WreckWatch end-to-end flow**
X86 option 2_1	34.3 ms	20.5 ms	16 ms	15.9 ms
X86 option 2_2	21.8 ms	8.17 ms	6.4 ms	6.37 ms
ARM option 1	14.2 ms	0.748 ms	0.535 ms	0.518 ms
ARM option 2_1	15.3 ms	1.59 ms	1.06 ms	1.02 ms
ARM option 2_2	14.3 ms	0.735 ms	0.515 ms	0.500 ms

Table 4.9. Maximum latency of all end-to-end flows under all platform variations.

Deployment Variations	**ILD end-to-end flow**	**Vanet end-to-end flow**	**eCall end-to-end flow**	**WreckWatch end-to-end flow**
X86 option 2_1	60134 ms	20320.5 ms	216 ms	215.9 ms
X86 option 2_2	60121.8 ms	20308.2 ms	206.4 ms	206.4 ms
ARM option 1	60114.2 ms	20300.8 ms	200.6 ms	200.5 ms
ARM option 2_1	60115.3 ms	20301.6 ms	201.1 ms	201.1 ms
ARM option 2_2	60114.3 ms	20300.8 ms	200.8 ms	200.5 ms

As we see in Tables 4.8 and 4.9, for each end-to-end flow, there is a minimum and a maximum actual latency. In the minimum actual latency, OSATE assumes that all the data is ready, and each thread is dispatched immediately without delay, in which it shows us the best case scenario of the system. On the contrary, in the maximum actual latency, OSATE assumes that each thread has to wait for the data to be ready and will be dispatched at the end of the period, in which it shows us the worst case scenario of the system. As shown in Table 4.8, we notice that the difference between each end-to-end flow of each deployment variation is marginal, because the only difference among the latency contributors are the tasks' execution times and task periods are ignored. However, Table 4.8 shows a big difference in the latency of each flow, where the periodic behavior of the tasks affects the system response. We see that the ILD flow has the longest latency due to the one minute period of the ild_Listener. The aim of flow latency analysis is to check whether the system actual latency achieve less than the specified latency. This provides us of the system's response time, in which we notice here that our system response time -in case of an accident- is less than three minutes.

4.5. Scheduling simulations in AADL Inspector

In this section, we provide the results of the scheduling simulations, where we were able to verify that the system meets all deadlines and that all tasks are schedulable. Using AADL Inspector, we loaded our RSU's AADL model files, and under Timing Analysis tab, we applied real time static testing under Cheddar simulator. We chose

to apply the simulation testing on the Siemens option 2_1, because this option has the longest execution times and a processor utilization of 68%. The results of the scheduling simulation test is shown in Figure 4.5. We see that all threads are scheduled according to their periods, and all threads finish before their deadlines.

Figure 4.5. Static scheduling simulation by AADL Inspector.

In addition to static simulations, we provide dynamic scheduling simulations under Marzhin simulator that is embedded in AADL Inspector. In the same Timing Analysis tab, we used run-time simulation to show how and when the RSU threads are dispatched and executed. We can see in Figure 4.6 threads' dispatching and execution timing. When a thread is finished, it dispatches the corresponding next thread according to each thread's scheduling properties. For example, the execution starts with the WirelessFilter thread, once it finishes executing, it dispatches the following thread Th_WreckWatchServer. When the Th_WreckWatchServer finishes, it dispatches the Th_Verification thread, and when the Th_Verification thread finishes, it dispatches the ToCloud_Th thread. Next, the rest of the Input Unit threads will start executing one by one and each will dispatch the corresponding threads. For clarification of color codes in Figure 4.6, the orange color means the thread is ready to execute and awaits its turn, the black color means the thread is executing, and the grey color means it's

busy which is assigned to the data components to show they are being used by their assigned threads.

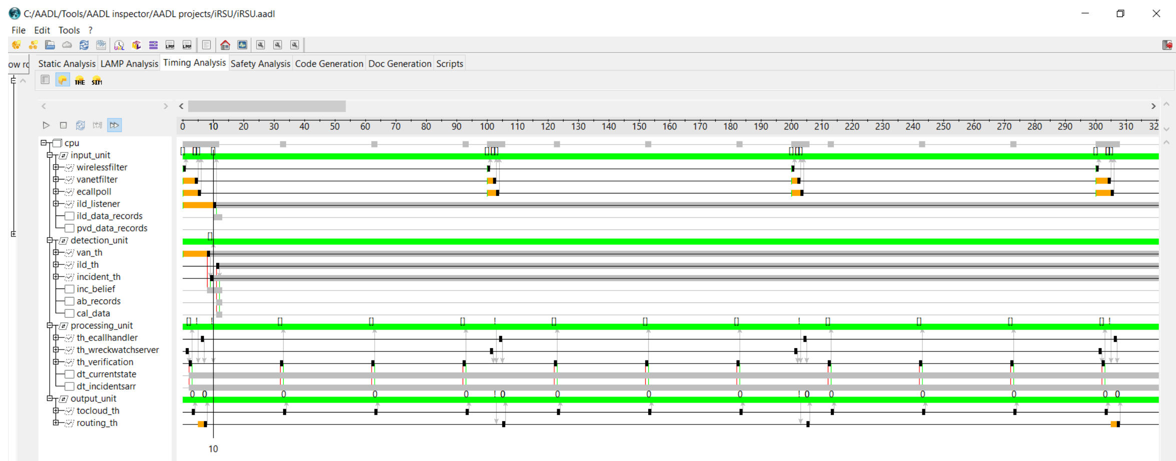

Figure 4.6. Dynamic scheduling simulation by AADL Inspector.

5. CONCLUSION AND FUTURE WORK

In this thesis, we proposed a system that consists of a cloud layer and a Roadside Unit (RSU). The RSU provides incident detection using multiple algorithms and solutions, while the cloud layer is responsible for incident classification and creating routing plans. Moreover, the cloud should convey the routing instructions to all RSUs on the route between the incident site and the nearest medical/emergency institution. We provided an internal architecture of the proposed RSU using AADL modelling language, where we were able to describe our RSU's subcomponents from software to hardware and execution platform components. We defined their characteristics, data types, and connections to each other. In addition, we provided five different options for the execution platform, where we applied scheduling testing and data flow latency analysis and compared the results among these five options. We also provided a real time scheduling simulations under AADL Inspector.

In the end, we see that the proposed RSU is schedulable with different utilization factors, where most of the provided options provide low processor utilization. From that we conclude that the system could be added as a functionality to an existing RSU to provide emergency management along its existing functionalities. We also conclude that, at this abstraction level, the system response time is lower than 3 minutes. Which means that to detect a traffic accident, our system will not take more than 3 minutes to report to the cloud of this accident, hence we meet the motivation goal (to report accidents under five minutes) that was mentioned in the Introduction chapter. We also come to the conclusion that the system still need to be more refined and detailed to allow for more tests to be applied and produce better results. In addition, what we provided at this level of abstraction is flexible to be extended and refined at any way a developer wants.

In the future we plan to refine the model where we choose the best two platform options and continue from there to finish the whole model and provide comparisons.

In addition, we will add system modes, add error and behaviour model extensions, apply safety analysis and produce better simulation results. We want to be able to create a complete emergency and disaster management system, where we can provide this system as an added functionality to an existing RSU and deploy the device into a network to provide smarter and safer roads.

REFERENCES

1. "Global status report on road safety 2018", `https://www.who.int/violence-injury-prevention/road-safety-status/2018/en/`, accessed at Dec 2019.

2. Lerner, E. B. and R. M. Moscati, "The golden hour: scientific fact or medical "urban legend"?", *Academic Emergency Medicine*, Vol. 8, No. 7, pp. 758–760, 2001.

3. Martinez, F. J., C.-K. Toh, J.-C. Cano, C. T. Calafate and P. Manzoni, "Emergency services in future intelligent transportation systems based on vehicular communication networks", *IEEE Intelligent Transportation Systems Magazine*, Vol. 2, No. 2, pp. 6–20, 2010.

4. "Intelligent Transportation System", `https://www.isbak.istanbul/intelligent-transportation-systems/`, accessed at Dec 2019.

5. Niar, S., A. Yurdakul, O. Unsal, T. Tugcu and A. Yuceturk, "A dynamically reconfigurable architecture for emergency and disaster management in ITS", *2014 International Conference on Connected Vehicles and Expo (ICCVE)*, pp. 479–484, Nov 2014.

6. "Siemens Connected Vehicle Roadside Unit (RSU)", `https://www.mobotrex.com/product/siemens-connected-vehicle-roadside-unit/`, accessed at Dec 2019.

7. "NXP, Intelligent Roadside Unit", `https://www.nxp.com/applications/solutions/ automotive/connectivity/intelligent-roadside-unit:INTELLIGENTRSU`, accessed at Dec 2019.

8. "The interoperable EU-wide eCall - European Commission",

https://ec.europa.eu/transport/themes/its/road/action_plan/ecall_en, accessed at Dec 2019.

9. Weil, R., J. Wootton and A. Garcia-Ortiz, "Traffic incident detection: Sensors and algorithms", *Mathematical and computer modelling*, Vol. 27, No. 9-11, pp. 257–291, 1998.

10. White, J., C. Thompson, H. Turner, B. Dougherty and D. C. Schmidt, "Wreckwatch: Automatic traffic accident detection and notification with smartphones", *Mobile Networks and Applications*, Vol. 16, No. 3, pp. 285–303, 2011.

11. "Model Based Systems Engineering", https://modelbasedengineering.com/, accessed at Dec 2019.

12. Gérard, S., H. Espinoza, F. Terrier and B. Selic, "6 Modeling Languages for Real-Time and Embedded Systems", *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, pp. 129–154, Springer, 2007.

13. Feiler, P. H. and D. P. Gluch, *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*, Addison-Wesley, 2012.

14. "OSATE", https://osate.org/, accessed at Dec 2019.

15. Jang, J. A., H. S. Kim and H. B. Cho, "Smart roadside system for driver assistance and safety warnings: Framework and applications", *Sensors*, Vol. 11, No. 8, pp. 7420–7436, 2011.

16. Saputro, N., K. Akkaya, R. Algin and S. Uluagac, "Drone-Assisted Multi-Purpose Roadside Units for Intelligent Transportation Systems", *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, pp. 1–5, IEEE, 2018.

17. Sundar, R., S. Hebbar and V. Golla, "Implementing intelligent traffic control system for congestion control, ambulance clearance, and stolen vehicle detection",

IEEE Sensors Journal, Vol. 15, No. 2, pp. 1109–1113, 2014.

18. Filjar, R., K. Vidović, P. Britvić and M. Rimac, "eCall: Automatic notification of a road traffic accident", *2011 Proceedings of the 34th International Convention MIPRO*, pp. 600–605, IEEE, 2011.

19. Oorni, R. and A. Goulart, "In-vehicle emergency call services: eCall and beyond", *IEEE Communications Magazine*, Vol. 55, No. 1, pp. 159–165, 2017.

20. "EMYNOS project website", `http://www.emynos.eu/`, accessed at Dec 2019.

21. Klein, L. A., M. K. Mills, D. R. Gibson *et al.*, *Traffic detector handbook: Volume I*, Tech. rep., Turner-Fairbank Highway Research Center, 2006.

22. "The Basics of Loop Vehicle Detection" (White paper)", `MarshProducts, 2000,https://marshproducts.com/PDF/Inductive%20Loop%20Write%20up.pdf`, accessed at Dec 2019.

23. Rossi, R., M. Gastaldi, G. Gecchele and V. Barbaro, "Fuzzy logic-based incident detection system using loop detectors data", *Transportation Research Procedia*, Vol. 10, pp. 266–275, 2015.

24. Parkany, E. and C. Xie, *A complete review of incident detection algorithms and their deployment: what works and what doesn't. Transportation Center, University of Massachusetts*, Tech. rep., Technical Report, NETÇR3 7, 2005.

25. "Fuzzy Logic", `https://www.sciencedirect.com/topics/computer-science/fuzzy-logic`, accessed at Dec 2019.

26. "Mamdani's product-sum inference system", `http://researchhubs.com/post/engineering/fuzzy-system/mamdani-fuzzy-model.html`, accessed at Dec 2019.

27. Jindal, V. and P. Bedi, "Vehicular ad-hoc networks: introduction, standards, rout-

ing protocols and challenges", *International Journal of Computer Science Issues (IJCSI)*, Vol. 13, No. 2, p. 44, 2016.

28. Campolo, C. and A. Molinaro, "On vehicle-to-roadside communications in 802.11 p/WAVE VANETs", *2011 IEEE wireless communications and networking conference*, pp. 1010–1015, IEEE, 2011.

29. "Sample of SAE J2735 DSRC Message List", `https://www.use-snip.com/kb/knowledge-base/sae-j2735-dsrc-message-list/`, accessed at Dec 2019.

30. "Official SAE J2735 DSRC Message List", `https://www.sae.org/standards/development/dsrc`, accessed at Dec 2019.

31. Popescu, O., S. Sha-Mohammad, H. Abdel-Wahab, D. C. Popescu and S. El-Tawab, "Automatic incident detection in intelligent transportation systems using aggregation of traffic parameters collected through V2I communications", *IEEE Intelligent Transportation Systems Magazine*, Vol. 9, No. 2, pp. 64–75, 2017.

32. Nguyen, H., C. Cai and F. Chen, "Automatic classification of traffic incident's severity using machine learning approaches", *IET Intelligent Transport Systems*, Vol. 11, No. 10, pp. 615–623, 2017.

33. "Systems Modeling Language, SysML", `http://www.omgsysml.org/`, accessed at Dec 2019.

34. Mallet, F. and R. De Simone, "MARTE: a profile for RT/E systems modeling, analysis–and simulation?", *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, p. 43, 2008.

35. Delange, J., *AADL IN PRACTICE*, Reblochon Development Company, 2017.

36. "COMPASS", http://www.compass-toolset.org/, accessed at Dec 2019.

37. "Ocarina", http://ocarina.readthedocs.io/en/latest/index.html, accessed at Dec 2019.

38. Singhoff, F., J. Legrand, L. Nana and L. Marcé, "Cheddar: a flexible real time scheduling framework", *ACM SIGAda Ada Letters*, Vol. 24, pp. 1–8, ACM, 2004.

39. "AADL Inspector", https://www.ellidiss.com/products/aadl-inspector/, accessed at Dec 2019.

40. "RS485 Connection", http://www.bb-elec.com/Learning-Center/All-White-Papers/Serial/RS-485-Connections-FAQ.aspx, accessed at Dec 2019.

41. "RS485 to UART Converter", http://qqtrading.com.my/rs485-uart-serial-converter-module, accessed at Dec 2019.

42. "UART to PCIe converter", https://www.siig.com/2-port-rs232-serial-pcie-with-16950-uart.html, accessed at Dec 2019.

43. "Gem5 Simulator", http://gem5.org/Main_Page, accessed at Dec 2019.

APPENDIX A: AADL MODEL SCRIPTS

In this appendix, we present our AADL modeling scritps for all components used in the proposed RSU. In OSATE, we separated the components of our system into multiple files, each containing specific component types for easy access and modification. As shown below, we separated the hardware components into the platform that consist of the computational core such as the processor, memory and bus, while the devices are the communication hardware used to connect to the outside world. The software components were divided into four units: Input, Detection, Processing and Output, in addition to associated Data types. Finally we present the system file (iRSU.aadl) which integrates all of these components into system implementations.

A.1. Software Components

A.1.1. Input Unit

```
package iRSU::Input_Unit
public
with iRSU::DataTypes;
--------------------Wireless Data filter------------------------
thread Wireless_DataFilter
features
  input_Data  : in data port iRSU::DataTypes::wirelessMsg.i;
  WreckWatch_Data  : out event data port iRSU::DataTypes::WWreckWatch_input.i;
flows
  f1 : flow path input_Data -> WreckWatch_Data;
properties
  Dispatch_Protocol => periodic;
  Priority => 1;
end Wireless_DataFilter;

--------------------Vanet Data filter------------------------
thread Vanet_DataFilter
features
  input_Data  : in data port iRSU::DataTypes::vehicleData.i;
  ProbVeh_Data  : out data port iRSU::DataTypes::PVD.i ;
  Cloud_Data  : out event data port iRSU::DataTypes::Routing_data.i;
  Clearing_Data  : out event data port iRSU::DataTypes::clearing_data.i;
```

```
flows
  f1 : flow path input_Data -> ProbVeh_Data;
  f2 : flow path input_Data -> Cloud_Data;
  f3 : flow path input_Data -> Clearing_Data;

properties
  Dispatch_Protocol => periodic;
  Priority => 2;
end Vanet_DataFilter;

--------------------Vanet Data filter------------------------
thread ild_inputListener
features
input_Data  : in data port iRSU::DataTypes::ILD_input.i;
ild_data : out event data port iRSU::DataTypes::ILD_data.i;
ild_data_records: requires data access iRSU::DataTypes::ILD_data.i
{access_right => Read_write; };
flows
f1: flow path input_Data -> ild_data;
properties
Dispatch_Protocol => periodic;
   Priority => 2;
end ild_inputListener;
--------------------eCall polling thread------------------------
thread eCall_Listner
features
   input_Data  : in data port iRSU::DataTypes::eCall_MSD.i;
   eCall_MSD  : out event data port iRSU::DataTypes::eCall_MSD.i;
flows
   f1 : flow path input_Data -> eCall_MSD;
properties
     Dispatch_Protocol => periodic;
   Priority => 1;
end eCall_Listner;

----------------Input process Definition-----------------------
process input_process
features
Vanet_input  : in data port iRSU::DataTypes::vehicleData.i;
Wireless_input  : in data port iRSU::DataTypes::wirelessMsg.i;
eCall_input  : in data port iRSU::DataTypes::eCall_MSD.i;
ild_in : in data port iRSU::DataTypes::ILD_input.i;
--ProbVeh_Data  : out data port iRSU::DataTypes::PVD.i;
ild_data : out event data port iRSU::DataTypes::ILD_data.i;
eCall_MSD  : out event data port iRSU::DataTypes::eCall_MSD.i;
Wreckwatch_Data : out event data port iRSU::DataTypes::WWreckWatch_input.i;
Cloud_Data  : out event data port iRSU::DataTypes::Routing_data.i;
```

```
Clearing_Data : out event data port iRSU::DataTypes::clearing_data.i;
pvdSharing : provides data access iRSU::DataTypes::PVD.i
{access_right => read_write; };
flows
  WreckWFlow  : flow path Wireless_input  -> Wreckwatch_Data;
  CloudFlow  : flow path Vanet_input -> Cloud_Data;
  ildFlow  : flow path ild_in  -> ild_data;
  eCallFlow  : flow path eCall_input  -> eCall_MSD;
  VanetFlow  : flow path Vanet_input -> pvdSharing;
end input_process;

--------------Input process Implementation----------------------
process implementation input_process.i
subcomponents
  WirelessFilter : thread Wireless_DataFilter;
  VanetFilter  : thread Vanet_DataFilter;
  eCallPoll  : thread eCall_Listner;
  ild_Listener  : thread ild_inputListener;
  ild_data_records : data iRSU::DataTypes::ILD_data.i
   {Concurrency_Control_Protocol => Priority_Ceiling;};
  pvd_data_records : data iRSU::DataTypes::PVD.i
   {Concurrency_Control_Protocol => Priority_Ceiling;};
connections
  c1 : port Vanet_input  -> VanetFilter.input_Data;
  c2 : port Wireless_input  -> WirelessFilter.input_Data;
  c3 : port Wirelessfilter.WreckWatch_Data  -> Wreckwatch_Data;
  c4 : port VanetFilter.Cloud_Data  -> Cloud_Data;
  c5 : port VanetFilter.ProbVeh_Data  -> pvd_data_records;
--  c12: port pvd_data_records -> ProbVeh_Data;
  c6 : port ild_in  -> ild_Listener.input_Data ;
  c7 : port eCall_input  -> eCallPoll.input_Data;
  c8 : port eCallPoll.eCall_MSD  -> eCall_MSD;
  c9 : port ild_Listener.ild_data  -> ild_data;
  c10: port VanetFilter.Clearing_Data -> Clearing_Data;
  c11: data access ild_data_records <-> ild_Listener.ild_data_records;
--  c12: data access pvd_data_records <-> VanetFilter.pvd_data_records;
  c13: data access pvd_data_records -> pvdSharing;
flows
  WreckWFlow : flow path wireless_input -> c2 -> wirelessfilter.f1-> c3 -> wreckwatch_data;
  CloudFlow  : flow path Vanet_input -> c1 -> VanetFilter.f2   -> c4 -> cloud_data;
  ildFlow   : flow path ild_in -> c6 -> ild_Listener.f1  -> c9 -> ild_data;
  VanetFlow  : flow path Vanet_input-> c1 -> VanetFilter.f1   -> c5 -> pvd_data_records
   -> c13 -> pvdSharing;
  eCallFlow  : flow path ecall_input -> c7 -> ecallpoll.f1   -> c8 -> ecall_msd;
end input_process.i;

end iRSU::Input_Unit;
```

A.1.2. Detection Unit

```
package iRSU::Detection_Unit
public
  with iRSU::DataTypes;
--==========================================================================================================
---------------------------------------- Detection Algorithms -----------------------------------------
--==========================================================================================================
--------------------Fusion of all inputs-------------------------
thread Incident_th
features
  From_ildAlg  : in event data port iRSU::DataTypes::ILD_IncInfo.i;
  From_VanetAlg : in event data port iRSU::DataTypes::Van_IncInfo.i;
  IncidentAlarm : out event data port iRSU::DataTypes::PossibleInc.i;
flows
  f1 : flow path From_VanetAlg  -> IncidentAlarm;
  f2 : flow path From_ILDAlg  -> IncidentAlarm;
properties
  Dispatch_Protocol => Sporadic;
  Dispatch_Trigger => (reference(From_ildAlg), reference(From_VanetAlg));
  Priority => 1;
end Incident_th;

--------------------Vehicle Data Algorithm-------------------------
thread Vanet_incidentDetection
features
  input : in data port iRSU::DataTypes::PVD.i;
  output : out event data port iRSU::DataTypes::Van_IncInfo.i;
  incident_Belief : requires data access iRSU::DataTypes::incident_belief
   {access_right => Read_write; };
flows
  f1 : flow path input -> output;
properties
  Dispatch_Protocol => Periodic;
  Priority => 1;
end Vanet_incidentDetection;
--------------------Sensor Data Algorithm-------------------------
thread ILD_incidentDetection
features
  input   : in event data port iRSU::DataTypes::ILD_data.i;
  output  : out event data port iRSU::DataTypes::ILD_IncInfo.i;
  ab_records : requires data access iRSU::DataTypes::Ab_records.i
```

```
   {access_right => Read_write; };
  cal_data : requires data access iRSU::DataTypes::cal_data.i
   {access_right => Read_write; };
flows
  f1 : flow path input -> output;
properties
  Dispatch_Protocol => Sporadic;
  Dispatch_Trigger => (reference(input));
  Priority => 1;
end ILD_incidentDetection;

----------------Input Process Definition-----------------------
process Detection_process
features
  inPVD_Data  : requires data access iRSU::DataTypes::PVD.i
   {access_right => read_only;};
  inILD_Data  : in event data port iRSU::DataTypes::ILD_data.i;
  IncidentAlarm : out event data port iRSU::DataTypes::PossibleInc.i;
flows
  VFlow  : flow path inPVD_Data -> IncidentAlarm;
  ILDFlow  : flow path inILD_Data -> IncidentAlarm;
end Detection_process;
--------------Input Process Implementation-----------------------
process implementation Detection_process.i
subcomponents
  Van_th  : thread Vanet_incidentDetection;
  ILD_th  : thread ILD_incidentDetection;
  Incident_th  : thread Incident_th;
  inc_belief : data iRSU::DataTypes::incident_belief
   {Concurrency_Control_Protocol => Priority_Ceiling;};
  ab_records : data iRSU::DataTypes::Ab_records.i
   {Concurrency_Control_Protocol => Priority_Ceiling;};
  cal_data : data iRSU::DataTypes::cal_data.i
   {Concurrency_Control_Protocol => Priority_Ceiling;};
connections
  c1 : port inPVD_Data -> Van_th.input;
  c2 : port inILD_Data -> ILD_th.input;
  c3 : port Van_th.output -> Incident_th.From_VanetAlg ;
  c4 : port ILD_th.output -> Incident_th.From_ILDAlg;
  c5 : port Incident_th.IncidentAlarm -> IncidentAlarm;
  c6 : data access inc_belief  <-> Van_th.incident_Belief;
  c7 : data access ab_records   <-> ILD_th.ab_records;
  c8 : data access cal_data    <-> ILD_th.cal_data;
flows
  VFlow : flow path inpvd_data-> c1 -> van_th.f1 -> c3 ->
   incident_th.f1 -> c5 -> incidentalarm ;
  ILDFlow: flow path inild_data-> c2 -> ild_th.f1 -> c4 ->
```

```
   incident_th.f2 -> c5 -> incidentalarm ;
end Detection_process.i;

end iRSU::Detection_Unit;
```

A.1.3. Processing Unit

```
package iRSU::processing_Unit
public
with iRSU::dataTypes;
--====================================================================================================
---------------------------------------------Main processing------------------------------------------
--====================================================================================================
--------------------eCall interrupt------------------------
thread eCall_Handler
features
ecallSignal : in event data port iRSU::DataTypes::eCall_MSD.i;
incidentAlarm : out event data port iRSU::DataTypes::ecall_IncInfo.i;
flows
f1: flow path ecallSignal -> incidentAlarm;
properties
Dispatch_Protocol => Sporadic;
Dispatch_Trigger => (reference(ecallSignal));
Priority => 1;
Compute_Deadline => 20 ms;
end eCall_Handler;
--------------------WreckWatch interrupt------------------------
thread WreckWatch_Server
features
MobileincSignal : in event data port iRSU::DataTypes::WWreckWatch_input.i;
incidentAlarm : out event data port iRSU::DataTypes::WWreckWatch_IncInfo.i;
flowS
f1: flow path MobileincSignal -> incidentAlarm;
properties
Dispatch_Protocol => Sporadic;
Dispatch_Trigger => (reference(MobileincSignal));
Priority => 1;
Compute_Deadline => 20 ms;
end WreckWatch_Server;
--------------------incident Verification------------------------
thread incident_Verification
features
CheckState   : requires data access iRSU::dataTypes::status.i
```

```
{access_right => Read_write; };
incidents_records : requires data access iRSU::dataTypes::incidents_records.i
{access_right => read_write; };
incdnt_Alarm   : in event data port iRSU::DataTypes::PossibleInc.i;
eCallAlarm   : in event data port iRSU::DataTypes::ecall_IncInfo.i;
WreckWatchAlarm   : in event data port iRSU::DataTypes::WWreckWatch_IncInfo.i;
ClearingAlarm   : in event data port iRSU::DataTypes::clearing_data.i;
incdnt_info   : out event data port iRSU::DataTypes::Incident_Info.i;
flows
f1: flow path incdnt_Alarm  -> incdnt_info;
f2: flow path eCallAlarm  -> incdnt_info;
f3: flow path WreckWatchAlarm  -> incdnt_info;
properties
Dispatch_Protocol => Sporadic;
Dispatch_Trigger => (reference(incdnt_Alarm), reference (eCallAlarm),
reference (WreckWatchAlarm), reference(ClearingAlarm));
Priority => 1;
Compute_Deadline => 50 ms;
end incident_Verification;

-----------------------Main process Definition-----------------------
process mainprocess
features
incdntAlarm  : in event data port iRSU::DataTypes::PossibleInc.i;
eCallevent  : in event data port iRSU::DataTypes::eCall_MSD.i;
WWatchevent  : in event data port iRSU::DataTypes::WWreckWatch_input.i;
Clearingevent: in event data port iRSU::DataTypes::clearing_data.i;
ToCloud  : out event data port iRSU::DataTypes::Incident_Info.i;
flows
f1: flow path incdntAlarm -> ToCloud;
f2: flow path eCallevent -> ToCloud;
f3: flow path WWatchevent -> ToCloud;
end mainprocess;

-----------------------Main process implementation-----------------------
process implementation mainprocess.i
subcomponents
Dt_CurrentState : data iRSU::dataTypes::status.i
{Concurrency_Control_Protocol =>  Priority_Ceiling; };
Dt_incidentsArr : data iRSU::dataTypes::incidents_records.i
{Concurrency_Control_Protocol => Priority_Ceiling; };
Th_eCallHandler : thread eCall_Handler;
Th_WreckWatchServer : thread WreckWatch_Server;
Th_Verification : thread incident_Verification;
connections
c1: port incdntAlarm  -> Th_Verification.incdnt_Alarm;
c2: port eCallevent  -> Th_eCallHandler.ecallSignal;
```

```
c3: port WWatchevent  -> Th_WreckWatchServer.MobileincSignal;
c4: port Clearingevent  -> Th_Verification.ClearingAlarm;
c5: port Th_eCallHandler.incidentAlarm   -> Th_Verification.eCallAlarm;
c6: port Th_WreckWatchServer.incidentAlarm  -> Th_Verification.WreckWatchAlarm;
c7: port Th_Verification.incdnt_info  -> ToCloud;
c8: data access Dt_CurrentState  <-> Th_Verification.CheckState;
c9: data access Dt_incidentsArr  <-> Th_Verification.incidents_records;
flowS
f1: flow path incdntalarm  -> c1 -> Th_Verification.f1    -> c7 -> toCloud;
f2: flow path ecallevent   -> c2 -> Th_eCallHandler.f1    -> c5 ->
Th_Verification.f2 -> c7 -> toCloud;
f3: flow path wwatchevent  -> c3 -> Th_WreckWatchServer.f1 -> c6 ->
Th_Verification.f3 -> c7 -> toCloud;
end mainprocess.i;
end iRSU::processing_Unit;
```

A.1.4. Output Unit

```
package iRSU::Output_Unit
public
with iRSU::DataTypes;

--======================================================================================
-----------------------------------------OUTPUT PROCESSING------------------------------
--======================================================================================
thread ToCloud
features
IncidentNotif: in event data port iRSU::DataTypes::Incident_Info.i;
ToCloud: out data port iRSU::DataTypes::Incident_Info.i;
flows
f1: flow path IncidentNotif -> ToCloud;
properties
Dispatch_Protocol => Sporadic;
Dispatch_Trigger => (reference (IncidentNotif));
Priority => 1;
Compute_Deadline => 20 ms;
--  reference_processor => classifier(iRSU::Platform::proc.i);
end ToCloud;

--
-- -------------------EV Routing and Evacuation-----------------------
thread Routing
features
Instructions: in event data port iRSU::DataTypes::Routing_data.i;
EVRouting: out data port iRSU::DataTypes::EV_Routing.i;
```

```
broadcast: out  data port iRSU::DataTypes::broadcast.i;
flows
f1: flow path Instructions -> EVRouting;
f2: flow path Instructions -> broadcast;
properties
Dispatch_Protocol => Sporadic;
Dispatch_Trigger => (reference (Instructions));
Priority => 1;
Compute_Deadline => 20 ms;
--  Reference_Processor => classifier(iRSU::Platform::proc.i);
end Routing;

process Output_Unit
features
EVRouting: out data port iRSU::DataTypes::EV_Routing.i;
FromCloud: in event data port iRSU::DataTypes::Routing_data.i;
IncidentNotif: in event data port iRSU::DataTypes::Incident_Info.i;
broadcast: out data port iRSU::DataTypes::broadcast.i;
ToCloud: out data port iRSU::DataTypes::Incident_Info.i;
flows
f1: flow path IncidentNotif -> ToCloud;
END Output_Unit;

-- ------------------------------------------------------
process implementation Output_Unit.i
subcomponents
ToCloud_Th: thread ToCloud;
Routing_Th: thread Routing ;
connections
c0: port IncidentNotif -> ToCloud_Th.IncidentNotif;
c1: port ToCloud_Th.ToCloud -> ToCloud;
c2: port FromCloud -> Routing_Th.Instructions;
c3: port Routing_Th.broadcast -> broadcast;
c4: port Routing_Th.EVRouting -> EVRouting;
flows
f1: flow path IncidentNotif -> c0 -> ToCloud_Th.f1 -> c1 -> ToCloud;
end Output_Unit.i;

end iRSU::Output_Unit;
```

A.1.5. Data Types

```
package iRSU::DataTypes
public
with Base_Types;
--=============================================================================
------------------------------------Data Types--------------------------------
--=============================================================================

data string extends Base_Types::String
properties
Data_size => 24 Bytes; --according to c++ sizeof function
end string;

data GpsLocation
end GpsLocation;

data implementation GpsLocation.i
subcomponents
Latitude :  data Base_Types::Float_32;
Longitude :  data Base_Types::Float_32;
properties
Data_size => 8 Bytes;
end GpsLocation.i;

data TimeStamp
end TimeStamp;

data implementation TimeStamp.i
subcomponents
hour  : data Base_Types::Unsigned_16;
minute : data Base_Types::Unsigned_16;
sec : data Base_Types::Unsigned_16;
dd : data Base_Types::Unsigned_16;
mm : data Base_Types::Unsigned_16;
yyyy : data Base_Types::Unsigned_16;
properties
Data_size => 12 Bytes;
end TimeStamp.i;


data Ab_records
-- records of all anomalies that happened the previous execution
end Ab_records;
data implementation Ab_records.i
```

```
subcomponents
Anom_record  : data Base_Types::Unsigned_8 [2];
--two numbers, section number and lane number
properties
Data_size => 2 Bytes;
end Ab_records.i;

data ILD_input
end ILD_input;

data implementation ILD_input.i -- per loop detector
subcomponents
-- we used integers, because the density
--and volume could be have large numbers
TrafficData : data Base_Types::Integer[4];
properties
Data_size => 16 Bytes;--1000 bits per minute
end ILD_input.i;

data cal_data
end cal_data;

data implementation cal_data.i
subcomponents
Ts : data TimeStamp.i;
TrafficData : data ILD_input.i [22];
properties
Data_size => 364 Bytes;
end cal_data.i;

data ILD_data
end ILD_data;

data implementation ILD_data.i
subcomponents
Ts : data TimeStamp.i;
TrafficData : data ILD_input.i [22];
properties
Data_size => 364 Bytes;
end ILD_data.i;

data eCall_MSD
end eCall_MSD;

data implementation eCall_MSD.i
subcomponents
ecall_msgNo : data string;
```

```
lane_no : data Base_Types::Unsigned_8;
vehicle_type : data string;
     vehicle_plate : data string;
        numOfVictims : data Base_Types::Unsigned_8;
     Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
    properties
Data_Size => 94 Bytes;
--the maximum as per eCall standard is 140 Bytes
end eCall_MSD.i;

data wirelessMsg
end wirelessMsg;
data implementation wirelessMsg.i
properties
Data_size => 92 Bytes;
end wirelessMsg.i;

data incident_belief extends Base_Types::Unsigned_8
properties
Data_Size => 1 Bytes;
end incident_belief;

data vehicleData
end vehicleData;

data implementation  vehicleData.i -- has to be rechecked
subcomponents
Record_no : data string;
    Lane_no : data Base_Types::Unsigned_8;
    Prev_lane_no : data Base_Types::Unsigned_8;
    TimetoChangeLanes : data Base_Types::Integer_16;
    ChangingLane_angle : data Base_Types::Float_32;
    Location : data GpsLocation.i;
    Timing : data TimeStamp.i;
properties
Data_Size => 52 Bytes;
end vehicleData.i;

data PVD
end  PVD;

data implementation PVD.i
subcomponents
--timeStamp  : data TimeStamp.i;
Vehicle_Data : data vehicleData.i [24];
properties
```

```
Data_size => 1248 Bytes;
end PVD.i;

data WWreckWatch_input
end WWreckWatch_input;

data implementation WWreckWatch_input.i
subcomponents
WW_message_no : data string;
     Type_sender : data string;
     Sender_IDno : data string;
     Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
    properties
Data_Size => 92 Bytes ;
end WWreckWatch_input.i;

data clearing_data
end clearing_data;

data Implementation clearing_data.i
 subcomponents
  Cloud_message_no : data string;
     IncdntRecord_id : data string;
     lane_no : data Base_Types::Unsigned_8;
     Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
     Cleared : data Base_Types::Boolean;
    properties
Data_size => 69 Bytes;
end clearing_data.i;

data Routing_data
end Routing_data;

data implementation Routing_data.i
subcomponents
Cloud_message_no : data string;
     Message_type : data Base_Types::Unsigned_8;
     lane_no : data Base_Types::Unsigned_8;
     Message_detail : data string;
    properties
Data_Size => 50 Bytes;
end Routing_data.i;

data ILD_IncInfo
end ILD_IncInfo;
```

```
data implementation ILD_IncInfo.i
subcomponents
ILD_MsgNo : data string;
LaneNo : data Base_Types::Unsigned_8;
Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
     properties
Data_Size => 45 Bytes;
end ILD_IncInfo.i;

data Van_IncInfo
end Van_IncInfo;

data Implementation Van_IncInfo.i
subcomponents
Van_MsgNo : data string;
LaneNo : data Base_Types::Unsigned_8;
Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
    properties
Data_Size => 45 Bytes;
end Van_IncInfo.i;

data PossibleInc
end PossibleInc;

data Implementation PossibleInc.i
subcomponents
PossibleInc_no : data string;
     detector : data Base_Types::Unsigned_8;
     lane_no : data Base_Types::Unsigned_8;
     Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
    properties
Data_Size => 46 Bytes ;
end PossibleInc.i;

data WWreckWatch_IncInfo
end WWreckWatch_IncInfo;

data Implementation WWreckWatch_IncInfo.i
subcomponents
WW_message_no : data string;
     Type_sender : data string;
     Sender_IDno : data string;
     numOfVehicles : data Base_Types::Unsigned_8;
```

```
     numOfVictims : data Base_Types::Unsigned_8;
     Lane_no : data Base_Types::Unsigned_8;
    Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
    properties
Data_Size => 95 Bytes ;
end WWreckWatch_IncInfo.i;

data ecall_IncInfo
end ecall_IncInfo;

data implementation ecall_IncInfo.i
subcomponents
ecall_message_no : data string;
     lane_no : data Base_Types::Unsigned_8;
     vehicle_type : data string;
     vehicle_plate : data string;
     numOfVictims : data Base_Types::Unsigned_8;
    Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
    properties
Data_Size => 94 Bytes;
end ecall_IncInfo.i;

data Incident_Info
end Incident_Info;

data implementation Incident_Info.i
subcomponents
incdnt_no : data string;
     incdnt_detectors  : data string;
     Lane_no : data Base_Types::Unsigned_8;
     numOfVehicles : data Base_Types::Unsigned_8;
     vehicle_plates : data string;
     vehicle_types : data string;
     numOfVictims : data Base_Types::Unsigned_8;
     victim_ID : data string;
     Acc_spot : data GpsLocation.i;
     TimeofAcc : data TimeStamp.i;
     details : data string;
    properties
Data_Size => 167 Bytes;
end Incident_Info.i;

data EV_Routing
end EV_Routing;
```

```
data implementation EV_Routing.i
subcomponents
CloudMsgNo : data string;
MessageType : data Base_Types::Unsigned_8;
Lane_no : data Base_Types::Unsigned_8;
Msg_details : data string;
 properties
Data_Size => 50 Bytes;
end EV_Routing.i;

data broadcast
end broadcast;

data implementation broadcast.i
subcomponents
CloudMsgNo : data string;
Lane_no : data Base_Types::Unsigned_8;
Msg_details : data string;
 properties
Data_Size => 49 Bytes ;
end broadcast.i;

data status
end status;

data implementation status.i
subcomponents
currentState: data Base_Types::Unsigned_8;
numOfIncdt : data Base_Types::Unsigned_8;
properties
Data_Size => 2 Bytes ;
end status.i;

data Incident_RecordInfo extends Incident_Info
end Incident_RecordInfo;

data implementation Incident_RecordInfo.i extends Incident_Info.i
subcomponents
confirmed  : data Base_Types::Boolean; --- short
incidents_record: data Incident_Info;
properties
Data_Size => 167 Bytes ;
end Incident_RecordInfo.i;

data incidents_records
end incidents_records;
```

```
data implementation incidents_records.i --- Array of current possible incidents
subcomponents
incidentsRecord : data Incident_RecordInfo.i[10];
properties
Data_Size => 1670 Bytes ;
end incidents_records.i;


end iRSU::DataTypes;
```

A.2. Hardware Components

A.2.1. Devices

```
package iRSU::Devices
public
with iRSU::Platform;
with iRSU::DataTypes;
----------------------------------DEVICES------------------------------------
device WirelessModem ---IEEE 802.11ah tech, 2.4 Ghz
features
Data_Line : requires bus access iRSU::Platform::WirelessAntenna;
incoming_Data : out data port iRSU::DataTypes::wirelessMsg.i;
flows
f1: flow source incoming_Data;
properties
Data_Rate => 300000000 bitsps;--300 Mbps
end WirelessModem;

---------------------------------------------------

device VanetTranseiver --DSRC device communicates with passing vehicles
features
Data_Line : requires bus access iRSU::Platform::VanetAntenna;
ethernet : requires bus access iRSU::Platform::Ethernet;
EVRouting : in data port iRSU::DataTypes::EV_Routing.i;
broadcast : in event data port iRSU::DataTypes::broadcast.i;
toCloud : in event data port iRSU::DataTypes::Incident_Info.i;
VanetMsgs : out data port iRSU::DataTypes::vehicleData.i;
Flows
f1: flow source VanetMsgs;
f2: flow sink  toCloud;
properties
```

```
Data_Rate => 3000000 bitsps;--3 Mbps
Period => 100ms;
--compute_execution_time => 50 ms .. 50 ms;
end VanetTranseiver;
---------------------------------------------------
-- ILD_interface : rs485 to uart converter
-- http://qqtrading.com.my/rs485-uart-serial-converter-module
device ILD_Interface
features
Data_Line0 : requires bus access iRSU::Platform::rs485;
Data_Line1 : requires bus access iRSU::Platform::UART;
ILD_Data : out data port iRSU::DataTypes::ILD_input.i;
Flows
f1: flow source ILD_Data;
end ILD_Interface;

-- UART to Pcie interface
-- https://www.siig.com/2-port-rs232-serial-pcie-with-16950-uart.html

device UART2PCIe
  features
  Data_Line0  : requires bus access iRSU::Platform::UART;
Data_Line1  : requires bus access iRSU::Platform::PCIe3;
ILD_Data_in  : in data port iRSU::DataTypes::ILD_input.i;
ILD_Data_out : out data port iRSU::DataTypes::ILD_input.i;
Flows
f1: flow path ILD_Data_in -> ILD_Data_out;
  properties
  Data_Rate => 921400 bitsps;
 end UART2PCIe;
---------------------------------------------------
device ecall_DataModem
features
Data_Line: requires bus access iRSU::Platform::GSM_Antenna;
inMSD: out data port iRSU::DataTypes::eCall_MSD.i;
Flows
f1: flow source inMSD;
-- data rate is not certain
end ecall_DataModem;

---------------------------------------------------
device GPS
end GPS;

end iRSU::Devices;
```

A.2.2. Platform

```
package iRSU::Platform
public
with SEI;
with Deployment;
with Processor_Properties;
with Bus_Properties;
with Cheddar_Multicore_Properties;
--=============================================================================
--------------------------------Processing HW----------------------------------
--=============================================================================
memory iCache
end iCache;

memory implementation iCache.i
properties
Cheddar_Multicore_Properties::Cache_Level => 1;
Cheddar_Multicore_Properties::Cache_Type => Instruction_Cache;
Memory_Size => 64 KByte;
end iCache.i;

memory dCache
end dCache;

memory implementation dCache.i
properties
Cheddar_Multicore_Properties::Cache_Level => 1;
Cheddar_Multicore_Properties::Cache_Type => Data_Cache;
Memory_Size => 32 KByte;
end dCache.i;

memory L2_Cache
features
BA1: requires bus access PCIe3;
properties
Cheddar_Multicore_Properties::Cache_Level => 2;
Memory_Size => 1 MByte;
end L2_Cache;

processor basicProc
features
BA: requires bus access PCIe3;
end basicProc;

processor implementation basicProc.i
```

```
end basicProc.i;

processor x86proc extends basicProc
properties
SEI::MIPSCapacity => 800.0 MIPS;
Allowed_Dispatch_Protocol => (Periodic, Sporadic, Aperiodic);
end x86proc;

processor implementation x86proc.i  extends basicProc.i
subcomponents
icache: memory iCache.i;
dcache: memory dCache.i;
properties
Processor_Properties::Processor_Family => x86;
Processor_Properties::Endianess => Little_Endian;
Processor_Properties::Word_Length => 64 bits;
Processor_Properties::Processor_Frequency => 800 MHz;
Scheduling_Protocol => (RMS);
Deployment::Execution_Platform => Native;
end x86proc.i;


processor arm_proc1 extends basicProc-- option 1
properties
SEI::MIPSCapacity => 1000.0 MIPS;
Allowed_Dispatch_Protocol => (Periodic, Sporadic, Aperiodic);
end arm_proc1;

processor implementation arm_proc1.i  extends basicProc.i-- option 1 arm
subcomponents
icache: memory iCache.i;
dcache: memory dCache.i;
--l2Cache: memory L2_Cache;
properties
Processor_Properties::Processor_Family => ARM;
Processor_Properties::Endianess => Little_Endian;
Processor_Properties::Word_Length => 64 bits;
Processor_Properties::Processor_Frequency => 1000 MHz;
Scheduling_Protocol => (RMS);
Deployment::Execution_Platform => Native;
end arm_proc1.i;


processor arm_proc2 extends basicProc
properties
SEI::MIPSCapacity => 800.0 MIPS;
Allowed_Dispatch_Protocol => (Periodic, Sporadic, Aperiodic);
```

```
end arm_proc2;

processor implementation arm_proc2.i  extends basicProc.i
subcomponents
icache: memory iCache.i;
dcache: memory dCache.i;
properties
Processor_Properties::Processor_Family => ARM;
Processor_Properties::Endianess => Little_Endian;
Processor_Properties::Word_Length => 64 bits;
Processor_Properties::Processor_Frequency => 800 MHz;
Scheduling_Protocol => (RMS);
Deployment::Execution_Platform => ARM_N770;
end arm_proc2.i;

virtual processor SporadicSH
properties
Allowed_Dispatch_Protocol => (Aperiodic, Sporadic);
end SporadicSH;

virtual processor Implementation SporadicSH.i
properties
Scheduling_Protocol => (RMS);
period => 10 sec;
Execution_Time => 3 sec;
Dispatch_Protocol => periodic;
end SporadicSH.i;


Memory Ram
features
BA: requires bus access PCIe3;
end Ram;

Memory implementation Ram.i
properties
Memory_Size => 1 GByte;
end Ram.i;

bus PCIe3 -- pcie x1
properties
Latency => 6us .. 11us; -- saw this info somewhere
SEI::BandWidthCapacity => 985.0 MBytesps;
-- mentioned in multiple references/ its this or 8 Gbps
Transmission_Time => [Fixed => 5us .. 10us; -- just an assumption,
--there is no record what so ever that mentions this info about pcie 3.0
PerByte => 1ns .. 1ns;]; -- calculated from the data rate (8GT/s)
```

```
end PCIe3;

bus Ethernet
properties
Bus_Properties::Available_Bandwidth => (10 MBytesps, 100 MBytesps, 1 GBytesps);
--  Data_Rate => 10000000000 bitsps; --transfer rate is 10 Gbps
Transmission_Time => [PerByte => 1 ns .. 2 ns;];
end Ethernet;

bus UART
properties
SEI::BandWidthCapacity => 9600.0 bitsps;
Transmission_Time => [PerByte => 833 us .. 833 us;];
-- based on calculations on the bandwidth
end UART;

-- rs585 is for ILD interface device
-- info can be found at
-- https://www.maximintegrated.com/en/design/technical-documents/app-notes/3/3884.html
bus rs485
properties
-- calculated from data rate for cable length of 1200 m to 600 m
--(would have data rate of 100 kbps .. 200 kbps)
Transmission_Time => [PerByte => 20 us .. 40 us;];
end rs485;

bus WirelessAntenna
end WirelessAntenna;

bus VanetAntenna
end VanetAntenna;

bus GSM_Antenna
end GSM_Antenna;

end iRSU::Platform;
```

A.3. RSU system integration

```
package iRSU::iRSU
public
with iRSU::Detection_Unit;
with iRSU::devices;
with iRSU::Input_Unit;
with iRSU::Output_Unit;
```

```
with iRSU::Platform;
with iRSU::Processing_Unit;

system iRSU_Sys
features
Wireless_Input : requires bus access iRSU::Platform::WirelessAntenna;
Vanet_Input : requires bus access iRSU::Platform::VanetAntenna;
eCall_Input : requires bus access iRSU::Platform::GSM_Antenna;
ILD_Input : requires bus access iRSU::Platform::rs485;
--ethernet : requires bus access iRSU::Platform::Ethernet;
end iRSU_Sys;

system implementation iRSU_Sys.UrbanMinimal
subcomponents
---------------------processes----------------------------------
Input_Unit : process iRSU::Input_Unit::input_process.i;
Detection_Unit : process iRSU::Detection_Unit::Detection_process.i;
Processing_Unit : process iRSU::Processing_Unit::mainprocess.i;
Output_Unit : process iRSU::Output_Unit::Output_Unit.i;
---------------------devices----------------------------------
ILD_Interface : device iRSU::devices::ILD_Interface;
Uart2Pcie : device iRSU::Devices::UART2PCIe;
Wifi_Modem : device iRSU::devices::WirelessModem;
Vanet_Modem : device iRSU::devices::VanetTranseiver;
--GSM_Modem : device iRSU::devices::GSM;
eCall_DataModem : device iRSU::devices::eCall_DataModem;
GPS : device iRSU::devices::GPS;
connections
----------------ports Connections-----------------
inILD1 : port ILD_Interface.ILD_Data  -> Uart2Pcie.ILD_Data_in;
inILD2 : port Uart2Pcie.ILD_Data_out  -> Input_Unit.ild_in ;
ildData : port Input_Unit.ild_data  -> Detection_Unit.inILD_Data ;
inVanet : port Vanet_Modem.VanetMsgs  -> Input_Unit.Vanet_input;
PVD : data access Input_Unit.pvdSharing -> Detection_Unit.inPVD_Data;
PossibleInc : port Detection_Unit.IncidentAlarm -> Processing_Unit.incdntAlarm;
inWless : port Wifi_Modem.incoming_Data  -> Input_Unit.Wireless_input;
Routing : port Input_Unit.Cloud_Data  -> Output_Unit.FromCloud;
WreckWatch : port Input_Unit.Wreckwatch_Data  -> Processing_Unit.WWatchevent;
Clearing : port Input_Unit.Clearing_Data  -> Processing_Unit.Clearingevent;
--eCallSignal : port GSM_Modem.eCallSignal  -> eCall_DataModem.inMSD;
eCallMSD : port eCall_DataModem.inMSD  -> Input_Unit.eCall_input;
eCallData : port Input_Unit.eCall_MSD  -> Processing_Unit.eCallevent;
IncNotif : port Processing_Unit.ToCloud  -> Output_Unit.IncidentNotif;
outVanet : port Output_Unit.ToCloud  -> Vanet_Modem.toCloud;
broadcast : port Output_Unit.broadcast  -> Vanet_Modem.broadcast;
EVRouting : port Output_Unit.EVRouting  -> Vanet_Modem.EVRouting;
```

```
Wifi_Acc : bus access Wireless_Input <-> Wifi_Modem.Data_Line;
Vanet_Acc : bus access Vanet_Input  <-> Vanet_Modem.Data_Line;
--GSM_Ac : bus access GSM_InOut  <-> GSM_Modem.Data_Line;
eCall_Acc : bus access eCall_Input <-> eCall_DataModem.Data_Line;
ILD_Acc : bus access ILD_Input  -> ILD_Interface.Data_Line0;

--pvdDA : data access Input_Unit.pvdsharing -> Detection_Unit.pvdRead;

flows
ILD_ETEflow: end to end flow ILD_Interface.f1 -> inILD1 -> Uart2Pcie.f1 ->
 inILD2 -> Input_Unit.ildFlow -> ildData ->
 Detection_Unit.ILDFlow -> PossibleInc ->
 Processing_Unit.f1 -> IncNotif -> Output_Unit.f1 ->
 outVanet -> Vanet_Modem.f2 {latency => 2 min .. 3 min;};

Van_ETEflow: end to end flow Vanet_Modem.f1 -> inVanet -> Input_Unit.VanetFlow -> pvd ->
 detection_unit.vflow -> PossibleInc -> Processing_Unit.f1 ->
 IncNotif -> Output_Unit.f1 -> outVanet -> Vanet_Modem.f2
{latency => 2 min .. 3 min;};
eCall_ETEflow: end to end flow  ecall_datamodem.f1 -> ecallmsd -> Input_Unit.ecallflow ->
 ecalldata -> Processing_Unit.f2 -> incnotif -> Output_Unit.f1 ->
  outVanet -> Vanet_Modem.f2 {latency => 1 sec .. 2 sec;};

WreckW_ETEflow: end to end flow Wifi_Modem.f1 -> inwless -> Input_Unit.wreckwflow -> wreckwatch ->
Processing_Unit.f3 -> incnotif -> Output_Unit.f1 -> outVanet ->
Vanet_Modem.f2  {latency => 1 sec .. 2 sec;};

properties

Period => 100 ms applies to  Input_Unit.WirelessFilter, Input_Unit.eCallPoll,
    Input_Unit.VanetFilter, Output_Unit.Routing_Th;
Period => 30 ms  applies to  Output_Unit.ToCloud_Th, Processing_Unit.Th_Verification
, Processing_Unit.Th_eCallHandler, Processing_Unit.Th_WreckWatchServer;
Period => 20 sec applies to  Detection_Unit.Van_th, Detection_Unit.Incident_th,
Detection_Unit.ILD_th;
Period => 1 min  applies to  Input_Unit.ild_Listener;

Deadline => 100 ms applies to Input_Unit.WirelessFilter, Input_Unit.eCallPoll,
  Input_Unit.VanetFilter, Output_Unit.Routing_Th;
Deadline => 30 ms  applies to Output_Unit.ToCloud_Th, Processing_Unit.Th_Verification,
Processing_Unit.Th_eCallHandler, Processing_Unit.Th_WreckWatchServer;
Deadline => 20 sec applies to Detection_Unit.Van_th, Detection_Unit.Incident_th, Detection_Unit.ILD_th;
Deadline => 1 min  applies to Input_Unit.ild_Listener;


end iRSU_Sys.UrbanMinimal;
---=========================================================================
```

```
system implementation iRSU_Sys.Deployment_x86_option2_1 extends iRSU_Sys.UrbanMinimal
--First option for x86, in order CPU
subcomponents
Cpu : processor iRSU::Platform::x86proc.i;
Ram : memory iRSU::Platform::Ram.i;
pcie3 : bus iRSU::Platform::PCIe3;
uart : bus iRSU::Platform::UART;
connections
B0 : bus access pcie3  <-> Cpu.BA;
B1  : bus access pcie3  <-> Ram.BA;
B2 : bus access uart   <-> ILD_Interface.Data_Line1;
B3 : bus access uart   <-> uart2pcie.Data_Line0;
B4 : bus access pcie3  <-> uart2pcie.Data_Line1;
properties
-----Binding the functional model to the execution platform----------
Actual_Memory_Binding  => (reference (Ram)) applies to
Cpu, Input_Unit, Detection_Unit, Output_Unit, Processing_Unit;
Actual_Connection_Binding  => (reference (uart)) applies to inILD1;
Actual_Connection_Binding  => (reference (pcie3)) applies to inVanet, eCallMSD, inWless, inILD2,
outVanet, broadcast, EVRouting;
Actual_processor_Binding  => (reference (Cpu)) applies to Input_Unit.ild_Listener,
Input_Unit.WirelessFilter, Input_Unit.eCallPoll, Input_Unit.VanetFilter,
Detection_Unit.ILD_th, Detection_Unit.Van_th,
Detection_Unit.Incident_th, Processing_Unit.Th_eCallHandler,
Processing_Unit.Th_WreckWatchServer, Processing_Unit.Th_Verification,
Output_Unit.Routing_Th, Output_Unit.ToCloud_Th;
-------------Threads properties as per CPU architecture-------------------------
-- using in-order CPU gives us the following execution results,
--these results were simulated on gem5
Compute_Execution_Time => 3650us .. 3655us applies to  Input_Unit.VanetFilter;
Compute_Execution_Time => 3945us .. 3950us applies to  Input_Unit.ild_Listener;
Compute_Execution_Time => 3625us .. 3630us applies to   Input_Unit.WirelessFilter;
Compute_Execution_Time => 3755us .. 3760us applies to Input_Unit.eCallPoll,
Output_Unit.ToCloud_Th;
Compute_Execution_Time => 4725us .. 4735us applies to Detection_Unit.ILD_th;
Compute_Execution_Time => 4540us .. 4550us applies to Detection_Unit.Van_th;
Compute_Execution_Time => 3820us .. 3830us applies to Detection_Unit.Incident_th;
Compute_Execution_Time => 3775us .. 3780us applies to Processing_Unit.Th_WreckWatchServer;
Compute_Execution_Time => 3785us .. 3790us applies to Processing_Unit.Th_eCallHandler;
Compute_Execution_Time => 4700us .. 4710us applies to Processing_Unit.Th_Verification;
Compute_Execution_Time => 3765us .. 3770us applies to  Output_Unit.Routing_Th;
end iRSU_Sys.Deployment_x86_option2_1;

 ---=======================================================================
system implementation iRSU_Sys.Deployment_x86_option2_2 extends iRSU_Sys.UrbanMinimal
-- Second option for x86, out of order CPU
subcomponents
```

```
Cpu : processor iRSU::Platform::x86proc.i;
Ram : memory iRSU::Platform::Ram.i;
pcie3 : bus iRSU::Platform::PCIe3;
uart : bus iRSU::Platform::UART;
connections
B0 : bus access pcie3  <-> Cpu.BA;
B1  : bus access pcie3  <-> Ram.BA;
B2 : bus access uart   <-> ILD_Interface.Data_Line1;
B3 : bus access uart   <-> uart2pcie.Data_Line0;
B4 : bus access pcie3  <-> uart2pcie.Data_Line1;
properties

Actual_Memory_Binding  => (reference (Ram)) applies to Cpu, Input_Unit, Detection_Unit,
Output_Unit, Processing_Unit;
Actual_Connection_Binding  => (reference (uart)) applies to inILD1;
Actual_Connection_Binding  => (reference (pcie3)) applies to inVanet, eCallMSD, inWless, inILD2,
outVanet, broadcast, EVRouting;
Actual_processor_Binding  => (reference (Cpu)) applies to Input_Unit.ild_Listener,
Input_Unit.WirelessFilter, Input_Unit.eCallPoll, Input_Unit.VanetFilter,
Detection_Unit.ILD_th, Detection_Unit.Van_th, Detection_Unit.Incident_th,
Processing_Unit.Th_eCallHandler, Processing_Unit.Th_WreckWatchServer,
Processing_Unit.Th_Verification, Output_Unit.Routing_Th, Output_Unit.ToCloud_Th;

-- using out of order CPU gives us the following execution results,
--these results were simulated on gem5
Compute_Execution_Time => 1450us .. 1455us applies to  Input_Unit.VanetFilter;
Compute_Execution_Time => 1600us .. 1605us applies to  Input_Unit.ild_Listener;
Compute_Execution_Time => 1460us .. 1465us applies to   Input_Unit.WirelessFilter;
Compute_Execution_Time => 1500us .. 1505us applies to Input_Unit.eCallPoll;
Compute_Execution_Time => 1935us .. 1940us applies to Detection_Unit.ILD_th;
Compute_Execution_Time => 1800us .. 1810us applies to Detection_Unit.Van_th;
Compute_Execution_Time => 1545us .. 1550us applies to Detection_Unit.Incident_th;
Compute_Execution_Time => 1530us .. 1535us applies to Processing_Unit.Th_WreckWatchServer;
Compute_Execution_Time => 1528us .. 1533us applies to Processing_Unit.Th_eCallHandler;
Compute_Execution_Time => 1850us .. 1855us applies to Processing_Unit.Th_Verification;
Compute_Execution_Time => 1528us .. 1533us applies to  Output_Unit.Routing_Th;
Compute_Execution_Time => 1515us .. 1520us applies to Output_Unit.ToCloud_Th;
end iRSU_Sys.Deployment_x86_option2_2;

---======================================================================
system implementation iRSU_Sys.Deployment_arm_option1 extends iRSU_Sys.UrbanMinimal
-- ARM CPU, with L2 cache
subcomponents
---------------------Platform-----------------------------------
Cpu : processor iRSU::Platform::x86proc.i;
Ram : memory iRSU::Platform::Ram.i;
pcie3 : bus iRSU::Platform::PCIe3;
```

```
uart : bus iRSU::Platform::UART;
l2cache : memory iRSU::Platform::L2_Cache;
connections
B0 : bus access pcie3  <-> Cpu.BA;
B1  : bus access pcie3  <-> Ram.BA;
B2 : bus access uart   <-> ILD_Interface.Data_Line1;
B3 : bus access uart   <-> uart2pcie.Data_Line0;
B4 : bus access pcie3  <-> uart2pcie.Data_Line1;
l2busaccess : bus access pcie3  <-> l2cache.BA1;
properties

Actual_Memory_Binding  => (reference (Ram)) applies to Cpu, Input_Unit, Detection_Unit,
 Output_Unit, Processing_Unit;
Actual_Connection_Binding  => (reference (uart)) applies to inILD1;
Actual_Connection_Binding  => (reference (pcie3)) applies to inVanet, eCallMSD, inWless, inILD2,
outVanet, broadcast, EVRouting;
Actual_processor_Binding  => (reference (Cpu)) applies to Input_Unit.ild_Listener,
Input_Unit.WirelessFilter, Input_Unit.eCallPoll, Input_Unit.VanetFilter,
Detection_Unit.ILD_th, Detection_Unit.Van_th, Detection_Unit.Incident_th,
Processing_Unit.Th_eCallHandler, Processing_Unit.Th_WreckWatchServer,
Processing_Unit.Th_Verification, Output_Unit.Routing_Th, Output_Unit.ToCloud_Th;

-- using out of order CPU gives us the following execution results,
--these results were simulated on gem5
Compute_Execution_Time => 85us .. 90us   applies to   Input_Unit.eCallPoll, Output_Unit.Routing_Th;
Compute_Execution_Time => 110us .. 115us applies to Input_Unit.ild_Listener;
Compute_Execution_Time => 68us .. 72us  applies to Input_Unit.WirelessFilter, Input_Unit.VanetFilter;
Compute_Execution_Time => 264us .. 270us applies to Detection_Unit.ILD_th;
Compute_Execution_Time => 220us .. 225us applies to Detection_Unit.Van_th;
Compute_Execution_Time => 105us .. 110us applies to Detection_Unit.Incident_th;
Compute_Execution_Time => 95us .. 100us  applies to  Output_Unit.ToCloud_Th,
 Processing_Unit.Th_eCallHandler, Processing_Unit.Th_WreckWatchServer;
Compute_Execution_Time => 250us .. 255us applies to Processing_Unit.Th_Verification;
--  in aadl inspector it insist on making the deadline matches the period

end iRSU_Sys.Deployment_arm_option1;
------=====================================================================
system implementation iRSU_Sys.Deployment_arm_option2_1 extends iRSU_Sys.UrbanMinimal
-- ARM CPU,
subcomponents
Cpu : processor iRSU::Platform::arm_proc2.i;
Ram : memory iRSU::Platform::Ram.i;
pcie3 : bus iRSU::Platform::PCIe3;
uart : bus iRSU::Platform::UART;
connections
B0 : bus access pcie3  <-> Cpu.BA;
B1  : bus access pcie3  <-> Ram.BA;
```

```
B2 : bus access uart   <-> ILD_Interface.Data_Line1;
B3 : bus access uart   <-> uart2pcie.Data_Line0;
B4 : bus access pcie3  <-> uart2pcie.Data_Line1;
properties

Actual_Memory_Binding  => (reference (Ram)) applies to Cpu, Input_Unit, Detection_Unit,
 Output_Unit, Processing_Unit;
Actual_Connection_Binding  => (reference (uart)) applies to inILD1;
Actual_Connection_Binding  => (reference (pcie3)) applies to inVanet, eCallMSD, inWless, inILD2,
outVanet, broadcast, EVRouting;
Actual_processor_Binding  => (reference (Cpu)) applies to Input_Unit.ild_Listener,
Input_Unit.WirelessFilter, Input_Unit.eCallPoll, Input_Unit.VanetFilter,
Detection_Unit.ILD_th, Detection_Unit.Van_th, Detection_Unit.Incident_th,
Processing_Unit.Th_eCallHandler, Processing_Unit.Th_WreckWatchServer,
Processing_Unit.Th_Verification, Output_Unit.Routing_Th, Output_Unit.ToCloud_Th;

-- using out of order CPU gives us the following execution results,
--these results were simulated on gem5
Compute_Execution_Time => 152us .. 157us applies to   Input_Unit.eCallPoll;
Compute_Execution_Time => 115us .. 120us applies to  Input_Unit.WirelessFilter;
Compute_Execution_Time => 120us .. 125us applies to  Input_Unit.VanetFilter;
Compute_Execution_Time => 230us .. 235us applies to  Input_Unit.ild_Listener;
Compute_Execution_Time => 780us .. 785us applies to  Detection_Unit.ILD_th;
Compute_Execution_Time => 525us .. 530us applies to  Detection_Unit.Van_th;
Compute_Execution_Time => 200us .. 205us applies to  Detection_Unit.Incident_th;
Compute_Execution_Time => 165us .. 170us applies to  Output_Unit.ToCloud_Th,
Processing_Unit.Th_WreckWatchServer;
Compute_Execution_Time => 170us .. 175us applies to  Processing_Unit.Th_eCallHandler;
Compute_Execution_Time => 160us .. 165us applies to  Output_Unit.Routing_Th;
Compute_Execution_Time => 565us .. 570us applies to  Processing_Unit.Th_Verification;

end iRSU_Sys.Deployment_arm_option2_1;

---===============================================================================================
system implementation iRSU_Sys.Deployment_arm_option2_2 extends iRSU_Sys.UrbanMinimal
-- ARM CPU,
subcomponents
Cpu : processor iRSU::Platform::arm_proc2.i;
Ram : memory iRSU::Platform::Ram.i;
pcie3 : bus iRSU::Platform::PCIe3;
uart : bus iRSU::Platform::UART;
connections
B0 : bus access pcie3  <-> Cpu.BA;
B1  : bus access pcie3  <-> Ram.BA;
B2 : bus access uart   <-> ILD_Interface.Data_Line1;
B3 : bus access uart   <-> uart2pcie.Data_Line0;
B4 : bus access pcie3  <-> uart2pcie.Data_Line1;
```

```
properties

Actual_Memory_Binding  => (reference (Ram)) applies to Cpu, Input_Unit, Detection_Unit,
Output_Unit, Processing_Unit;
Actual_Connection_Binding  => (reference (uart)) applies to inILD1;
Actual_Connection_Binding  => (reference (pcie3)) applies to inVanet, eCallMSD, inWless, inILD2,
outVanet, broadcast, EVRouting;
Actual_processor_Binding  => (reference (Cpu)) applies to Input_Unit.ild_Listener,
Input_Unit.VanetFilter, Input_Unit.WirelessFilter, Input_Unit.eCallPoll,
Detection_Unit.ILD_th, Detection_Unit.Van_th, Detection_Unit.Incident_th,
Processing_Unit.Th_eCallHandler, Processing_Unit.Th_WreckWatchServer,
Processing_Unit.Th_Verification, Output_Unit.Routing_Th, Output_Unit.ToCloud_Th;

-- using out of order CPU gives us the following execution results,
--these results were simulated on gem5
Compute_Execution_Time => 75us .. 80us applies to    Input_Unit.eCallPoll;
Compute_Execution_Time => 114us .. 120us applies to Input_Unit.ild_Listener;
Compute_Execution_Time => 60us .. 65us applies to Input_Unit.WirelessFilter,
Input_Unit.VanetFilter;
Compute_Execution_Time => 260us .. 265us applies to Processing_Unit.Th_Verification;
Compute_Execution_Time => 355us .. 360us applies to Detection_Unit.ILD_th;
Compute_Execution_Time => 220us .. 225us applies to Detection_Unit.Van_th;
Compute_Execution_Time => 100us .. 105us applies to Detection_Unit.Incident_th;
Compute_Execution_Time => 85us .. 90us applies to  Output_Unit.ToCloud_Th,
Processing_Unit.Th_eCallHandler, Processing_Unit.Th_WreckWatchServer;
Compute_Execution_Time => 80us .. 85us applies to  Output_Unit.Routing_Th;

end iRSU_Sys.Deployment_arm_option2_2;
end iRSU::iRSU;
```