УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Аутоматско генерисање програмске подршке за сервисно-оријентисану комуникацију између инфо-забавних и система за напредну подршку возачу у аутомобилима

- ДОКТОРСКА ДИСЕРТАЦИЈА -

Ментор:
ванр. проф. др Марија Антић

Кандидат:
Душан Кењић

Нови Сад, 2023.

УНИВЕРЗИТЕТ У НОВОМ САДУ **ОБРАЗАЦ – 5а**
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА [1]

Врста рада:	Докторска дисертација
Име и презиме аутора:	Душан Кењић
Ментор (титула, име, презиме, звање, институција)	Др Марија Антић, ванредни професор, Факултет техничких наука, Универзитет у Новом Саду
Наслов рада:	Аутоматско генерисање програмске подршке за сервисно-оријентисану комуникацију између инфо-забавних и система за напредну подршку возачу у аутомобилима
Језик публикације (писмо):	Српски (ћирилица)
Физички опис рада:	Унети број: Страница 177 Поглавља 8 Референци 204 Табела 37 Слика 42 Графикона 0 Прилога 0
Научна област:	Електротехничко и рачунарско инжењерство
Ужа научна област (научна дисциплина)	Рачунарска техника и рачунарске комуникације
Кључне речи / предметна одредница:	Комуникација унутар возила, инфо-забавни системи, напредни системи за помоћ возачу, сервисно-оријентисана архитектура, генеративно програмирање, развој вођен моделом

[1] Аутор докторске дисертације потписао је и приложио следеће Обрасце:
5б – Изјава о ауторству;
5в – Изјава о истоветности штампане и електронске верзије и о личним подацима;
5г – Изјава о коришћењу.
Ове Изјаве се чувају на факултету у штампаном и електронском облику и не кориче се са тезом.

Резиме на језику рада:	Комуникациони подсистем аутомобила захтева унапређење решењем које омогућава ефикасну размену података, како са компонената програмске подршке тако и са елемената физичке архитектуре. У тренутно заступљеној доменској архитектури, овај проблем се своди на комуникацију између два програмски најнапреднија, домена за пружање напредне помоћи возачу и инфо-забавног домена. Спречавање редундансе у имплементацији настале мешањем функционалних захтева међу доменима би унапредило интеграциони процес, који има кључну улогу у трајању развојног циклуса аутомобских система и омогућило бржи развој нових идеја и функционалности. У овом раду предложено решење примењује принципе сервисно-оријентисане архитектуре употребом *SOME/IP* протокола и решава неслагања у усвајању механизама и парадигме овог протокола у хетерогеном аутомобилском систему. Додатно, решење пружа аутоматизацију генерисањем читаве програмске подршке која врши размену података. Усвајање принципа развоја вођеног моделом у имплементацији самог генератора оставља на кориснику искључиво задатак описа жељених спрега у језику за дефинисање спрега типичном за домен за напредну подршку возачу. Након тога, модел спрега у овом језику ће бити преведен у моделе других језика за дефинисање спрега типичних за инфо-забавни домен. Решење је развијано по стандардима аутомобилске индустрије и верификовано мерењима перформанси, тестирањем сваког сегмента решења и формалном верификацијом превођења. Рад је потврђен на више различитих физичких уређаја.
Датум прихватања теме од стране надлежног већа:	
Датум одбране: (Попуњава одговарајућа служба)	
Чланови комисије: (титула, име, презиме, звање, институција)	Председник: Др Никола Теслић, редовни професор, Факултет техничких наука, Нови Сад Члан: Др Мирослав Поповић, редовни професор, Факултет техничких наука, Нови Сад Члан: Др Иван Каштелан, ванредни професор, Факултет техничких наука, Нови Сад Члан: Др Марио Врањеш, редовни професор, ФЕРИТ Осијек, Хрватска Ментор: Др Марија Антић, ванредни професор, Факултет техничких наука, Нови Сад
Напомена:	

UNIVERSITY OF NOVI SAD
FACULTY OF TECHNICAL SCIENCES

KEY WORD DOCUMENTATION [2]

Document type:	Doctoral dissertation
Author:	Dušan Kenjić
Supervisor (title, first name, last name, position, institution)	PhD Marija Antić, associate professor, Faculty of Technical Sciences, University of Novi Sad
Наслов рада:	Generating Software for Service-Oriented Communication between Infotainment and Advanced Driver Assistance Systems in Vehicles
Language of text (script):	Serbian (cyrillic)
Physical description:	Number of: Pages 177 Chapters 8 References 204 Tables 37 Illustrations 42 Graphs 0 Appendices 0
Scientific field:	Electrical and Computer Engineering
Scientific subfield (scientific discipline):	Computer Engineering, Engineering of Computer Based Systems
Subject, Key words:	In-vehicle communication, infotainment systems, advanced driver assistance systems, service-oriented architecture, generative programming, model-driven development

[2]The author of doctoral dissertation has signed the following Statements:
5б – Statement on the authority,
5в – Statement that the printed and e-version of doctoral dissertation are identical and about personal data,
5г – Statement on copyright licenses.
The paper and e-versions of Statements are held at he faculty and are not included into the printed thesis.

Abstract in English language:	The communication subsystem in vehicles requires improvement through a solution that enables efficient data exchange between both, software and hardware components. In the current domain-based architecture, this issue involves communication between two most advanced domains: advanced driver assistance system and in-vehicle infotainment. Preventing redundancy in the implementation caused by mixing functional requirements among domains improves the integration process, which plays a crucial role in duration of the automotive systems development cycle and enables faster adoption of new ideas and functionalities. This research proposes a solution that applies the principles of service-oriented architecture, utilizing the SOME/IP protocol and resolving discrepancies in its adoption within the heterogeneous automotive system. Additionally, the solution automates the generation of entire software needed for data exchange. Adopting model-driven development principles leaves the end users only with the task of defining interfaces in the language typical for advanced driver assistance system. Subsequently, the model of interfaces in this language will be translated into models of other languages typical for the infotainment domain. The solution is developed following automotive industry standards, verified through performance measurements, testing each solution segment, and formally verifying the translation. The study is confirmed on multiple different physical devices.
Accepted on Scientific Board on:	
Defended: (Filled by the faculty service)	
Thesis Defend Board: (title, first name, last name, position, institution)	President: PhD Nikola Teslić, full professor, Faculty of Technical Sciences, Novi Sad Member: PhD Miroslav Popović, full professor, Faculty of Technical Sciences, Novi Sad Member: PhD Ivan Kaštelan, associate professor, Faculty of Technical Sciences, Novi Sad Member: PhD Mario Vranješ, full professor, FERIT, Osijek, Croatia Mentor: PhD Marija Antić, associate professor, Faculty of Technical Sciences, Novi Sad
Note:	

Захвалност

Вилијем Артур Ворд је једном рекао да је осећати захвалност и не изразити је исто као и умотати поклон, а не предати га. Из тог разлога, најмање што могу да урадим је да дам поклон захвалности свима који су допринели да спроведем ово истраживање.

Најпре бих желео да се захвалим Институту РТ-РК који је обезбедио све потребне ресурсе да се овај пројекат заправо и изврши. Упутио бих велику захвалност свом ментору, професорки Марији Антић за сате и сате разговора и саветовања, као и осталим професорима са катедре за знање које сам имао прилику да усвојим у протеклим годинама.

Хвала колеги др Николи Јовалекићу који ми је помогао да се спремим за овај вишегодишњи пут поделивши са мном своје знање о раду аутомобилских система и савете о писању доктората из ове области и колегиници Соњи Мандић за мотивационе и охрабрујуће говоре.

Волео бих да се захвалим и свима који су учествовали у изради овог пројекта. Хвала колегама Томиславу Маруни, Душану Живкову, Крсти Лазићу, Тихомиру Анђелићу, Давору Рапићу, др Бранимиру Ковачевићу и др Милени Милошевић који су делили своје искуство и драгоцене савете при изради. Изузетну захвалност упућујем својим верним саборцима и колегама из *LAVA* тима Александру, Анђели, Вуку, Душану, Луки, Маријани, Милици, Немањи, Огњену, Раденку, Радомиру и Стефану, хвала вам за сав уложени труд и сваки дан који смо провели заједно.

Најважније, захваљујем се својим најближима, а посебно својој девојци Јовани што је имала оволико разумевања, стрпљења и толеранције када ми је било најпотребније и својој породици која је жртвовала све да бих ја данас био то што јесам и нису сумњали да ћу то постати, вама посвећујем овај рад.

Садржај

Листа слика

Листа табела

Скраћенице и појмови

ADAS	**A**dvanced **D**river **A**ssistance **S**ystem - *Пружање напредне помоћи возачу*
AES-GCM	**A**dvanced **E**ncryption **S**tandard **G**alois/**C**ounter **M**ode - *Стандард за шифровање података*
AGL	**A**utomotive **G**rade **L**inux - *Оперативни систем заснован на Linux систему проширен спрегама за возила*
AIDL	**A**ndroid **I**nterface **D**efinition **L**anguage - *Језик за дефинисање спрега у Android оперативном систему*
ART	**A**ndroid **Run**time - *Окружење за извршавање апликација у Android оперативном систему*
ARXML	**A**UTOSA**R** **XML** - *Језик за дефинисање спрега за AUTOSAR платформу*
ASPICE	**A**utomotive **S**oftware **P**erformance **I**mprovement and **C**apability d**E**termination - *Стандард за процену развоја пројеката у аутомобилској индустрији*
AVB	**A**udio **V**ideo **B**ridging - *Скуп стандарда за пренос аудио и видео података путем Ethernet протокола*
BE	**B**est **E**ffort - *Класа неприоритизованих учесника у мрежном саобраћају*
BSW	**B**asic **Soft**ware - *Основни слој програмске подршке*
CAN	**C**ontroller **A**rea **N**etwork - *Магистрала у возилима*
CBS	**C**redit **B**ased **S**haper - *Протокол за контролисану резервацију мрежног саобраћаја уз умањење пристрасности*
CDT	**C**ontroll-**D**ata-**T**raffic - *Осетљива класа управљачких података на мрежи*
CPU	**C**entral **P**rocessing **U**nit - *Централна процесорска јединица*
DDS	**D**ata **D**istribution **S**ervice - *Скуп стандарда за дистрибуирање података*
DoS	**D**enial-**o**f-**S**ervice - *Напад са циљем блокирања услуга*
DMA	**D**irect **M**emory **A**ccess - *Непосредан приступ меморији независно од процесора*
DSL	**D**omain **S**pecific **L**anguage - *Језик специфичан за област*
DSP	**D**igital **S**ignal **P**rocessor - *Процесор за обраду сигнала*
ECU	**E**lectronic **C**ontrol **U**nit - *Електронске контролне јединице*
EMF	**E**clipse **M**odelling **F**ramework - *Радни оквир за моделовање базиран на Eclipse решењу*
FIDL	**F**ranca **I**nterface **D**efinition **L**anguage - *Језик за дефинисање спрега у Franca радном оквиру*
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray - *Програмабилни чипови*
FPS	**F**rames per **s**econd - *Број оквира у секунди*
GMSL	**G**igabit **M**ultimedia **S**erial **L**ink - *Гигабитна мултимедијална серијска веза*
GPS	**G**lobal **P**ositioning **S**ystem - *Систем за глобално позиционирање*
GPU	**G**raphics **P**rocessing **U**nit - *Графичка процесорска јединица*
HMI	**H**uman **M**achine **I**nterface - *Спрега човека и машине (која се односи на читаву дигиталну таблу која у возилу)*
HSM	**H**ardware **S**ecurity **M**odule - *Физички уређај који пружа сигурносне механизме*
HTML	**H**yper **T**ext **M**arkup **L**anguage - *Језик намењен опису веб страница*
I2C	**I**nter-**I**ntegrated **C**ircuit - *Синхрона серијска комуникациона магистрала*
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers

IPC	**I**nter **P**rocess **C**ommunication - *Међупроцесна комуникација*
IV	**I**nitialization **V**ector - *Иницијализациони вектор*
IVI	**I**n-**V**ehicle **I**nfotainment - *Забава и информисање у возилу, тј. инфо-забавни систем*
JNI	**J**ava **N**ative **I**nterface - *Спрега између Java и C/C++ програмског кода*
LAN	**L**ocal **A**rea **N**etwork - *Скуп уређаја повезаних на заједничкој физичкој локацији*
LIN	**L**ocal **I**nterconnect **N**etwork - *Серијски комуникациони протокол*
MAC	**M**essage **A**uthentication **C**ode - *Ознака за аутентикацију поруке*
MCAL	**M**icrocontroller **A**bstraction **L**ayer - *Ниво апстракције микроконтролера*
MDD	**M**odel **D**riven **D**evelopment - *Развој вођен моделом*
MPSoC	**M**ulti**p**rocessor **S**ystem-**o**n-**C**hip - *Вишепроцесорски систем на чипу*
MOST	**M**edia **O**riented **S**ystems **T**ransport - *Протокол за размену мултимедијалног садржаја*
MQTT	**M**essage **Q**ueuing **T**elemetry **T**ransport - *Протокол за размену података у интернету ствари*
NDK	**N**ative **D**evelopment **K**it - *Скуп алата за развој C/C++ програма у Android систему*
NoC	**N**etwork-**o**n-**C**hip - *Мрежа на чипу*
POSIX	**P**ortable **O**perating **S**ystem **I**nterface - *Скуп стандарда системске програмске спреге базиране на Unix оперативним системима*
PTP	**P**recision **T**ime **P**rotocol - *Протокол за синхронизацију часовника*
RC	**R**ate **C**onstrained - *Осетљива класа саобраћаја за кориснике ограниченог протока*
RPC	**R**emote **P**rocedure **C**all - *Позивање удаљене методе*
RTE	**Run**time **E**nvironment - *Компонента за контролу комуникације у AUTOSAR систему*
RTP	**R**eal-**T**ime **P**rotocol - *Протокол за пренос података у реалном времену*
RTPS	**R**eal **T**ime **P**ublisher **S**ubscriber - *Протокол за остваривање претплатничког режима преноса у реалном времену*
RTSP	**R**eal-**T**ime **S**treaming **P**rotocol - *Протокол за пренос тока података у реалном времену*
SDP	**S**oftware-**D**evelopment **P**lan - *Документ који описује план развоја програмске подршке*
SOA	**S**ervice-**O**riented **A**rchitecture - *Сервисно-оријентисана архитектура*
SoC	**S**ystem-**o**n-**C**hip - *Систем на чипу*
SOME/IP	**S**calable service-**O**riented **M**iddlewar**E** over **IP** - *Протокол за сервисно-оријентисана архитектура у возилима*
SPI	**S**erial **P**eripheral **I**nterface - *Протокол за серијска спрегу са периферијама*
SQL	**S**tructured **Q**uery **L**anguage - *Програмски језик за складиштење и обраду података у базама*
SRP	**S**tream **R**eservation **P**rotocol - *Протокол за резервацију преноса*
TAS	**T**ime **A**ware **S**haping - *Протокол за временско распоређивање употребе медијума*
TCP	**T**ransmission **C**ontrol **P**rotocol - *Протокол за контролисану размену података*
TSN	**T**ime **S**ensitive **N**etwork - *Скуп стандарда за имплементацију Ethernet протокола осетљивог на време*
TT	**T**ime-**T**riggered - *Класа саобраћаја осетљива на време*

TTL	**T**ime-**T**o-**L**ive - *Механизам за ограничавање живота података на мрежи*
UART	**U**niversal **A**syncrhonous **R**eceiver-**T**ransmitter - *Универзални серијски асинхрони протокол за пренос података*
UDP	**U**ser **D**atagram **P**rotocol - *Транспортни протокол за непоуздан пренос података међу рачунарима*
USB	**U**niversal **S**erial **B**us - *Универзални серијски шински протокол*
V2X	**V**ehicle-**t**o-**E**verything - *Комуникација између возила и околине*
VLAN	**V**irtual **L**ocal **A**rea **N**etwork - *Виртуелна локална мрежа*
V2I	**V**ehicle-**t**o-**I**nfrastructure - *Комуникација између возила и инфраструктуре*
V2V	**V**ehicle-**t**o-**V**ehicle - *Комуникација између возила*
V2P	**V**ehicle-**t**o-**P**edestrian - *Комуникација између возила и пешака*
V2D	**V**ehicle-**t**o-**D**evice - *Комуникација између возила и уређаја*
V2G	**V**ehicle-**t**o-**G**rid - *Комуникација између возила и електричне мреже*
V2H	**V**ehicle-**t**o-**H**ome - *Комуникација између возила и куће*
V2N	**V**ehicle-**t**o-**N**etwork - *Комуникација између возила и мреже*
VM	**V**irtual **M**achine - *Виртуелна машина*
VLAN	**V**irtual **L**ocal **A**rea **N**etwork - *Виртуелна локална мрежа*
WLAN	**W**ireless **L**ocal **A**rea **N**etwork - *Бежична локална мрежа (Wi-Fi мрежа)*
XCP	**XC**alibration **P**rotocol - *Протокол за калибрацију и мерење*
XML	**Ex**tensible **M**arkup **L**anguage - *Језик за означавање текста у облику ознака*

Сажетак

Последњих година, аутомобилски системи су предмет многобројних истраживања великих светских лабораторија, које се надмећу у покушајима да обезбеде максималне перформансе кроз поуздану, али и комфорну и забавну вожњу. Екосистем истраживача и инжењера који се баве овим темама је значајно проширен, као и функционалности и могућности које аутомобилски систем данашњице може да понуди. За испуњење таквих потреба, захтеви стављени пред уграђене системе који чине возило се усложњавају и доводе до преплитања разноликих технологија и компонената физичке архитектуре и програмске подршке.

Стога је потребно унапредити комуникациони подсистем како би могао да испрати еволуцију функционалности имајући у виду кључне изазове: хетерогеност архитектуре аубомобилских система, разноликост случаја коришћења, дефинисање комуникационих механизама и коришћених протокола, сигурност и безбедност, технике аутоматизације, верификационе и валидационе механизме.

У овој докторској дисертацији предложено је, по нашем сазнању прво и за сад једино решење за аутоматско генерисање сервисно-оријентисане међудоменске комуникације унутар аутомобилског система преко *SOME/IP* протокола. Дат је приказ комуникационих магистрала, механизама и протокола у возилу и њихова потенцијална употреба у оваквом решењу као и међусобно поређење и путања дефинисања *SOME/IP* протокола за ову конкретну улогу. Предложено решење користи принципе генеративног програмирања и развоја вођеног моделима чиме је комуникација аутоматизована. Аутоматизација подразумева, на првом месту, имплементирано превођење између типичних језика за дефинисање комуникационих спрега као што су *ARXML*, *FIDL* и *AIDL*. Наведени језици припадају стандардима усвојеним у различитим доменима и нису усаглашени по основи, па је за њихово превођење било потребно додатно прилагођавање. Такође, поред превођења ових језика међу доменима, имплементирано је и генерисање делова који међусобно размењују податке и додатно послужују крајње апликације размењеним подацима. С обзиром на то да језици за дефинисање спрега имају различите парадигме, односно могућност примењивања различитих комуникационих механизама, генерисане компоненте које врше размену података је било потребно имплементирати тако да буду прилагођене *SOME/IP* протоколу и његовом принципу рада.

Овиме се, поред поспешивања рада комуникационог подсистема у циљу олакшане интеграције, скраћивања развојног циклуса и остваривања функционалности модерних возила, омогућава истраживачима и инжењерима бржа и једноставнија имплементација која у фокус ставља развој сопствене идеје без потребе за стицањем доменских знања из различитих делова аутомобилског система и имплементацију пропратних зависности тиме што ће на располагању имати податке из другог домена који долазе од аутоматски генерисаних компонената.

Abstract

In the contemporary era, automotive systems have become a subject of extensive research in renowned global laboratories, seeking to achieve maximum performance while providing reliable, comfortable, and enjoyable driving experiences. The community of researchers and engineers involved in automotive-related topics has significantly expanded, alongside the continuous advancement of functionalities and capabilities offered by modern vehicular systems. As a result, the requirements for embedded systems within vehicles are constantly growing, leading to the convergence of diverse technologies and components of both, physical architecture and software.

Therefore, enhancing the communication subsystem is crucial to accommodate the evolution of functionalities while addressing key challenges such as the heterogeneity of automotive system architecture, diverse use cases, definition of communication mechanisms and protocols, security, safety, automation techniques, and verification and validation mechanisms.

In this doctoral dissertation, we propose the first and to the best of our knowledge only solution for the automatic generation of service-oriented inter-domain communication within automotive systems, utilizing the SOME/IP protocol. The study includes an overview of communication buses, mechanisms, and protocols used within vehicles, exploring their potential application in this solution. Additionally, a comparative analysis and a roadmap for defining the SOME/IP protocol for this specific role are presented. Our proposed solution employs generative programming and model-driven development principles, enabling automated communication. The automation process involves the translation between typical languages used for defining communication interfaces like ARXML, FIDL, and AIDL. As these languages belong to different standards adopted in various domains and are inherently incompatible, additional adaptation was necessary for successful translation. Furthermore, the solution includes the generation of components facilitating data exchange among themselves and effectively serving end applications. Adapting these components to accommodate the SOME/IP protocol and its principles was a critical part of the implementation.

This approach not only accelerates the operation of the communication subsystem, simplifying integration, shortening development cycles, and realizing modern vehicle functionalities but also empowers researchers and engineers with a faster and more straightforward implementation process focused on their unique ideas. It eliminates the need to acquire domain-specific knowledge from various automotive system parts and integrate accompanying dependencies since data from other domains can be efficiently obtained through automatically generated components.

1 Увод

1.1 Мотивација и допринос докторске дисертације

Развој аутомобилских система је сложен процес, у коме се комбинују различите научне и инжењерске дисциплине. Традиционално, аутомобилска индустрија примењује достигнућа из области физике, саобраћајног и транспортног инжењеринга, као и машинског и инжењерства производње делова и опреме разноврсних материјала и намене. Временом се у процес све више укључују и инжењери електронике и електротехнике, и различите физичке архитектуре рачунара налазе своју примену у возилима. Развој модерних возила захтева и учешће софтверских инжењера, задужених за стварање програмске подршке сачињене од преко 100 милиона линија програмског кода и алгоритама који управљају разноврсним подацима. Данас у аутомобилској индустрији сведочимо и све распрострањенијој примени вештачке интелигенције, различитим жичним и бежичним комуникационим технологијама, као и технологијама усвојеним из потрошачке електронике. Такође, све већи значај даје се развоју сигурносних механизама и одбрани од напада, као и развоју по стандардима и процесима неопходним да би квалитет крајњег производа био обезбеђен.

Један овако сложен систем аутомобила модерног доба за кориснике више не представља искључиво средство намењено транспорту путника и робе са једног места на друго, већ пружа комфорно, поуздано и оптимизовано кретање, уз подршку система за безбедну аутономну вожњу и помоћ возачу, као и извор информација и забаве за све који се налазе у возилу. У складу са разнородним корисничким захтевима, догађа се и еволуција у дизајну аутомобилског система, померајем примарног фокуса са механичких на електричне и електронске делове, а тиме коначно и на програмску подршку која ће овим деловима давати одговарајућу функцију [1].

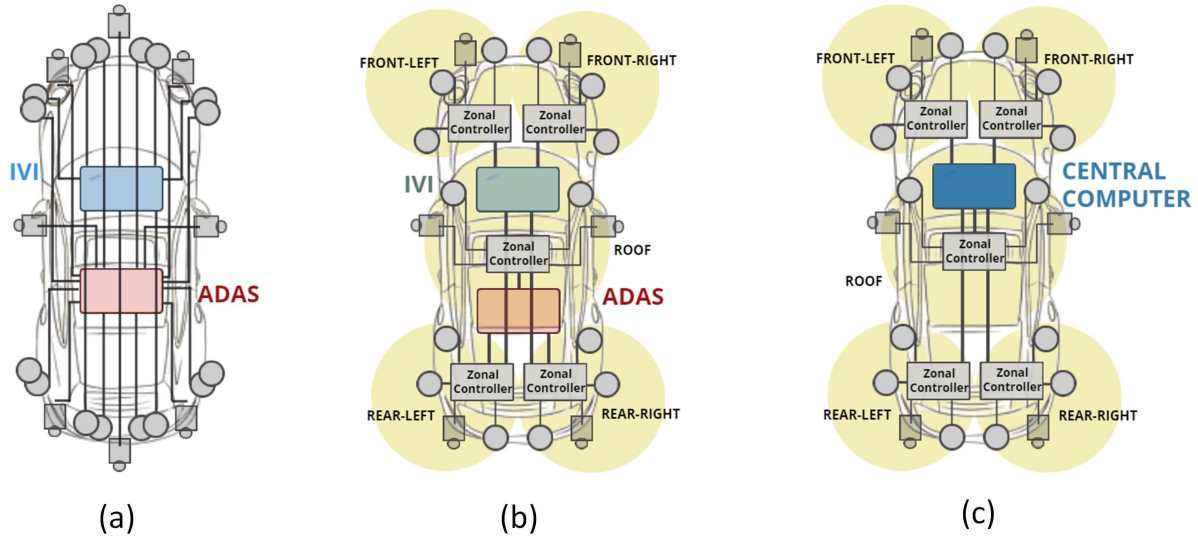
Последично, сложеност аутомобилских система расте и повећава се изазов интеграције различитих компонената у усаглашену целину. Пораст броја компонената физичке архитектуре возила двојако утиче на комуникацију унутар возила: отежавајући физичко ожи-

чавање и одређивање погодних комуникационих магистрала, као и отежавајући размену података међу компонентама програмске подршке и одређивање комуникационих механизама у ту сврху. Из овог разлога се и архитектура аутомобилског система мења у складу са захтевима [2]. Испочетка, прелазак са механичких на електронске контролне јединице (енгл. *Elecrtonic Control Unit - ECU*) извршен је како би се омогућила аутоматизација надзорних операција, као и развој и употреба сензора и уграђене програмске подршке у сврху контролисаног понашања потцелине [3, 4]. Такође, тиме се остварила и могућност за увођење напредних функционалности које су контролисане од стране електронских контролних јединица, попут прилагодљивог темпомата, функције праћења траке, детекције и избегавања препрека и судара, и слично. Умножавање оваквих модула захтевало је неопходну артикулацију, тј. организацију у инфраструктури самог система возила, јер је циклус развоја, интеграције и одржавања више од стотину електронских контролних модула постао једнако велики проблем као и сама имплементација [5, 6].

Први корак у решавању овог проблема представља организација архитектуре аутомобилског система у различите домене [7], на основу функције коју сваки од домена извршава:

- Погонски домен (енгл. *Powertrain*) - састоји се од компонената које стварају и преносе енергију потребну за кретање возила
- Тело аутомобила (енгл. *Body*) - компоненте које чине структуру и највећи део аутомобила попут прозора, врата, крова, разних носача, поклопаца и прекривача, итд.
- Шасија (енгл. *Chassis*) - костур возила који практично "носи"читав аутомобил и на који непосредно належу компоненте из домена тела
- Домен за пружање напредне помоћи возачу (енгл. *Advanced Driver Assistance System - ADAS*) - представља компоненте (програмске подршке и физичке архитектуре) које омогућавају поуздану подршку у вожњи кроз функционалности као што су темпомат, детекције знака и других учесника, праћење линија пута, предвиђање и избегавање удеса, итд.
- Домен за забаву и информисање, односно инфо-забавни домен (енгл. *In-Vehicle Infotainment - IVI*, тј. *Human Machine Interface - HMI*) - представља компоненте (програмске подршке и физичке архитектуре) које путницима омогућавају приступ забавном садржају и информацијама о самом возилу као и информацијама из спољашње околине.
- Домен за телеметрију и повезаност (енгл. *Telemetry/Connectivity*) - омогућава размену података између возила и околине (укључујући друга возила, паметну инфраструктуру и удаљене станице)

Сваки од домена је контролисан доменским контролером [8] и имплементиран је као енкапсулирана целина у којој коегзистирају различите јединице физичке архитектуре, сензори, актуатори и комуникационе магистрале и протоколи са заједничким циљем да



Слика 1: Еволуција архитектуре аутомобилског система

изврше одговарајућу функцију. Иако су домени замишљени као потпуно дисјунктне целине, неретко долази до потребе за извршавањем истих или сличних функција у различитим доменима. Стога настаје редундантност у имплементацији истих или сличних алгоритама и употреби истих или слични сензора и јединица за обраду података. Дуплирање ресурса указује на то да и у оваквој организацији има доста простора за оптимизацију. У овом раду је истраживачки окулар усмерен ка, по питању програмске подршке, најнапреднијим доменима *ADAS* и *IVI*, као и ка њиховом међусобном односу. Ова два домена су приказана у контексту аутомобилског система на слици 1.а.

Један од начина који би, на првом месту, поспешио ожичење и искоришћење ресурса јесте прелазак на зоналну архитектуру [9, 10] која је приказана на слици 1.б. Њена главна одлика је организација ресурса на основу главних зона у возилу којих најчешће има четири, пет или шест:

- Предња десна (енгл. *Front-right*) - ако бисмо аутомобил поделили на квадранте координатног система, ова зона прикупља и организује податке из првог квадранта
- Предња лева (енгл. *Front-left*) - ако бисмо аутомобил поделили на квадранте координатног система, ова зона прикупља и организује податке из другог квадранта
- Стражња лева (енгл. *Rear-left*) - ако бисмо аутомобил поделили на квадранте координатног система, ова зона прикупља и организује податке из трећег квадранта
- Стражња десна (енгл. *Rear-right*) - ако бисмо аутомобил поделили на квадранте координатног система, ова зона прикупља и организује податке из четвртог квадранта
- Опционо једна или две средишње (енгл. *Roof* или *Mid-zone*, односно у случају две: *Mid-left* и *Mid-right*)

Свака од зона поседује зонални контролер задужен за извршавање основне обраде и манипулацију над подацима и ресурсима који припадају одговарајућој зони. Ови подаци

би даље били прослеђивани доменским контролерима на коришћење. На овај начин, доменски контролери би и даље постојали, али би њихова улога била релаксирана, а подаци и ресурси боље искоришћени. Пример је разлика у употреби сензора камере. У доменској архитектури, сваки од домена би захтевао појединачни сензор, непосредну повезаност и сву обраду података са сензора би вршио сам доменски контролер (са изузетком могућности употребе паметних сензора који у себи садрже мање или више напредну обраду података). У зоналној архитектури, био би довољан један сензор над којим би манипулација и иницијална обрада података била извршена у оквиру зоналних контролера, а потом по потреби прослеђена одговарајућем доменском контролеру. Ова архиктектура се сматра хибридном јер може да коегзистира уз доменску као и уз централизовану архитектуру [11] у зависности од тога да ли се подаци са зоналних контролера прослеђују доменском или централном рачунару на даље коришћење.

Идеја о оптималном искоришћењу ресурса била би остварена у централизованој архитектуру приказаној на слици 1.ц, где би постојао један супер рачунар који би добијао улазе са свих сензорских елемената, вршио све задатке и у складу са задацима покретао одговајуће актуаторске јединице. Такође, централизовани рачунар би задржао поделу језгара и других јединица обраде према функционалности тако да добију форму интерне доменске расподеле. Оваква централозована архитектура је футуристички циљ који тренутно стање у области не може да усвоји из многоструких разлога [12]: неопходна реорганизација читаве инфраструктуре аутомобилског система (1), безбедносни и сигурносни изазови које додатно отежава проблематика једне тачке отказа система и интензивнија интерференција задатака различите функционалности које обавља јединствена платформа (2), непостојање платформе која би могла да задовољи све функционалне захтеве које треба да поседује један супер рачунар (3), итд.

Ни различити приступи у еволуцији модела аутомобилског система не решавају интеграционе проблеме у потпуности, остављајући простор за оптимизацију. Зонални контролери, као ни централни рачунар сам по себи не могу омогућити размену података између компонената програмске подршке и апликација које се извршавају на раличитим платформама доменских контролера, односно различитим јединицама обраде централног рачунара. Иако омогућавају мањи број јединица физичке архитектуре (попут сензора из претходно наведеног примера) и смањују потребно ожичавање, неизоставни фактор у процесу интеграције је комуникација и усаглашеност између компонената програмске подршке. Стога, основни проблем и мотивацију за израду ове тезе представља проналажење решења за размену података између компонената које има пет особина наведених у наставку: може бити применљиво у свим споменутим архитектурама (1); омогућава развој нових функционалности базираних на размени података између компонената које већ постоје у различитим доменима (2); омогућава оптимизацију ресурса неопходних за извршавање већ постојећих функционалности (3); пружа платформу која олакшава развој и интеграцију потпуно нових функционалности (4); омогућава размену како алгоритамских и података са компонената програмске подршке, тако и дељење ресурса физичке архитектуре односно података који долазе са њих (5). У циљу постизања решења, неопходно је узети у обзир и смер тока самих података.

Решење представљено овим радом предлаже сервисно-оријентисану комуникацију организовану тако да јединице обраде домена за помоћ возачу, било да се налазе на засебном доменском контролеру или у оквиру јединственог централизованог рачунара, снадбдевају подацима инфо-забавне јединице обраде, користећи *Scalable service-Oriented MiddlewarE over IP - SOME/IP* као главни комуникациони протокол. Контекст размењиваних информација је двојак: информације могу бити подаци из компонената програмске подршке, алгоритама, односно апликација, као и подаци са сензора или других ресурса физичке архитектуре. С обзиром на проблеме сложености интеграције и развојног циклуса програмске подршке у аутомобилу, још један важан аспект који доноси предложено решење је аутоматизација, односно чињеница да је у оквиру предложеног решења реализован алат за аутоматско генерисање свих делова програмске подршке потребне за комуникацију, а на основу описних језика који се користе за дефинисање спрега својствених за специфичан домен (*AUTOSAR XML - ARXML, Franca Interface Definition Language - FIDL* и *Android Interface Definition Language -AIDL*). Ови језици у основи нису међусобно усаглашени, а неки од њих нису ни прилагођени за примену са *SOME/IP* протоколом. Стога је у оквиру истраживања реализовано превођење између поменутих језика, као и прилагођавање комуникационих механизама са оним што је дефинисано *SOME/IP* протоколом. Компоненте добијене генерисањем на страни инфо-забавног домена за било које типично окружење или оперативни систем (нпр. *Linux* базирана решења) дају јасно дефинисану програмску спрегу, путем које је могуће преузети податке из система за напредну подршку возача и даље их дистрибуирати или употребити. За *Android* је, као истраживањем препознат оперативни систем инфо-забавног демена за који се произвођачи већински опредељују, генерисан сервис којим је омогућено и прослеђивање података до самих корисничких апликација путем међупроцесне комуникације јединствене за *Android* уз детаљну анализу и опис различитости у раду за одговарајуће случаје употребе.

Кључни изазови којима се ова теза бави у решавању представљеног проблема су:

- **Хетерогеност архитектуре аутомобилских система** — Приликом конструисања решења потребно је узети у обзир различите оперативне системе, оквире за програмску подршку и стандарде развијене за различите подсистеме, који су настали еволуцијом архитектуре аутомобилског система.

- **Разноликост случајева употребе** — Подршка функционалностима аутомобила нове ере, у којима се укрштају конвенционалне потребе за аутомобилом као средством које пружа максимално поуздану и комфорну вожњу са контекстом потрошачких захтева.

- **Дефинисање комуникационих механизама и коришћених протокола** — Мноштво комуникационих магистрала, протокола и механизама заступљених у аутомобилском систему треба да буду испитани, са циљем утврђивања и дефинисања приступа коришћеног за реализацију оваквог решења.

- **Сигурност и безбедност** — Интерференција омогућена комуникацијом потенцијално доприноси нарушавању безбедносних и сигурносних аспеката система, пого-

тово имајући у виду да безбедност, у до сада у потпуности енкапсулираном аутомобилском систему, није имала велики значај и да су комуникациони механизми адаптирани за аутомобилске системе на начин који у многоме изоставља безбедносна питања.

- **Технике аутоматског генерисања програмске подршке** — Укључивање генеративног програмирања и развоја вођеног моделима као начина да се комуникациони слојеви програмске подршке адекватно опишу. Подразумева и дефинисање правила којима ће из створених описа бити генерисана програмска подршка. Додатно, подразумева и решавање неусаглашености међу моделима различитих домена из једног овако хетерогеног система као што је аутомобилски.

- **Верификациони и валидациони механизми** — Одређивање скупа тестова и адекватног начина тестирања који је отежан укључивањем генеративног програмирања, развоја вођеног моделима и манипулацијама, односно превођењем између дефинисаних модела.

Овиме дисертација, поред непосредног поспешивања комуникационог подсистема и доприноса архитектури возила, даје допринос и инжењерској заједници, пружајући алат који омогућава инжењерима да пажњу усмере на сопствену идеју и програмско решење, без потребе за развијањем различитих доменских знања и поновним имплементирањем помоћних алгоритама од којих то решење може да зависи. Поред оптимизоване интеграције, овиме би био отворен пут за нове функционалности и случаје употребе у возилима нове ере. Неки од примера су:

- **Визуелизација околине возила** — Из поузданих алгоритама из домена који пружа подршку возачу, могуће би било издвојити податке о координатама других учесника у саобраћају, њихове путање кретања и брзину, детектоване саобраћајне знаке, сигнализацију, траке и линије на путу и слично, те их кроз механизме сервисно-оријентисане архитектуре прилагођене за аутомобилску индустрију проследити ка домену за информисање и забаву. Инфо-забавни домен ће бити у могућности да потпуно графички или кроз проширену стварност (енгл. *Augmented Reality*) представи визуелно окружење возила путницима, ради пружања напреднијег корисничког искуства и комфора, чиме се путници могу уверити у квалитет аутономије и начина рада возила. Овиме би потреба за двоструком имплементацијом алгоритама за перцепцију околине возила у оквиру два различита домена последично била елиминисана.

- **Обучавање возача** — На основу поузданих алгоритама из домена који пружа подршку возачу, могуће би било издвојити тачне информације о оптималним путањама и начину на који би возило требало да се креће како би кретање било што ефикасније у постизању што мањег трзаја, оптималне брзине, потрошње и слично. Ове информације би у оквиру домена за инфо-забаву могле бити коришћене за обуку возача (примарно употребљиво у супер аутомобилима и тркачким аутомобилима), упоре-

ћујући их са остварeним кретњама и предлажући промене на визуелан и звучним сигналом приказан начин.

- **Употреба ресурса физичке архитектуре** — Ресурси физичке архитектуре попут сензора из домена за пружање помоћи возачу би такође могли бити искоришћени и у оквиру инфо-забавног домена. Кабинска камера за надгледање возача може се искористити и за видео позив, а предња камера за перцепцију околине може се искористити и за фотографисање пејзажа испред возила или као улазни податак за апликације које на основу пејзажа и локације препознају врхове планина и слично. Са друге стране, модели тренираних неуронских мрежа који се користе у алгоритмима инфо-забавног домена могу бити извршавани на знатно снажнијим елементима физичке архитектуре система за подршку возачу.

- **Друге информације о околини или самом возилу** — С обзиром на то да у практичном смислу сви подаци из алгоритама домена за подршку возачу као и унутрашњих или спољашњих сензора могу бити једноставно изложени путницима, тако би деца поред већ имплементираних могућности да играју игрице или гледају филмове на инфо-забавним системима могли да уче саобраћајне знакове, правила и ситуације детектоване у току вожње.

Предложено решење је скалабилно и двосмерно, па подржава и проширење другим комуникационим протоколима, односно омогућава и супротан смер тока података. Међутим, због природе инфо-забавног домена и његових платформских недостатака у виду поузданости и извршавања у реланом времену, компоненте и алгоритми из домена подршке возача не би смели да се ослањају на овакве податке и из тог разлога овај смер тока података неће бити обрађен у овој дисертацији.

Да би решење било имплементирано, истраживање је подељено у четири велике целине: анализа и одабир погодне комуникационе магистрале, протокола и механизма (1), имплементација која подразумева првобитно дефинисање архитектуре модела решења и разлагање на компоненте, покривајући разноликости коришћених стандарда из оба домена (2), процес аутоматског генерисања који подразумева стварање модела за превођење између језика за опис спреге који припадају различитим доменима и који нису међусобно усаглашени, а потом генерисање компонената програмске подршке које омогућавају комуникацију (3) и испитивање решења која подразумева мерења, тестирања и формалну верификацију (4). Декомпозиција сваке од целина и начин њихове реализације ће бити представљени у појединачним поглављима решења. Методе истраживања које су коришћене су квалитативна и квантитативна. Квалитативна је примењена у почетном делу истраживања који подразумева студију случаја и преглед тренутно коришћених комуникационих канала, протокола и механизама за комуникацију, анализу језика за моделовање комуникације у различитим доменима као и анализу принципа генеративног програмирања. Са друге стране, квантитативна метода је примењена кроз вршење свих мерења перформанси имплементираних компонената програмске подршке, како би погодни приступи били искуствено утврђени и проверени најпре у одабиру модула средњег слоја, потом у архитектуралним приступима решењу, а напослетку и у процени утицаја решења на свео-

бухватне перформансе у виду кашњења, оптерећења и робусности за различите случајеве употребе.

1.2 Организација дисертације

Дисертација је организована у седам поглавља.

Уводно поглавље представља мотивацију, образлаже истраживачку потребу за решавањем представљене проблематике и дефинише циљ истраживања. Даје кратак опис хипотезе, преглед изазова са којима се сусреће решење приказано овом дисертацијом, фазе, начин, као и методологију развоја решења.

Стање у области је наредно поглавље, и у њему су представљене све потребне теоријске основе као и релевантне информације из научних и индустријских достигнућа из области истраживања. Први део овог поглавља представља преглед досадашњег развоја и актуелних истраживања везаних за домен за пружање подршке у вожњи, као и за инфо-забавни домен. Као последица представљених захтева модерне аутомобилске индустрије, дедукована је потреба за хетерогеношћу архитектуре аутомобилског система, потреба за употребом различитих јединица обраде, сензора и актуатора, као и потреба за к,онсолидацијом и стандардизацијом решења за програмску подршку, оквире и посредничке програме што даље доводи до потребе за оптимизовањем интеграције и комуникације међу компонентама. Укратко су описане основне магистрале, комуникациони механизми и протоколи са фокусом на сервисно-оријентисану архитектуру и њену примену кроз *SOME/IP* протокол. Такође, на крају поглавља су представљени и принципи генеративног програмирања и развоја вођеног моделима, са акцентом на њихову примену у развоју програмске подршке аутомобилског система кроз најкоришћеније језике за дефинисање комуникационих спрега (*ARXML*, *FIDL* и *AIDL*).

Одабир погодног комуникационог протокола је треће поглавље у ком је представљен сегмент истраживања којим је *SOME/IP* протокол одређен као погодан за примену у предложеном решењу и дато кратко поређење са другим могућим приступима. Додатно, дефинисани су уочени случајеви употребе у којима није пожљено ослонити се на *SOME/IP* протокол и предожене су идеје за заменска решења у тим случајевима.

Реализација комуникације применом сервисно- оријентисане парадигме је четврто по реду поглавље и у њему је дато решење за проблем размене података између једница обраде из домена за пружање напредне подршке возачу и инфо-забавног домена кроз примену сервисно-оријентисане парадигме путем *SOME/IP* протокола. Приказана су два потенцијална начина распоређивања компонената, дистрибуирани и централизовани, и дато је њихово поређење. Затим, представљена је интеграција *SOME/IP* парадигме у неке концептуално различите механизме као што је међупроцесна комуникација путем повезивача у *Android* оперативном систему (енгл. *Android Binder*). Дискутована су решења за безбедносна и сигурносна питања чији аспекти стандардом *SOME/IP* протокола нису покривени. Такође, имплементирана је и представљена посебна програмска спрега канала за размену велиих података као случај коришћења за који није погодна употреба *SOME/IP* протокола. Овај канал покрива могућност комуникације преко *UDP, TCP, RTP* и *RTSP* протокола.

Пето поглавље је **аутоматско генерисање средњег слоја комуникације** и у њему је дат читав модел и описан је процес генерисања делова програмске подршке потребних за остваривање комуникације описане у претходном поглављу. У склопу овог решења је имплементиран мета-модел језика за дефинисање спрега у оквиру *Android* оперативног система (енгл. *AIDL*) и представљен је његов однос са неким већ постојећим мета-моделима језика за дефинисање спрега из других система (*ARXML* и *FIDL*). Дато је решење за њихово међусобно превођење као и генерисање у претходном поглављу имплементираних компонената потребних за постизање комуникације између ова два домена.

Верификација и резултати је шесто поглавље по реду и у њему је приказана валидација и верификација модела процеса генерисања. Такође, представљен је вишеслојни вид тестирања у склопу превођења и генерисања програмског кода коришћењем развоја вођеног моделом како би процес генерисања био што веродостојније испитан. Дати су и сви тестни случаји у виду тестова најмањих јединица за сваку од компонената решења, интеграционих тестова за њихову симбиозу и функционалних тестова за валидацију почетних захтева. За само превођење међу моделима је извршена и формална верификација.

Закључак је седмо поглавље и у њему је дат преглед постигнутих доприноса решења које нуди ова дисертација. Дата је кратка дискусија и осврт на ограничења и недостатке и наведен је правац даљег истраживања.

На послетку је дат приказ **извора и литературе** коришћене у истраживању.

2 Стање у области

У овом поглављу дате су теоријске основе дисертације, као и преглед достигнућа релевантних научних и индустријских истраживања у области. У првој половини поглавља је дат преглед задатака и архитектура домена за пружање подршке у вожњи и инфо-забаног домена, како би се јасније разумели разлози истраживачке тежње за омогућавањем њихове интеграције и компатибилности, и објаснила сложеност овог изазова. Затим, дат је преглед магистрала, протокола и комуникационих механизама заступљених унутар једног возила, са циљем да се разјасни њихова примена и идентификују тренутни правци развоја. Напослетку, дат је преглед техника генеративног програмирања и развоја вођеног моделима, који истиче све већи значај аутоматизације у савременим решењима.

2.1 Еволуција домена за пружање подршке у вожњи

Интензивна улагања у покушаје имплементације механизама који омогућавају поуздану аутономну вожњу или помоћ возачу нису случајна путања развоја аутомобилске индустрије. Разлог представља чињеница да је око 95% саобраћајних несрећа узроковано људским факторима, тј. погрешним одлукама, непажњом и замором возача [13]. Иако је развој возила нове генерације вођен идејом да њихове функционалности пруже већу безбедност путницима, у пракси се она сусрећу са многим правним и моралним факторима који отежавају њихову примену. Из тог разлога су развијени стандарди који описују различите нивое помоћи возачу, тј. нивое аутономије возила, као и законске регулативе које дефинишу правила у вожњи, правила производње аутомобила, као и права и обавезе потрошача и осигуравајућих кућа [14]. Са друге стране, ту су и етичке дилеме у примени возила са новим функционалностима, које морају узети у обзир утицај нових технологија на сваког појединца који учествује у саобраћају и захтевају анализу потенцијалних незгода, постављају изазов одабира најмање штетних опција и питања да ли је такав одабир уопште прихватљив [15], а скрећу пажњу и на чињеницу да се у овом тренутку чини да

ће незгода изазваних радом аутомобила бити заправо све више [16]. Додатно, један од најважнијих фактора је и утицај свих ових аспеката на инжењере и процес инжењерског развоја аутомобилских система. Резултат је потреба модерних возила да буду опремљена мноштвом механизама за помоћ возачу чији је квалитет потврђен. Неки од ових механизама су детекција коловозних трака и других учесника у саобраћају, предвиђање трајекторија, процена пажње возача, као и различити системи за упозорење возача на промене у окружењу. Већина наведених функционалности је већ у некој мери усвојена, док се поред њихове оптимизације ради постизања већег степена аутономије возила интензивно ради и на стварању механизама за аутономну вожњу.

2.1.1 Улога алгоритама и компонената програмске подршке у реализацији захтева

Алгоритми у системима за помоћ возачу (*ADAS*) су различите врсте и сложености. Функције *ADAS* система могу се поделити у неколико функционалних сегмената: перцепција околине, предикција и рачунање потенцијалних акција, оптимизација која води ка одабиру најефикасније акције, те сама акција. Перцепција околине представља први у низу од изазова који имају за циљ да омогуће возилу сналажење у реалном окружењу и реалним ситуацијама. О покушајима њеног остваривања сведоче разни изазови, демонстрације и светска такмичења попут *DARPA Grand/Urban Challenge, European Land-Robot Trial, Indy Autonomous Challenge* и слични. У литератури [17] је представљено рашчлањивање процеса перцепције околине на следеће компоненте:

- Детекција и праћење саобраћајних линија, трака и пута — За остваривање ових задатака често су довољни само основни принципи обраде слике и манипулација над различитим системима боја [18, 19], како би било могуће квалитетно екстраховати ивице. Некад се користе и принципи конволуционих неуронских мрежа како би се извршила сегментација возног дела саобраћајнице у односу на околину [20, 21]. Након тога се од овако идентификованих ивичних тачака различитим апроксимацијама формирају линије. Са стране математичке представе, формирање линија представља највећи проблем па је у том погледу тренутно активно неколико приступа. Један од њих је приказ параметризованих модела који су погодни за краће праволинијске деонице и који су често коришћени у Хофовим трансформацијама [22, 23]. Међутим, закривљене путање је проблематично представити искључиво параболичним, односно хиперболичним једначинама [24]. У случајевима када саобраћајне траке имају неправолинијске путање, многи покушавају да их опишу коефицијентима Безиерових кривих [25–27], односно различитим апроксимацијама помоћу полиноминалних функција [28]. Примена методе Бајесове мреже је такође активан приступ, који се ослања на континуираност, али не и на диференцијабилност криве [29]. Треба узети у обзир и да је за јасну представу читавог пута неоходно имати и рељефне податке, а за то су потребни подаци са нешто напреднијих оптичких система као што су лидари или стерео визија, што додатно чини овај задатак изазовним.

- Препознавање саобраћајних знакова — За остваривање ових задатака често се ко-

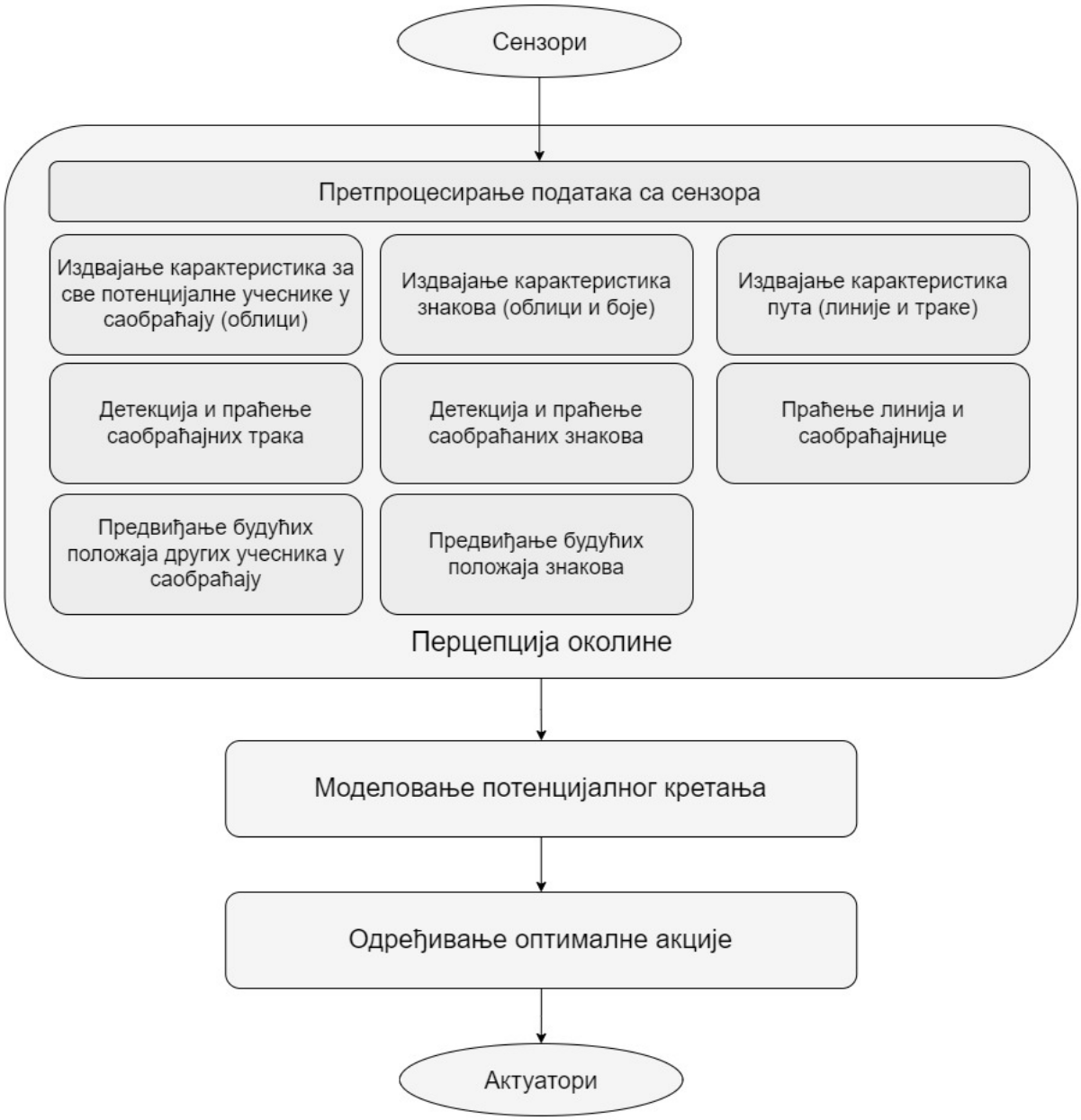
ристи комбинација конвенционалних приступа обраде слике како би се по боји и облику издвојио регион који потенцијално може представљати саобраћајни знак, а потом над тако издвојеним регионом извршила класификација и препознавање на основу обраде слике [30] или неуронских мрежа [31].

- Препознавање и праћење других учесника у саобраћају — Остваривање ових задатака и даље представља најсложенији изазов за решавање [32]. Разлог је разноликост облика, димензија и боја учесника у саобраћају и разноликост начина њиховог кретања. Иницијални покушаји решавања ових задатака који се и даље користе у неким решењима употребљавају хистограм градијената (енгл. *Histogram of Gradients - HoG*) [33]. Међутим, са применом конволуције у неуронским мрежама фокус је прешао на примену дубоких неуронских мрежа [34]. С обзиром на садржајност изазова, решења захтевају примену великих модела неуронских мрежа који захтевају извршавање милиона операција и задају највећи проблем за постизање адекватних перформанси у реалном времену [35] поготово при великим брзинама кретања возила.

Алгоритам за обраду сваке од претходно наведених компонената би се могао генерализовати и представити у неколико основних фаза као што је приказано на слици 2: претпроцесирање, издвајање значајних карактеристика, детекција и праћење детектованих објеката и затим предвиђање трајекторија и маневара детектованих објеката. Ове фазе се у бити могу тумачити и као фазе стварања перцепције околине јер се информације добијене из ових фаза појединачних компонената збрајају са циљем да се формира што веродостојнији модел реалне сцене у вожњи потребан за даљу анализу.

Претпроцесирање — Имајући у виду да је перцепција изузетно рачунарски захтевна, претпроцесирање има за циљ да припреми податке и омогући каснијим фазама ефикасније извршавање и тиме релаксира читав процес. Најчешће подразумева филтрирање улазних података [36] у циљу отклањања шума, уклањања мртвих пиксела, назначавања ивица и промене система боја [37, 38]. Неретко се примењују и неке манипулације попут издвајања региона од интереса како би се остале фазе алгоритама примењивале над смањеним скупом података и тиме скратило укупно време извршавања.

Издвајање значајних карактеристика — У овој фази се примењују различити механизми како би тражене одлике биле правилно екстраховане (детекција ивичних пиксела, апроксимација одговарајућих контура, облика, углова, образаца и шара). Било да је механизам за издвајање карактеристика и препознавање облика имплементиран коришћењем конвенционалних постулата обраде слике или новијих модела дубоких неуронских мрежа, неопходан је јер се комбиновањем препознатих карактеристика у наредном кораку формира закључак о томе који је објекат детектован и од квалитета издвајања препознатљивих образаца зависиће ефикасност детекције. Разни модели дубоког учења (надгледано, ненадгледано и појачано учење) покушавају да нађу примену и наилазе на бројне препреке. Истраживања [39] и [40] омогућавају детекцију објеката и семантичку сегментацију сцена из вожње на основу модела дубоких неуронских мрежа чији слојеви препознају и најмање облике. Нека истраживања, као на пример [41], предлажу и учење вожње непосредно, на



Слика 2: Основни алгоритми програмске подршке *ADAS* домена

основу конкретних одлука и маневара возача у реалним ситуацијама. Надгледано учење (енгл. *Supervised learning*) је ипак најзаступљеније, међутим, доступност скупа података за тренирање неуронских мрежа представља изузетан проблем. Због несавршености у расподели и обиму скупова података, модели неуронских мрежа буду боље научени на већински засупљене карактеристике на које утичу разни фактори попут временских услова, најучесталијих учесника у саобраћају, итд. Са друге стране, карактеристике типичне за опасне и несвакидашње ситуације, којих у скуповима за тренирање има мање, се лошије препознају. Стога је потребно детаљно применити наведене принципе претпроцесирања и анализе података које дају трениране мреже како би мреже биле унапређиване и тренинзи понављани све успешније [42]. Са друге стране, појачано учење (енгл. *Reinforcement learning*) није лако издводљиво јер је за аутомобилску област опасно тренирати неуронске мреже

на основу одлука у реалном времену. У случају оба претходно наведена модела учења коришћење симулатора за генерисање сценарија из вожње налази све већу примену [43–45]. Ненадгледано учење (енгл. *Non-supervised learning*) такође налази примену, али с обзиром на то да се базира на груписању података, више је заступљен у потрошачкој електроници или самом маркетингу него у постизању сегмената аутономне вожње.

Детекција и праћење односно анализа динамике детектованих објеката — Модели неуронских мрежа често енкапсулирају издвајање значајних карактеристика и детекцију. Међутим, то није случај у свим приступима. Неретко је неопходно, након што су идентификоване и локализоване карактеристике које чине објекат од значаја, проверити да ли је заиста то детектовани објекат. Провере се најчешће своде на операције које проверавају комбинације издвојених карактеристика док алгоритам не уочи ону која представља циљани објекат. Само идентификовање и детекција нису крај посматрања околине. Неопходно је на основу узастопних положаја истих објеката у времену реализовати математички модел за њихово праћење. Постоје различити приступи у реализацији ове фазе, а тренутно су актуелни комбиновани приступи који рачунају будуће положаје детектованих објеката узимајући у обзир у потпуности израчунате, као и положаје до којих се долази хеуристичким путем. Задавањем тежинских коефицијената за ова два приступа, долази се до најповољнијег односа. Један од оваквих примера за праћење објеката и предвиђања будућих положаја је поменути Калманов филтар [46].

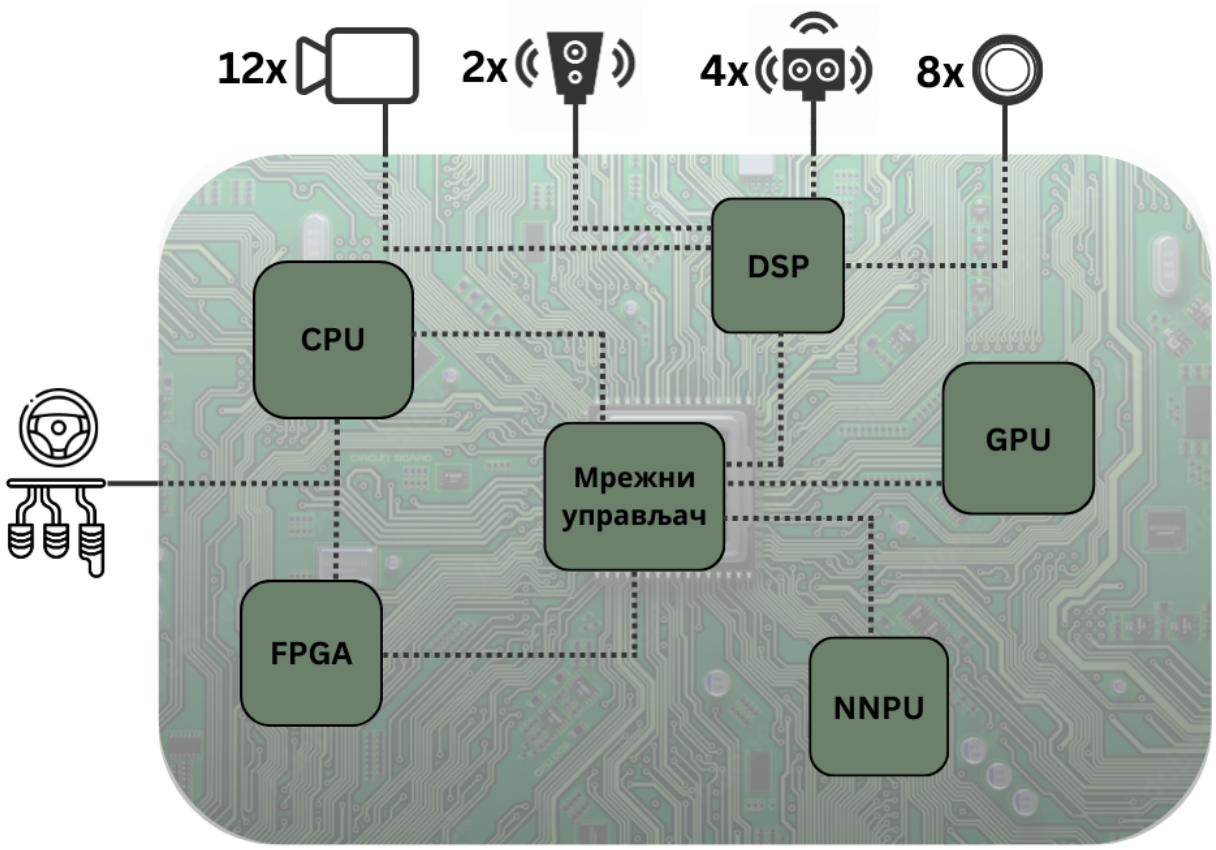
Предвиђање трајекторија и маневара других учесника — На основу предвиђених положаја детектованих објеката у времену, апроксимирају се и формирају математички модели који описују предвиђене трајекторије [47] и динамику кретања свих учесника и објеката у саобраћају. Ови подаци су кључни за даљу калкулацију маневара и путање кретања референтног возила.

Информације добијене из овако описаних алгоритама за перцепцију се збрајају са циљем да се што веродостојније направи модел сцене у вожњи потребан за даљу анализу. Поврх свега, повезаност возила са околином је најсвежија грана истраживања која омогућава размену података између возила, удаљених серверских база и паметне инфраструктуре увођењем нових технологија и идеје о поузданијем одређивању маневара размењивањем података са околином [48, 49], али чини и додатно оптерећење за читав систем.

Следећи кораци који представљају одлучивање су матматички проблеми моделовања потенцијалних путања кретања сопственог возила на основу детектоване возне подлоге и маневара других учесника. Од овако добијених модела проналази се најбоља путања узимајући у обзир мноштво фактора, брзину, трзај, потрошњу, прописана правила и наредбе добијена саобраћајним знацима и сигнализацијом и тако даље. По добијању жељеног кретања, задатак се преноси на актуаторе, убризгавање горива, регулисање преноса, позицију волана, кочнице и слично.

2.1.2 Улога ресурса физичке архитектуре у реализацији захтева

За успешно извршавање претходно описаних алгоритама очевидна је потреба за мноштвом ресурса физичке архитектуре: сензори, јединице обраде и актуатори као што је илустровано на слици 3.



Слика 3: Физичка архитектура *ADAS* домена

2.1.2.1 Сензори у *ADAS* домену Како би аутомобили нове ере имали што прецизније тумачење окружења потребно је прибављати податке са различитих сензорских система, баш као што и човек гради представу окружења на основу више чула. Сензори се могу поделити на унутрашње и спољашње. Унутрашњи служе за мерење температуре мотора, нивоа уља, батерије, горива и притиска у компонентама. Спољашњи служе да пруже информације о возилу у односу на околину. Ту спадају системи за позиционирање, радар, лидар, камере и ултразвучни сензори [32].

Систем за позиционирање се практично састоји од две навигационе компоненте: глобални систем за позиционирање (енгл. *Global Positioning System - GPS*) који пружа временске и локацијске информације о објектима и инерцијални навигациони систем (енгл. *Inertial Navigation System - INS*) који на основу акцелерометара и жироскопа пружа информације о положају објекта.

Ултразвучни сензори се користе за детектовање блиских објеката и површине у окружењу возила, најчешће при паркирању или споријој вожњи. Доступни су по питању цене и инсталације, међутим, ограничени су кратком раздаљином, најчешће до 5 метара [50]. У аутомобилима их обично има од 4 до 12 [51, 52]. Количина података по ултразвучном сензору представља нешто мање од 0.01Mb/s и подаци добијени са ових сензора се возачу најчешће визуализују кроз приказ раздаљине уз звучни сигнал који указује на близину објеката из непосредног окружења.

Радар је сензор који се користи за детектовање објеката на већој раздаљини. На основу овог сензора се може одредити раздаљина, али и брзина кретања других објеката. Број

радара у колима варира између 2 и 12 [53, 54]. Највећа предност радара је поузданост у извршавању при временским условима који доводе до смањене видљивости на путу као што су падавине и магла. Поред тога, постоје и друге одлике радара које га чине погодним сензором за употребу у аутомобилској области. Одређивање удаљености и брзине других објеката, детекција више објеката истовремено, већи домет у односу на ултразвучне сензоре и ценовна приступачност спадају у ове погодности. Са друге стране, мане у примени радара могу бити утицај других објеката као препрека за радарски сигнал који путује ваздухом, немогућност препознавања објеката, фреквентност пристизања и велика количина података [55]. Количина података по радарском сензору износи од 0.1 до 15 Mb/s.

Иако је употребом више радара могуће доћи до тродимензионалне представе објеката, у ову сврху је поузданије и квалитетније коришћење лидара. Лидар је сензор који такође може да служи за детектовање објеката из окружења возила, с тим да се од података који долазе са овог сензора формирају изузетно прецизне рељефне мапе окружења са тачношћу која износи од 2 до 10cm [56]. Број лидара у колима је променљив и варира од 2 до 6 [57,58]. Неке од негативних одлика лидара су неупотребљивост у лошим временским условима, могућ утицај сунчаних зрака и висока цена за масовну употребу. Количина података по лидарском сензору значајно варира у зависности од представе јединице податка, као и од броја јединица податка који лидар може да забележи у једној секунди, и креће се у распону од 20 до 400 Mb/s.

Камере су најзаступљенији сензори у аутомобилској индустрији, по узору на човека који готово све податке током вожње прикупља путем чула вида. Број камера варира, а најчешће их је око 12: 4 камере широких углова за формирање 360 степени приказа непосредне околине, као и предње и задње камере за праћење других објеката и већег дела пута - често у пару како би подржале стереовизију на основу које се може рачунати дубина и удаљеност, а напослетку и камера у кабини која контролише пажњу возача [32]. Камере имају бројне предности, уз развијене алгоритме за разне обраде и манипулације слике, имају широк спектар различитих одлика самих сензора у виду ширине поља, ноћне видљивости, броја оквира у секунди (енгл. *Frames per second - fps*) и разне моделе представе података (*RGB, YUV, HSI, H264, MJPEG*, итд.). Количина података по једном камера сензору, као и у случају лидара, значајно варира у зависности од представе јединице податка и броја јединица податка може бити забележен у јединици времена, и креће се у распону од 500 Mb/s до 3.5 Gb/s. Примена камера у алгоритмима за помоћ возачу је у тој мери заступљена да постоје и такозване паметне камере које већ врше иницијалну обраду слике у виду претпроцесирања или чак издвајања одређених карактеристика слике [59]. Међутим, као и за чуло вида човека, отежани временски и услови видљивости могу представљати проблем и у примени камера [60].

2.1.2.2 Јединице обраде у *ADAS* домену С обзиром на претходно описану сложеност алгоритама и компонената програмске подршке, било је неопходно и да јединице обраде, такође, испрате ту еволуцију захтева. Контролне јединице које обрађују податке постају све сложеније, па су тренутно систем на чипу (енгл. *System-on-Chip - SoC*), па и вишепроцесорски систем на чипу (енгл. *Multiprocessor System-on-Chip - MPSoC*) под-

разумеване технологије које доносе предности на пољу перформанси, олакшавају размену података и употребу заједничких ресурса, смањују цену и повећавају флексибилност система омогућавањем поделе оперативних система и генерално функционалности међу језгрима, односно чиповима [61, 62]. Аутори представљају начин употребе вишепроцесорских система на чипу за развој и покретање алгоритма за праћење објеката [62] и детекцију препрека [61], и показују да су овакве архитектуре итекако погодне за постизање захтева везаних за брзину извршавања алгоритама.

Међутим, функционалности нису раздељене само по језгрима, већ је због оперативно изузетно скупих задатака које извршавају компоненте програмске подршке било неопходно усвојити примену различитих јединица обраде које оптимизују алгоритме различите природе, односно функционалности што знатно усложњава читав систем. Из тог разлога, централне процесорске јединице, графичке процесорске јединице, програмабилни чипови, процесори за обраду сигнала, процесорске јединице за обраду неуронских мрежа су заступљени у аутомобилским системима и користе се у различите сврхе. Аутори у [63] пружају њихово поређење.

Коришћење процесора за обраду сигнала (енгл. *Digital Signal Processor – DSP*) је због оптимизације извршавања математичких операција и руковања меморијским токовима повољно за иницијалну обраду сензорских података у виду филтрирања и екстраховања тражених одлика са улаза [63]. *Texas Instruments* развија читаву серију система на чипу *TDA*, која за алгоритме и апликације рачунарске визије користи баш *DSP* јединице обраде и нуди могућност имплементације компонената потребних за достизање виших нивоа аутономне вожње [64, 65].

Графичке процесорске јединице (енгл. *Graphics Processing Unit – GPU*) су постале неизбежни део платформских ресурса у возилу и убрзано се развијају, експоненцијално повећавајући број језгара, а самим тим и број операција које могу извршити у датом времену. Додатне оптимизације распоређивања задатака као што је на пример представљено истраживачким радом у [66] указују на моћ ових платформи. У [67], аутори такође предлажу решење које представља виртуелну поделу језгра (енгл. *Kernel*) на мање целине, задајући им различите приоритете и омогућавајући заједничко планирање које доноси значајан бенефит у извршавању у односу на секвенцијално извршавање или основно распоређивање. Из ових разлога, све моћније графичке картице постају главни избор за извршавање најзахтевнијих алгоритама и примену дубоких неуронских мрежа. *Nvidia* као водећи представник индустрије у овој области улаже велике напоре да омогући извршавање овако захтевних задатака, стварајући радне оквире за оптимизацију алгоритама и употребу акцелератора физичке архитектуре [35]. Зато графичке јединице за обраду података постају нужна, али не и јефтина решења. Поред тога, са паралелизацијом извршавања, јавља се проблем детерминизма, који је за алгоритме у аутомобилској индустрији од изузетне важности. Следећи радови доносе различите приступе при коришћењу *GPU* јединица, како би се обезбедио детерминизам у извршавању алгоритама за аутономну вожњу [68–71]. У прва два од наведених радова је описан начин примене графичких контролних јединица у рачунарским програмима [68, 69]. Изнета је потреба за детерминизмом, односно аспекти које је могуће унапредити њиме и извршена подела детерминизма на врсте. Идентифико-

вана су ограничења и дискутовано је успорење које уводе нека од решења. Трећи рад [70] практично тумачи исте постулате, примењујући их конкретно на једну од најкоришћенијих платформи у развоју алгоритама у аутомобилској индустрији, док се у [71] предлаже хибридни режим динамичке и предефинисане резервације ресурса за извршавање раздвајањем задатака за извршавање у редове различитих приоритета и особина. Сви радови закључују да оптимално решење за проблем недетерминизма на графичким процесорским јединицама са мноштвом језгара не постоји и да је то отворена актуелна тема у којој постоји места за напредак.

Програмабилни чипови (енгл. *Field-Programmable Gate Array – FPGA*) су повољни по питању енергетске ефикасности и могућности извршавања одређеног задатка на основу интегрисаног кола физичке архитектуре које је могуће преконфигурисати. Такође, могуће их је путем комуникационих канала лако спојити на остале делове система што их чини кандидатом за различите функције у виду комбиновања или компресије података, као и сензорске фузије или доношења оптимизоване одлуке о кретању на основу различитих улаза [63, 72].

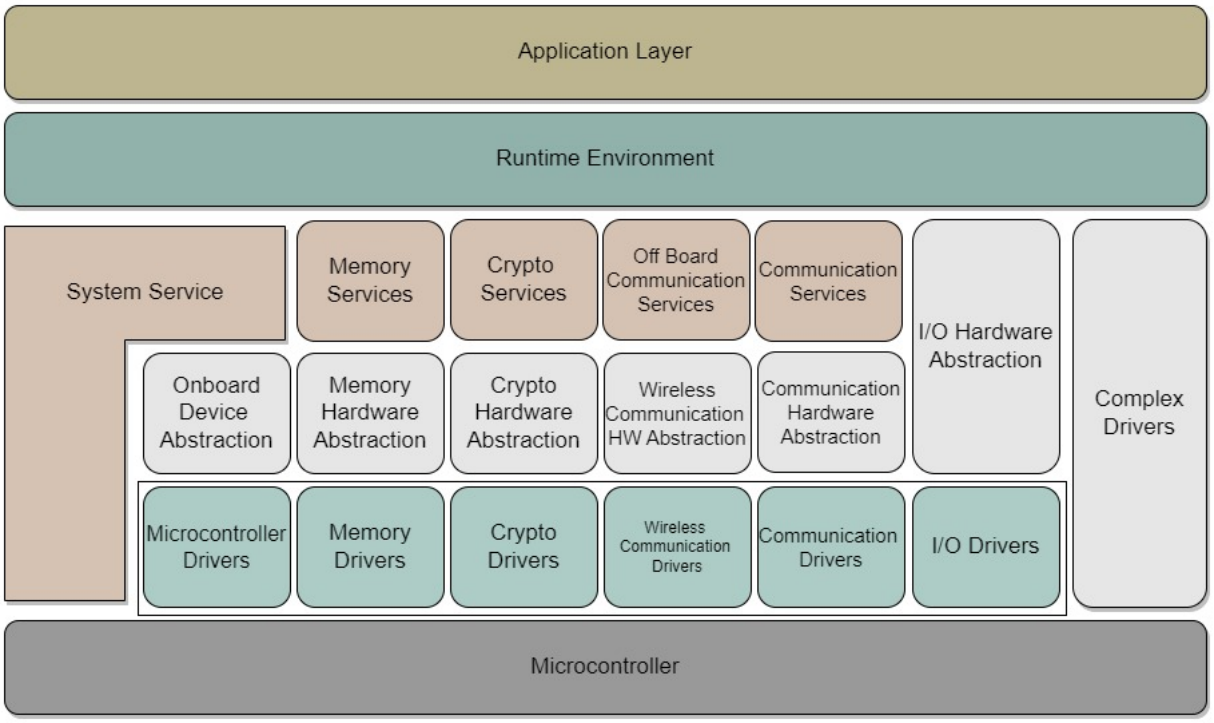
Централна процесорска јединица (енгл. *Central Processing Unit - CPU*) због своје поузданости и степена могућности управљања извршавањем задатака превасходно може служити за распоређивање задатака и надгледање, односно омогућавање безбедносних и сигурносних одлика које не могу бити подржане од стране јединица које тешко испуњавају детерминизам у извршавању. Такође, може бити укључена у део ланца задатака одређеног алгоритма, преузимати део рачунице, и прослеђивати податке на друге јединице обраде или ентитете система као што су актуатори, чије деловање мора бити адекватно контролисано [73]. За највиши степен безбедносно критичних функција надгледања рада система је неопходно интегрисати неки од микроконтролера посебно намењених за ове задатаке као што су *Aurix* [74], *RH 850-P1x* [75] и слични.

Разноврсност програмске подршке и физичких архитектура описана у претходним пасусима представља отежавајући фактор за имплементацију, интеграцију, одржавање и ажурирање аутомобилских система. Различите компоненте програмске подршке и физичке архитектуре су имплементиране од стране различитих тимова или чак компанија што неповољно утиче на интеграциони процес. Из тог разлога долази до потребе за стандардизацијом.

2.1.3 Стандардизација помоћу слојевите архитектуре

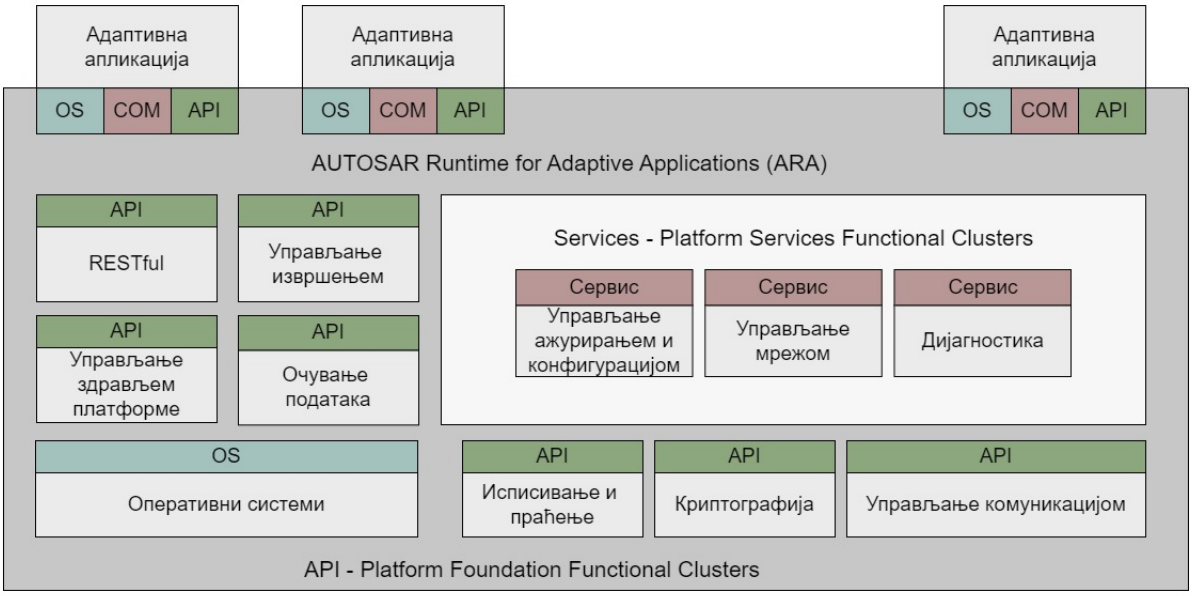
AUTOSAR [76] је архитектурални стандард формиран од стране групације водећих произвођача аутомобила са циљем да пружи безбедно, преносиво и прилагодљиво решење за *ADAS* платформе. На овај начин, фокус развоја аутомобилског система је на имплементацији компонената програмске подршке и физичке архитектуре, као што и мото овог конгломерата каже "*Cooperate on standards, compete on implementation*". То се омогућава стандаризовањем модула у слојевитој архитектури система, као што је приказано на слици 4, чиме се зависност највишег апликативног слоја од саме платформе физичке архитектуре као најнижег слоја редукује имплементацијом модула између ових слојева.

На најнижем нивоу се налази сам контролер, односно платформа физичке архитектуре



Слика 4: *Classic AUTOSAR* програмски стек

електронске контролне јединице. Затим на ову платформу належе основни слој програмске подршке (енгл. *Basic Software - BSW*). *BSW* се састоји из три нивоа: ниво апстракције микроконтролера (енгл. *Microcontroller abstraction layer - MCAL*), ниво апстракције електронске контролне јединице (енгл. *ECU abstraction layer*) и ниво сервиса (енгл. *Service layer*). *MCAL* има за циљ да више слојеве архитектуре учини независним од самог микроконтролера. Садржи интерне управљачке и покретачке програме који имају непосредан приступ микроконтролеру и унутрашњим периферијама. Ниво апстракције електронских контролих јединица омогућава програмска спрега за приступ управљачким програмима *MCAL* нивоа и садржи управљачке програме за екстерне периферије. На тај начин омогућава коришћење истих микроконтролера у различитим електронским контролим јединицама и пружа абстракцију меморијских, комуникационих и улазно-излазлних прикључака и чини више нивое независним од распореда физичке архитектуре електронске контролне јединице. Ниво сервиса садржи меморијске, дијагностичке, комуникационе, распоређивачке и друге сервисе који могу бити имплементирани на начин да буду у потпуности независни од платформе (енгл. *Hardware-agnostic*). Преко овог нивоа се из апликација приступа сервисима који належу на абстракционе нивое платформе, односно самим тим и на саму платформу. Након *BSW* слоја следи средњи слој (енгл. *Middleware*), тачније виртуелна магистрала (енгл. *Virtual Bus*) звана *Runtime Environment - RTE* која омогућава комуникацију између компонената програмске подршке на највишем слоју и сервиса. Задаци сваке компоненте се извршавају у ентитетима званим *Runnables* у којима је могуће вршити позиве за комуникацију са другим компонентама или коршћеним сервисима. Ови позиви омогућавају имплементацију независну од комуникационих механизама јер су спреге које се користе за комуникацију путем вирутелне магистрале исте без обзира на



Слика 5: *Adaptive AUTOSAR* програмски стек

комуникациони механизам који се користи у позадини.

AUTOSAR стандард покрива две палтформе, класичну и адаптивну (енгл. *Classic u Adaptive*). До сада је описана архитектура класичне *AUTOSAR* платформе, из 2003. године, чија је имплементација намењена за електронске контролне јединице које имају изражене безбедносне и сигурносне одлике (енгл. *Safety hosts*). Због тога има дизајном предвиђене модуле за надгледање, контролисање и у потпуности детерминистичко распоређивање извршавања задатака. Комуникација у класичној платформи је сигнално-оријентисана, унапред дефинисана и заснована на *Controller Area Network - CAN* и *Local Interconnect Network - LIN* магистралама. Са друге стране, адаптивна платформа [77] омогућава већу флексибилност у извршавању и комуникацији, па је предвиђена за платформе намењене за извршавање алгоритама високих перформанси (енгл. *Performance hosts*). С обзиром на то да се на оваквим платформама извршавају алгоритми и компоненте описани у претходном делу поглавља, чије информације су од већег значаја за инфо-забавни домен него компоненте које су задужене за надгледање и поуздано управљање читавим системом, решење за међудоменску комуникацију ће бити примарно фокусирано на примену адаптивне платформе.

Адаптивна платформа је заснована 2016. године на истим постулатима као и класична платформа, са циљем не да замени постојећу, већ да допуни спектар и недостатке у примени *AUTOSAR* стандарда [78]. Главна разлика је прелазак са сигнално-оријентисане на сервисно-оријентисану архитектуру комуникације, омогућавање бежичног ажурирања програмске подршке и прелазак са *CAN* и *LIN* магистрала окосница на *Ethernet*, као и прелазак са парадигме функционалног програмирања и програмског језика *C* у објектно оријентисану парадигму и програмски језик *C++*. Такође, класичан *AUTOSAR* мора бити постављен над посебним оперативним системом, док стек адаптивне платформе може бити имплементиран и над оперативним системима који подржавају *POSIX 51* стандард.

Архитектура нема подељен *BSW*, већ су модули нижег дела програмске подршке им-

плементирани сваки засебно и кроз програмску спрегу представљени компонентама програмске подршке које их користе као што је приказано на слици 5. У оквиру ових модула су, такође, имплементирани меморијски, дијагностички, комуникациони, распоређивачки, криптографски и други сервиси. Модул који свака имплементирана компонента програмске подршке мора да користи јесте *ara::exec*. Овај модул покреће процесе компонената, шаље сигнале за терминирање и контролише њихово распоређивање. Поред њега, други најважнији модул је *ara::com* — модул за комуникацију који имплементира претходно објашњени *RTE* и има исту сврху, да пружи спреге којима ће сакрити комуникационе механизме како апликација не би зависила од њих непосредно. Има две главне компоненте *Skeleton* и *Proxy* и у њима стандардом дефинисане функције које користе сервис и клијент респективно и којима је омогућена размена података без обзира на комуникациони протокол.

Једна од предности овако стандардизованих архитектура је омогућавање примене генеративног програмирања, односно развоја вођеног моделима. У *AUTOSAR* стандарду се користи имплементација *XML* језика прилагођеног за аутомобилску област. Овај језик се назива *AUTOSAR XML - ARXML* и служи за дизајн модела система, и то првенствено спрега за комуникацију. О њему и његовој улози ће бити више речи у наставку рада.

2.2 Еволуција инфо-забавног домена

Поред функционалности које пружају безбедну и поуздану вожњу, пред модерна возила се постављају и захтеви вођени струјама потрошачке електронике, са коначним циљем да аутомобил постане "паметни уређај на точковима" [79]. Из тог разлога се кроз инфо-забавни домен возачу, а и свим путницима, омогућава приступ разном садржају информативног и забавног карактера.

Технологије које се користе у овом домену се разликују од оних коришћених у *ADAS* домену у томе што треба да испуне висока очекивања у дизајну и имплементацији корисничког доживљаја и корисничког сучеља, без толике потребе за стриктношћу у виду поузданости, детерминизма и перформанси. Разлог је у томе што се штета која настаје отказом инфо-забавног система (изузимајући дигиталну таблу) своди на незадовољство у испуњености корисничког комфора, док грешка у *ADAS* домену може за последицу имати смртан исход. Због ових разлика у функционалним захтевима и саме доменске платформе се разликују у погледу примењених чипова, оперативних система и стандарда, па се у инфо-забавном домену често усвајају технологије већ потврђене у дигиталној телевизији и индустрији мобилних уређаја, што додатно повећава хетерогеност читавог аутомобилског система. Из тог разлога, примена *AUTOSAR* стандарда као одговор на хетерогеност у овом случају није добро решење.

Првобитно су QNX [80] и *Linux*, односно персонализовани систем базиран на *Linux* систему (*yocto*) представљали најчешће коришћена решења за оперативни систем у инфо-забавном домену, поготово када је у питању читав *HMI* систем. Најчешће се читав овакав систем извршава на јединственој платформи, а са обзиром да различити делови који чине овај систем имају нешто другачије захтеве, решење неретко бива засновано на комбинацији различитих оперативних система. Један од начина којим се ово постиже је посредовање

посебне платформе - хипервизора (енгл. *Hypervisor*) у ком се као гости покрећу различити оперативни системи [81, 82]. За саму дигиталну таблу (енгл. *Digital Instrument Cluster*), *QNX* је један од најраспрострањенијих оперативних система. Разлог је то што програмска подршка за дигиталну таблу подразумева усаглашеност са безбедносним и сигурносним захтевима за извршавање у реалном времену јер носи кључне информације за возача (брзину кретања, контролне поруке и слично) које не смеју бити угрожене [83]. Додатно, приметан је и раст примене решења базираног на *Automotive Grade Linux - AGL* [84].

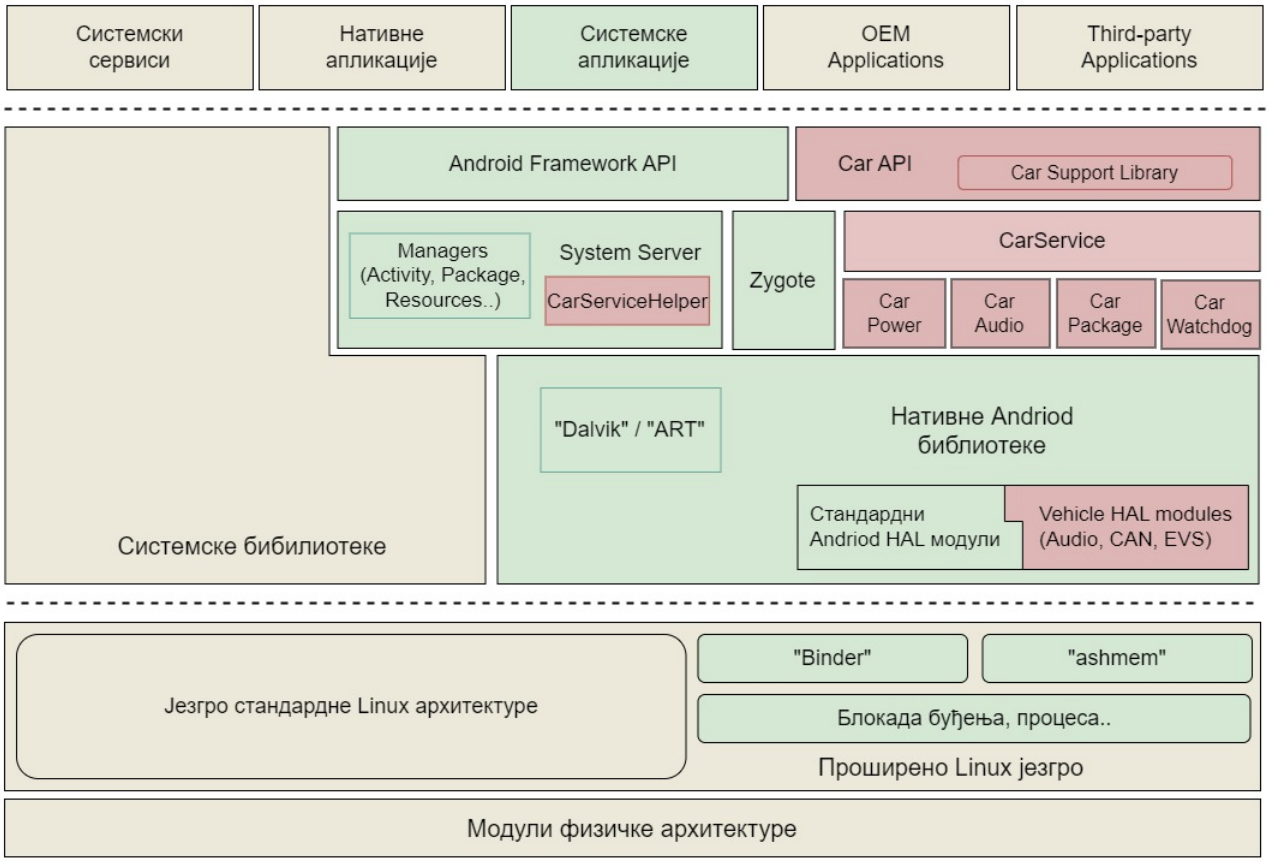
Међутим, тренутно најраспрострањенији оперативни систем за уграђене уређаје - *Android*, заузима прво место и на тржишту у аутомобилској индустрији. Као решење за инфо-забавни домен га усвајају многи велики произвођачи аутомобила попут *Volvo, BMW, Mercedes-Benz, Renult-Nissan Group, GeneralMotors, Stellantis, Honda, Hyndai-Kia Motors Corporation* и други. Правац развоја *Android* система наговештава још већу његову употребу, која подразумева преузимање свих делова *HMI*, па чак и потенцијалну интеграцију и у остале домене аутомобилског система. С обзиром на дужину циклуса развоја производа у аутомобилској индустрији, *Android* још увек не може у потпуности да замени досадашња решења, па су због тога у употреби и хибридне опције попут додатних екстерних модула физичке архитектуре или једноставно контејнера у оквиру постојећих оперативних система [85, 86] на којима је *Android* покренут.

Поред основног случаја употребе инфо-забавних система који представља праћење радио програма и репродукцију аудио садржаја, као и навигацију као први напреднији случај употребе, у претходној деценији тенденција за додатним проширивањем функционалности је интензивна. Као што је већ споменуто, усвајани су принципи корисничке електронике како би била омогућена контрола гласом и упаривање мобилних уређаја са опцијом приказа садржаја на екранима аутомобила. Додатно, улаже се много напора у стварање и раздвајање корисничких налога на систему који ће бити продужетак онога што је доступно на телефонима, телевизорима и другим паметним уређајима, како би све било повезано у јединствени екосистем. Овиме су отворена врата ка функционалностима који су по студијама случаја рађеним 2019. и допуњаваним у претходне две године тренутно главни правци развоја [87–89]. На првом месту је повезаност која има за циљ да омогући проток информација из читавог возила (што укључује и друге домене) кроз персонализоване налоге до облака и других спољашњих ентитета [90] који могу бити коришћени за удаљено надгледање перформанси у вожњи [91]. Други велики правац развоја представља потреба за омогућавањем проширене стварности [92] којом ће бити обогаћен приказ околине возила па и саме навигације.

Тражене функционалности се развијају у тој мери да је пуно пажње уложено у сам приказ садржаја и да се стандардизују одређени начини приказа попут контраста, дневног и ноћног осветљења екрана, величине приказаних ентитета и слично, како богатство садржаја не би одвраћало превише пажње и како би употреба била што једноставнија [93, 94].

2.2.1 *Android Automotive*

Неке од карактеристика *Android* оперативног система које га квалификују као добро решење за уграђене системе је подршка, односно могућност извршавања оперативног си-



Слика 6: Стек *Android* оперативног система уз *Android Automotive* додате појединости

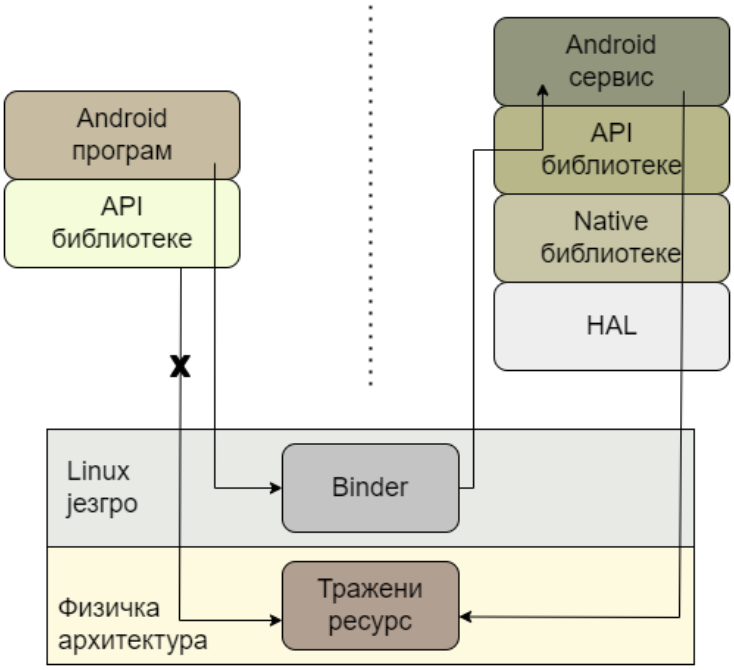
стема на много различитих уређаја, развој аутомобилске гране *Android* пројекта од стране *Google* компаније, подршка за већину програмских спрега за контролу графике (као што су *OpenGL* и *Vulkan*), као и добри алати који пружају могућност једноставног развоја апликација, због чега је заједница програмера који раде на *Android* оперативном систему широка [95]. Ове карактеристике су управо оно што омогућава испуњавање захтева система за инфо-забавни домен. *Android* је у возилима примењен на два начина, као *Android Auto* или *Android Automotive OS* који се суштински веома разликују [96]. *Android Auto* подразумева дељење апликација са мобилног уређаја на инфо-забавни систем представљајући их на оптимизован начин за приказ у возилу, како би одвлачиле што мање пажње и биле што једноставније за коришћење. *Android Automotive* је, са друге стране, оперативни систем који је покренут на платформи физичке архитектуре инфо-забавног система и дизајниран је посебно за употребу у инфо-забаним системима аутомобила. На овај начин, подразумевани *Android* стек је проширен ентитетима експлицитно прилагођеним за употребу у аутомобилским случајевима коришћења. Ови ентитети су *Car Manager*, *Car Services*, *Car Libs* и *Vehicle HAL*, који имају за циљ да обезбеде ефикаснију потрошњу и управљање електричном енергијом, бржи процес подизања система, доступност сензорских модула за перцепцију околине, чак и могућност коришћења типичних аутомобилских магистрала као што су *CAN* и *LIN*. Решења за употребу *Android* система у аутомобилској области су због прилагодљивости самог оперативног система веома различита. Тако неки радови

врше његову интеграцију у инфо-забавни домен [97] фокусирајући се на кооперативност уз друге оперативне системе на чипу, док неки радови у фокус стављају мапирање свих инфо-забавних функционалности и сервиса кроз средњи слој, пружајући типичан *Java API* [96].

Структура *Android Automotive* оперативног система је дата на слици 6. На најнижем нивоу се налази *Linux* језгро, а одмах изнад њега модули који садрже управљачке програме и повезиваче који користе физичке ресурсе, и стога су увек имплементирани од стране произвођача, тако да користе спрегу ка вишим слојевима стандардизовану *Android* системом. Поред стандардних модула као што су на пример *Bluetooth* и *GPS*, *Android Auomotive* уводи проширење модула додајући модуле за коришћење *CAN* магистрале, аутомобилских светала, визије околине возила и још неколико мањих. Затим, на нивоу изнад се налазе сервиси који се користе кроз управљчке компоненте чији су позиви изложени путем спреге радног оквира, тј. *Android API* библиотека. На исти начин се користе и проширења слоја сервиса за напајање, аудио, надзор и сличне употребом *Car Service* компоненте (односно *CarServiceHelper* компоненте уколико задатак мора бити извршен у оквиру компоненте системског сервера - *System Server*).

Иако у основи садржи *Linux* језгро, ово језгро садржи неке ентитете који га разликују од класичног *Linux* језгра као што су механизам за буђење, односно блокаду суспензије процеса који контролише управљање животног циклуса, а самим тим и рада батерије, механизме за руковање и дељење меморије и комуникационе механизме као што је *binder* [79]. *Binder* је проширење које карактерише рад у *Android* оперативном систему из разлога што уводи у потпуности јединствен механизам за комуникацију међу процесима [98]. Коришћење *binder* механизма прави нов модел употребе сервиса у *Android* уграђеним системима као што је приказано на слици 7. На овај начин, *Android* апликације немају непосредан приступ ресурсима, већ их добављају путем позива спреге радног оквира (*Android API*) кроз одговарајуће *Android* сервисе који су задужени за контролу појединог ресурса. Ресурс може бити део физичке архитектуре попут екрана, камере, звучника или слично, а може бити и само информација коју ствара сервис на основу алгоритма или функционалности која је у њему енкапсулирана. Из апликације се шаље захтев путем *binder* међупроцесног комуникационог механизма јер непосредан приступ није дозвољен. Овај захтев се обрађује на страни сервиса и у зависности од тога да ли апликација има одговарајуће дозволе које се декларишу по потреби (енгл. *Permissions*) добија одговор који садржи потребну информацију. Дозволе су ништа друго до низови карактера који додатно потврђују намеру одређеног процеса. Уколико је потребна информација ресурс саме физичке архитектуре, *Android* сервис њима управља кроз *HAL* имплементацију у којој су садржани потребни управљачки програми (енгл. *Drivers*).

Комуникацију путем *binder* механизма је могуће описати на основу језика за дефинисање спреге *Android Interface Definition Language - AIDL* [99]. *Android* омогућава генерисање сервиса на основу *AIDL* модела у програмским језицима *C++*, *Java* и *Kotlin* и чини комуникацију између клијента и сервиса независном од конкретног програмског језика у ком су написани. То такође значи да клијент и сервис могу да буду написани у различитим програмским језицима и комуникација путем *binder* механизма, а настала генерисањем



Слика 7: Принцип рада сервиса у *Android* оперативном систему

од *AIDL* модела ће бити могућа. Једини начин да се међупроцесна комуникација настала од *AIDL* модела изврши јесте да клијент затражи извршавање методе на сервис страни. По клијентском позиву методе, долази до маршаловања параметара методе, односно разлагања параметара на примитивне објекте, тј. типове, који се на тај начин коришћењем *ioctl()* путем *binder* повезивача преноси и окида нит у процесу сервиса која чека на захтеве. У сервису се врши такозвано демаршаловање примитивних објеката и прозива се конкретан метод који садржи енкапсулирану имплементацију фунцкионалности сервиса. Након што је сервис обавио свој задатак, повратна вредност, тј. резултат се враћа назад клијенту истом путањом, "одмотавањем"позваних функција поново користећи процес серијализације/десеријализације.

Постоје три врсте чвора *binder* повезивача */dev/binder*, */dev/hwbinder* и */dev/vndbinder*. Првом је додељена улога у комуникацији искључиво између процеса из радног оквира, други има улогу у комбинацији комуникације између процеса из радног оквира и партиције где су смештене извршне датотеке разних добављача, Трећи чвор постоји од *Android* верзије 8, и задужен је за комуникацију између процеса покренутих из партиције добављача.

Постоје три врсте сервиса у *Android* систему:

- Сервис у првом плану (енг. *Foreground*) — Обавља радњу која је видљива самом кориснику, а ове радње се обављају чак и када корисник није непосредно у додиру са апликацијом. Ови сервиси приказују обавештења кориснику како би он био свестан да је нека операција у току. Обавештење се не може отклонити докле год сервис не заврши извршавање. Пример за овај сервис би могао бити преузимање података са интернета или пуштање музике.

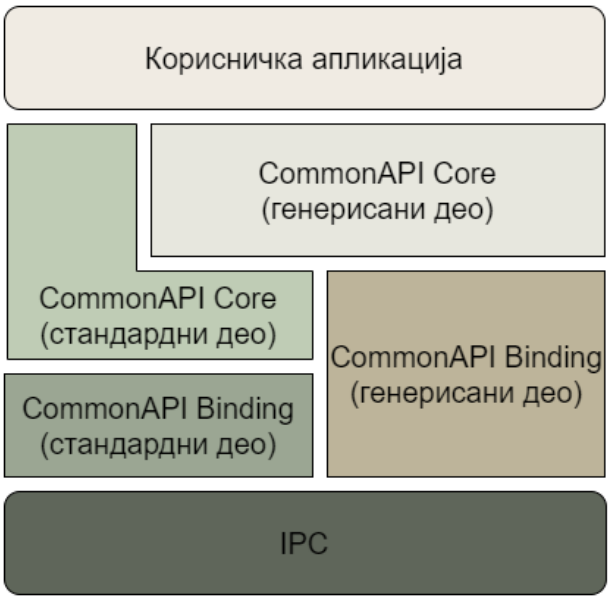
- Позадински сервис (енгл. *Background*) — Овај сервис обавља радње које нису видљиве самом кориснику. Такође, он не може бити жив уколико корисник изађе из апликације. Пример за овај сервис би могла бити оптимизација простора у складишту меморије. У новијим верзијима не представља предложени приступ из безбедносних разлога.

- Увезани сервис (енгл. *Bounded*) — Ови сервиси пружају искључиво вид клијент-сервер структуре у којој клијенти могу да затраже одређене податке или услуге од стране сервера, а он им то омогућава. Овај вид услуге може се одвијати и између одвојених процеса. На сервис могу да се повежу један или више клијената, а он је жив док је барем један од њих повезан на њега.

Треба такође водити рачуна и о томе на који начин и у ком тренутку ће се одређени процес (било да је то сервис или не) активирати. Након што програм за подизање система (енгл. *Bootloader*) учита *Linux* језгро, први процес који оно покреће се назива *init*. *Init* има главну улогу да подеси окружење, приступне тачке и директоријуме, покрене неке основне системске *C++* сервисе међу којима је најпре *ServiceManager*, а убрзо након тога и користећи *Android Runtime - ART* покреће процес који се назива *Zygote*. *Zygote* покреће *System Server* који даље покреће готово све *Java* системске сервисе и инстанцира управљачке компоненте из спреге радног оквира који је доступан апликацијама. На овај начин, покренути сервиси се могу одмах регистровати у претходно покренут *ServiceManager* и на тај начин бити доступни у свим процесима у систему. Додатно, имплементиране *C++* сервисе је могуће додати у *init.rc* скрипту чиме ће они бити покретани у почетним фазама подизања система. Овде је потребно бити пажљив и покренути сервис након других елемената који се извршавају у овој фази, а од којих сервис може да зависи, попут приступних тачака, употребе мреже и слично. Сервиси покренути кроз *init* процес могу бити регистровани путем *ServiceManager* компоненте, само уколико се покрећу након што је она сама створена.

2.2.2 *CommonAPI*

Једна од најзначајнијих алијанси за развој решења у инфо-забавном и домену повезаности возила и околине је *COVESA* [100], односно бивши *GENIVI* чија су решења често део архитектуре програмске подршке аутомобила [101]. Решења ове заједнице последица су бројних истраживања спровођених у корелацији са произвођачима аутомобила, најчешће су отвореног програмског кода и примењена су у индустрији. Једно од таквих решења је *vsomeip* [102], имплементација сервисно-оријентисаног механизма комуникације преко *SOME/IP* протокола о ком ће бити више речи у наставку. Такође, *CommonAPI* [103] је решење за модул средњег слоја имплементиран од стране *COVESA* заједнице заснован на *Franca* радном оквиру [104] који омогућава комуникацију између процеса кроз сервисно-оријентисану архитектуру. Динамичко понашање програмске спреге је обезбеђено дефинисањем спрега, стања и транзиције између клијента и сервиса [104] употребом језика за дефинисање спреге званог *Franca Interface Definition Language - FIDL*. *FIDL*, као и



Слика 8: *CommonAPI* стек

ARXML и *AIDL*, омогућава флексибилну примену развоја вођеног моделом о ком ће више речи бити у наставку.

Еквивалентно другим модулима средњих слојева, *CommonAPI* има за циљ да уклони зависност имплементације саме апликације од имплементације комуникациног механизма који се налази на нижим слојевима. Такође, интерно је подељен на два дела: *CommonAPI Core*, у ком не постоје зависности од комуникационог протокола који се користи за размену података између сервиса и клијента и *CommonAPI Binding*, који непосредно належе на имплементацију комуникационог протокола и у ком је садржана сва зависност од истог. Архитектура *CommonAPI* модула средњег слоја се може видети на слици 8. Компоненте програмске подршке на врху стека користе *Stub* и *Proxy* компоненте за имплементацију комуникације на серверској, односно клијентској страни респективно. Тренутно, протоколи које *CommonAPI* подржава су *SOME/IP* и *D-Bus*, без промене корисничке програмске спреге у зависности од тога који од протокола је у употреби.

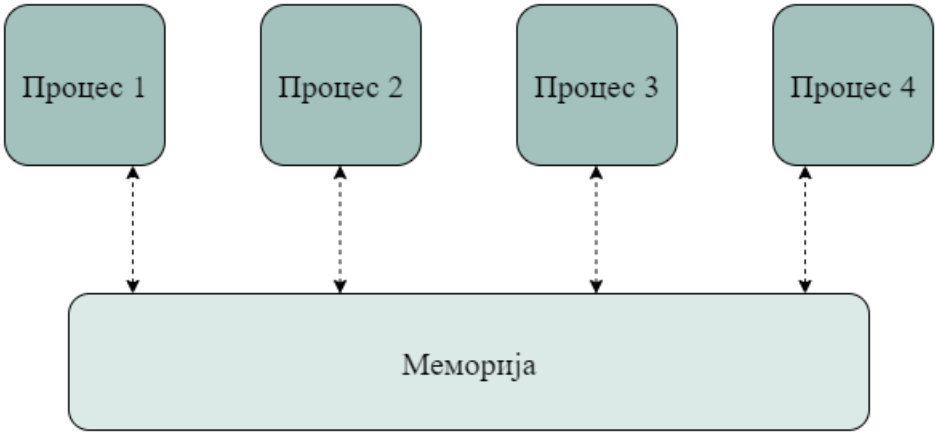
2.3 Комуникација унутар возила

Стандардизација формирана кроз слојевиту архитектуру чини да апликативни слој не зависи од комуникационих механизама, и омогућава позиве комуникационе програмске спреге без потребе за дубљим разумевањем комуникационих сервиса на нижим слојевима. Међутим, разумевање комуникационих магистрала и протокола и правилно одабирање њиховог коришћења у одговарајућим случајима имају велику улогу у функционисању целокупног система. Из тог разлога, неопходно је дефинисати случајеве употребе различитих комуникационих магистрала, протокола и механизама узимајући у обзир архитектуралну поставку система (као што су број, удаљеност и природа јединица за обраду, број језгара, итд), брзину комуникације, поузданост, проток као и контекст података који се размењују [105]. Типични комуникациони путеви којима су размењивани подаци између

Табела 1: Комуникациони механизми у возилима

Комуникациони механизам	Распоред			Природа податка		
	Исто језгро исти домен	Разл. језгра исти домен	Разл. језгра разл. домен	Обичан	Са сензора	За актуатор
Дељена меморија	✓	✓	НП	✓	НП	НП
NoC базиран на пакетима	НП	✓	НП	✓	НП	НП
CAN, FlexRay и LIN	НП	✓	✓	✓	✓	✓
UART, SPI и I2C	НП	✓	НП	НП	НП	✓
PCIe	НП	✓	НП	✓	НП	НП
Ethernet	НП	✓	✓	✓	✓	НП
SOA RPC	✓	✓	✓	✓	НП	НП

* НП - није примењиво
* Обичним податком се сматра податак размењиван међу компонентама и алгоритмима који не долази са сензора и циљ му није побуда актуатора



Слика 9: Дељена меморија

компонената у аутомобилима су представљени у табели 1.

2.3.1 Дељена меморија

Сваки процес који се извршава на уређају има додељен сопствени меморијски простор. За комуникацију између различитих процеса који се извршавају на истом језгру најчешће се као преносни медијум користи дељена меморија, а присутни су и остали механизми за међу-процесну комуникацију попут редова за поруке, pipeline-ова, итд. Ови принципи комуникације се, такође, користе и за комуникацију процеса, односно компонената програмске подршке које се извршавају и на различитим чиповима уколико им физичка спецификација уређаја то дозвољава.

Размена података се врши као што је приказано на примеру на слици 9, захтевима за

уписивање или читање из предодређене меморијске локације, при чему оба захтева пружају информацију о валидности извршене акције. Највеће предности овог комуникационог механизма су једноставна имплементација и мало кашњење, уз могућност размене како малих тако и великих података. Са друге стране, усклађивање [106] и временски недетерминизам су највеће мане приступа [107]. Механизми за усклађивање, у циљу спречавања блокада и застоја у петљи, морају бити имплементирани и контролисани у самом процесу. Детерминистично умрежавање је представљено као озбиљна претња за парадигму дељене меморије као што је приказано у раду [108], баш због временског недетерминизма који је у аутомобилској индустрији изузетно неповољна особина.

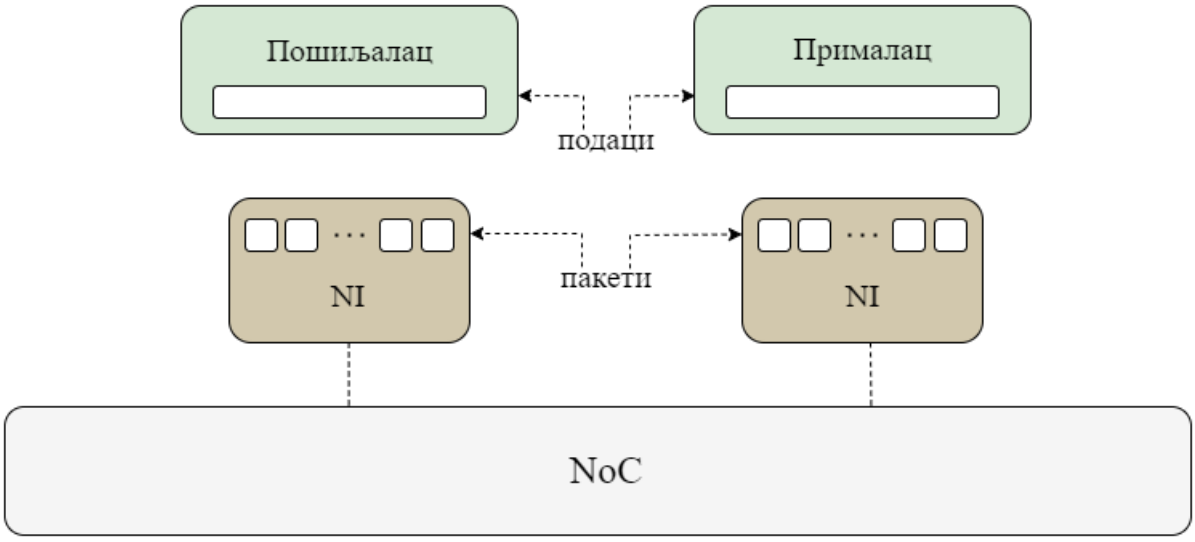
2.3.2 *Network-on-Chip*

Са развојем система чипова и повећавањем броја језгара на самим чиповима, комуникација међу процесима, односно компонентама програмске подршке је постала сложенија. Стога, потребна су прилагодљива решења за размену података међу многобројним компонентама и процесорским језгрима на којима се ове компоненте извршавају. Једно од таквих решења је *Network-on-Chip - NoC* који у оваквим случајима коришћења показује бенефите у односу на традиционалне неблокирајуће мрежне топологије [109]. *NoC* представља једноставну парадигму прослеђивања порука, комутације пакета базирану на рутирању којима процеси размењују инструкције и релевантне податке. Принципи комуникације интернет протокола су примењени у оквиру механизама оваквог преноса информација између модула система чипова, где је свака порука издељена на мање јединице како би се омогућио већи проток и мањи застој у мрежи [110]. Уколико је *NoC* парадигма искоришћена на систему на чипу, архитектура, односно топологија којом су повезана процесорска језгра и чипови такође мора бити узета у обзир, јер може утицати на кашњења у приступу дељених ресурса [111, 112]. *Network Interface - NI* [113] је ентитет одговоран како за дељење порука у мање целине, тако и за њихово касније спајање као што је приказано на слици 10. *NI* је компонента круцијална за омогућавање проширивости и поновне употребе јер она раздваја дизајн језгара од дизајна мреже.

2.3.3 Магистрале у аутомобилским системима

2.3.3.1 *Controller Area Network - CAN* и *FlexRay* Иако су прилагодљивост, проширивост и једноставност имплементације најважнији са потрошачког аспекта, у традиционалним сегментима развоја аутомобилске индустрије су детерминизам и поузданост ипак карактеристике од највећег значаја. Из тог разлога су *CAN* и *FlaxRay* биле најважније магистрале које чине окосницу и омогућавају комуникацију између круцијалних компонената [114].

CAN [115]је због свог пропусног опсега од 1Mb/s (односно 10Mb/s или 20Mb/s за проширење протокола флексибилном брзином за случај *CAN/FD* [115] или *CAN XL* [115] стандардизованог преноса, респективно) повољан за пренос претежно мањих података, који у виду сигнала путују између компонената [114]. Битска арбитража која се одвија у преносу на основу приоритета идентификатора који се шаље *CAN* сигналом омогућава



Слика 10: *Network-on-Chip*

поузданост и зато се користи за пренос података у кључним деловима аутомобила одговорним за погон и вожњу [116]. Са друге стране, топологија коју подржава *CAN* је дељена магистрала, што може представљати отежавајући аспект по погледу прилагодљивости система. Додатно, истраживачи и инжењери који фаворизују ову магистралу настоје да унапреде њене перформансе најпре по питању пропусног опсега као што је наведено на примеру *CAN/FD* и *CAN XL* проширења, али и по питању унапређења сигнала *CAN SIC* [115] и подржавања различитих топологија.

FlexRay [117] је такође магистрала која се користи у случајевима када је потребно остварити поузданост, јер подржава аспект временске критичности, односно временског детерминизма. Из тог разлога се користи за комуникацију са неким од кључних компонената као што су кочиони системи чије је време одзива критично за читав систем [118]. Проток који остварује ова магистрала је 10Mb/s, а подржане топологије су магистрала, звезда, тачка на тачку и хибридне.

2.3.3.2 *Local Interconnect Network - LIN* Још једна од традиционалних магистрала је *LIN* [119]. Међутим, за разлику од *CAN* и *FlexRay* поузданост није оно што је њена главна карактеристика, већ једноставност и цена [116]. Зато се користи за повезивање компонената које немају захтевне задатке, као ни критичне временске одреднице попут контроле прозора, седишта, климе, итд. Проток који остварује ова магистрала је до 20Kb/s, а најчешћа топологија је водећег-пратећег уређаја (енгл. *Master-slave*).

2.3.3.3 *PCIExpress* Поред поузданости, повећавање броја компонената, њихових функционалности и количине података коју размењују доводи до потребе за комуникационим магистралама већег протока. Обезбеђивање протока од 16Gb/s у трећој, а уједно и комерцијално најраспрострањенијој генерацији у аутомобилској индустрији, чини *PCIe* магистралу изузетно квалификованом за примену у случајевима размене велике количине различитих типова података. Такође, шеста генерација *PCIe* магистрале има план да

оствари проток од 64Gb/s, што је далеко више од било које друге коришћене магистрале. Рад ове магистрале заснован је на *DMA* приступу, што омогућава непосредан приступ без посредсва додатног рачунара, односно јединице обраде. Ипак, још увек није примењена у аутомобилима у свом пуном потенцијалу из разлога што не одговара неким од карактеристика као што су физичка дужина кабла, опсег радне температуре, поузданост и шум. Међутим, најављене карактеристике будућих генерација ове магистрале и еволутивни помераји ка зоналној и централизованој архитектури, који елиминишу раздаљину између компонената као највећуману ове магистрале, заједно стварају потенцијал за све већу примену *PCIe* у возилима нове ере.

2.3.3.4 ***Ethernet*** Потреба за већим протоком, дометом и масовном употребом дали су простора за примену глобално најраспрострањеније магистрале и стандарда протокола и у аутомобилској индустрији [120]. Ипак, потрошачки *Ethernet* сам по себи није могао да задовољи критеријуме аутомобилске индустрије по питању брзине преноса, шума, непотребно великог домета и броја жица па самим тим и цене кабла и могућности поузданог рада у неповољним условима (температуре од -40 до +125) [121]. Стога је прво *BroadR-Reach*, а затим и *IEEE* имплементирао нови стандард [122] посебно дизајниран за Automotive *Ethernet* који одговара на претходно наведене недостатке. Једна од главних карактеристика је што су два или четири пара жица замењени јединственом упреденом парицом, а проток износи 100 или 1000 Mb/s, за *100* и *1000 Base-T1* варијанте *IEEE* стандардизованог *Ethernet* протокола из године 2015/16 [123], односно 2.5, 5 И 10 Gb/s за вишегигабитни пренос путем *IEEE Base-T1* варијанти из 2020. године [124] респективно.

Постоје и принципи и стандардизоване *Ethernet* парадигме које омогућавају временски детерминизам јер *Ethernet* у основи због начина преноса то не може да обезбеди [125]. Као такве су од изузетног значаја за аутомобилски систем јер омогућавају имплементацију безбедносно-критичних апликација и апликација са стриктно постављеним роковима извршавања. Најкоришћенији механизми за овакав тип преноса путем *Ethernet* протокола су *Time-Triggered Ethernet - TTEthernet* и *Time-Senitive Networking - TSN*. Иако оба механизма почивају на сличним принципима, начини на који су ти принципи спроведени се разликују. Битно је нагласити да не постоји победник у поређењу ова два механизма, као и да ниједан од механизама не гарантује апсолутни детерминизам као такав, јер и даље постоји фактор распоређивања који мора бити изузетно прецизан како би остварио ефикасност и не може бити идеалан [126].

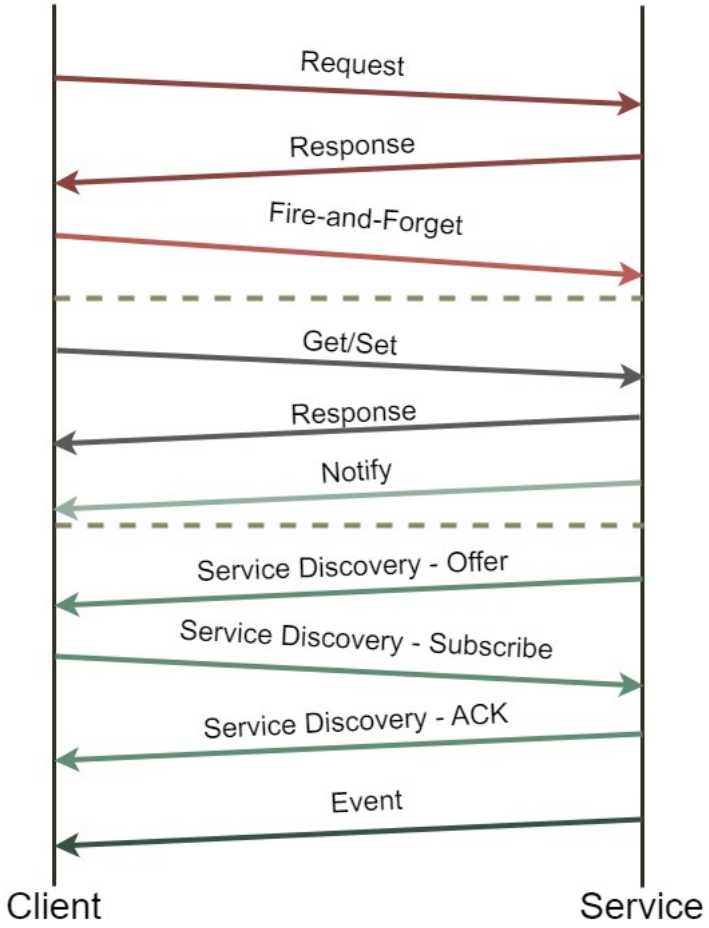
TSN је скуп стандарда који омогућавају усклађивање односно синхронизацију времена, класификацију и суживот различитих врста саобраћаја и распоред размене података [127]. За усклађивање времена се користи *Precision Time Protocol - PTP* [128] који се заснива на слању четири типа порука: *sync*, *follow up*, *delay request* и *delay response*. Ове поруке започињу слањем са уређаја чије је време одабрано као најтачније и најпрецизније, а имају за циљ периодично поравнање других уређаја са овим временом. Да би се подржао пренос садржаја различитог приоритета, саобраћај се планира и уједначава. Ово се постиже тиме што се *Time/Aware Shaper - TAS* стандардом [129] уводи најприоритетнија класа критичног контролног саобраћаја (енгл. *Control Data Traffic - CDT*). Постојање различитих класа

саобраћаја уводи могућност резервације тока за пренос података (енгл. *Stream Reservation Protocol - SRP*) [130] тако што је статички предефинисано распоређивање најприоритетнијег саобраћаја, а остатак саобраћаја се распоређује у реалном времену тако што је до 75% преносног медија доступно за осетљиву класу саобраћаја, док је 25% доступно за спорадичан саобраћај најнижег приоритета. За то је потребно извршити усклађивање осетљивог и спорадичног саобраћаја које се изводи на основу *TAS* и/или *Credit-Based Shaper - CBS* алгоритма [131]. Примена *CBS* алгоритма се спроводи тако што се најпре користе 3 бита за приоритетизацију из *VLAN* ознаке одређене на основу *IEEE 802.1Q* стандарда [132]. Ови битови служе да раздвоје до 8 класа саобраћаја. У случају употребе *TSN* механизма користе се само две класе (A и B) за осетљив и једна класа за спорадичан саобраћај. На основу ових класа, мрежни спрежни пролази за излазни саобраћај смештају податке у одговарајуће редове за пренос. Из ових редова подаци се узимају и делегирају даље до преносног медијумa. Редослед преноса података из одговарајуће класе, односно реда, одређује се на основу контролисања кредита додељеног подацима из сваког реда, који се линерално смањују и повећавају на основу дужине чекања и регулаторног параметра различитог за сваки ред, чиме се задаје и сам приоритет. На овај начин се саобраћај свих класа преноси усклађено у зависности од приоритета, а опет без пристрасности.

TTEthernet је стандардизована имплементација *Ethernet* протокола која сегментира саобраћај на три типа од којих је најприоритетнији временски осетљив саобраћај (енгл. *Time-Triggered - TT*). Као такав, он има приоритет и неопходно га је предефинисати како би се доступност канала за пренос у одговарајућим временским роковима осигурала. Поред тога, присутни су и саобраћај осетљив на кашњења (енгл. *Rate Constrained - RC*) и спорадичан саобраћај (енгл. *Best Effort - BE*) који је најмањег приоритета [133]. Такође, постоје и разни принципи распоређивања саобраћаја на основу класе којој припада као и механизми попут замене приоритета, претпражњења и блокирања у реалном времену, који могу да спрече евентуалне конфликте између различитих класа саобраћаја и који по стандардима *TSN* механизма нису предвиђени [127].

Актуелни су такође и покушаји имплементације *Ethernet* протокола *10 Base-T1S* [134] који би имао мањи проток од свега 10Mb/s, напајање путем жица за пренос података и који би евентуално могао да замени *CAN* магистралу по питању поузданости преноса омогућавајући други приступ у арбитражи за избегавање судара на магистрали, али не и по питању цене и алата за имплементацију и тестирање. Важно је напоменути да *Ethernet* суштински нема за циљ да замени и избаци из употребе све постојеће магистрале већ представља окосницу, односно костур комуникације у возилу, спајајући сензоре, јединице обраде и микроконтролере коегзистирајући са другим магистралама које имају специфичну употребу.

2.3.3.5 Остале С обзиром на сложеност захтева у аутомобилском свету, постоји још много комуникационих магистрала и протокола који су пронашли примену у различитим случајевима. *MOST* је служио за пакетску размену саобраћаја у сврху репродукције аудио и видео сигнала пропусним опсегом од највише 150 Mb/s [135]. Међутим, *Ethernet/IP* се врло брзо показао као медијум већег потенцијала, па је *MOST* постао наслеђена тех-



Слика 11: *SOME/IP* механизми комуникације

нологија у појединим решењима [136]. Такође, постоје и разне серијске магистрале попут *UART, SPI, I2C*, пропусних опсега од 20Kb/s, 250Kb/s - 2Mb/s и 100Kb/s - 5Mb/s респективно, које се користе за спајање процесора са периферијским уређајима. Најчешће налазе употребу у омогућавању конфигурације примопредајничких компонената, сензора, актуатора и слично од стране процесора и у дијагностици [127].

2.3.4 Сервисно-оријентисана архитектура

Традиционални приступи на којим почивају конвенционалне комуникационе магистрале попут *CAN* су сигнално-оријентисани. Међутим, нова генерација возила препознаје сервисно-оријентисану комуникацију као приступ који доноси једноставнији дизајн, флексибилнију реализацију и ефикасније одржавање, енкапсулирајући посао који извршавају одређене компоненте у сервисе и послужују остале компоненте које су у улози клијената [137]. Из тог разлога су принципи из web и рачунарства у облаку пренети и у аутомобилски свет. Имајући у виду разлику у захтевима и њиховој приоритетизацији у овим областима, у програмској подршци аутомобилског система је имплементиран и додатни протокол који служи да омогући сервисно-оријентисану архитектуру. Овај протокол је *SOME/IP* и пратећи његов стандард комуникација се извршава на основу три механизма као што је приказано на слици 11:

- Метод (енгл. *Method*) — омогућава клијентима да извршавају позиве удаљених процедура, тј. функција (енгл. *Remote Procedure Call - RPC*) и добију резултат. Овиме је специфичан посао смештен у сервис а клијент може да користи удаљене процедуре а да притом нема потребу да зна где се тачно сервис налази (може бити на истој или удаљеној платформи). Такође, поред *RPC* захтева који враћају одговарајући резултат, постоје и они који не очекују никакав резултат него имају за циљ само да доведу до покретања одређене функционалности (енгл. *Fire-and-forget*).

- Догађај (енгл. *Event*) — даје могућност сервису да обавести претплаћене клијенте и достави податке асинхроно по потреби.

- Поље (енгл. *Field*) — омогућава клијентима да имају приступ пољима сервиса на три начина, кроз захтев за добављање вредности, захтев за постављање вредности и обавештење о промени вредности. Сервис на основу функционалности која је у њему енскапсулирана управља сопственим пољем, а клијент може да приступи вредности коју то поље чува са циљем да на захтев добави тренутну вредност, постави нову вредност и да се претплати и буде обавештен на сваку промену вредности која се догоди. Ова три начина нису међусобно искључива, што значи да кроз један комуникациони механизам поља вредност може бити манипулисана различитим приступима.

SOME/IP протокол се састоји из три модула: базични, модул за откривање сервиса и трансформатор [138]. Базични модул [139] описује понашање горе наведених механизама комуникације и серијализацију корисних података који се преносе, водећи рачуна о њиховом поравнању и пољима која служе да опишу дужину, редослед бајтова и слично. Трансформатор модул [140] служи за серијализацију, односно десеријализацију поруке у складу са очекиваним форматом *SOME/IP* протокола који је дат на слици 12 и спецификацију уграђених механизама функционисања као што је руковање грешкама на пример.

Модул за откривање сервиса [141] је модул који се користи за реализацију комуникације и слања података путем догађаја, имајући за главни задатак управљање доступношћу сервиса. Састоји се из три фазе: фаза чекања при иницијализацији, фаза понављања и главна фаза, а своди се на слање и одговоре на поруке клијената за проналазак, односно у случају сервиса за нуђење услуге. У првој фази нема слања никаквих порука које служе за успостављање и потврду доступности, што значи да клијент у овој фази не шаље поруке за проналазак сервиса за који је заинтересован, а сервис не шаље поруке нуђења које објављују да је активан, већ и клијент и сервис само чекају неко насумично одређено време. У случају да је сервис за који је клијент заинстересован у некој од поодмаклих фаза извршавања и клијент ипак добије поруку која објављује активност сервиса, клијент прескаче фазу понављања и иде у главну фазу извршавања. Сервис који прими поруку за потраживање од клијента у поодмаклој фази игнорише тај захтев и наставља извршавање у репетитивној фази. У фази понављања клијент шаље вишесмерне (енгл. *Multicast*) поруке за потраживање, све док не добије поруку од стране сервиса која обавештава да је услуга активна, односно доступна или док бројач захтева за потражњу не пређе највиши задати износ. Сервис функционише на сличан начин, с тим да уколико прими поруку о

Слика 12: Формат *SOME/IP* поруке

потражњи услуге од одређеног клијента, шаље једносмерну (енгл. *Unicast*) поруку за потврду активности, након чега наставља са слањем вишесмерних порука за нуђење услуга. У главној фази клијент и сервис раде тако да смање оптерећење мреже, клијент више не шаље поруке за потражњу услуге, док сервис наставља да нуди услугу, односно обавештава о својој активности и одговара на захтеве нових клијената.

Ипак, кашњење изазвано откривањем сервиса се може израчунати али није детерминистично, односно предефинисано [138]. Поред тога, откривање сервиса не гарантује ваљаност у раду из разлога што сервисно-оријентисана архитектура постоји да би повећала динамику и флексибилност система, што значи да сервиси и клијенти могу бити терминирани и покретани изнова и нема механизама који могу испратити ту динамику на прави начин а да притом представљају минимално оптерећење за читаву мрежу. Овај модул представља један корак ка праћењу активности и доступности сервиса. Формат заглавља дефинисан у модулу за откривање сервиса је приказан на слици 13.

При регуларној размени порука путем *SOME/IP* протокола постоји идентификатор поруке који садржи информације о томе који је тачно циљани сервис, као и који је на пример циљани догађај или метод у питању.

SOME/IP заглавље се састоји од идентификатора захтева који у себи садржи идентификатор клијента и идентификатор сесије. Служе да означе једнозначну комбинацију комуникације између сервера и клијента, па се у одговорима ово поље идентификатора захтева копира. При откривању сервиса идентификатор клијента се не користи, већ само идентификатор сесије како би се могло утврдити да ли је дошло до поновног покретања ради праћења инстанце откривања сервиса. Потом постоје поља која која служе за дефинисање верзије *SOME/IP* протокола и верзије спреге сервиса која је у случају откривања сервиса увек постављена на 0x01. Поље које одређује тип поруке говори о томе да ли је реч о захтеву са или без одговора, одговору, обавештењу или коду грешке (у случају откривања сервиса то је увек обавештење зато што служи само за руковање догађајима као механизму комуникације), док поље за повратну информацију служи да укаже на то да ли је захтев успешно послат.

Поред ових поља, при откривању сервиса постоје и поља показатеља да почињу специ-

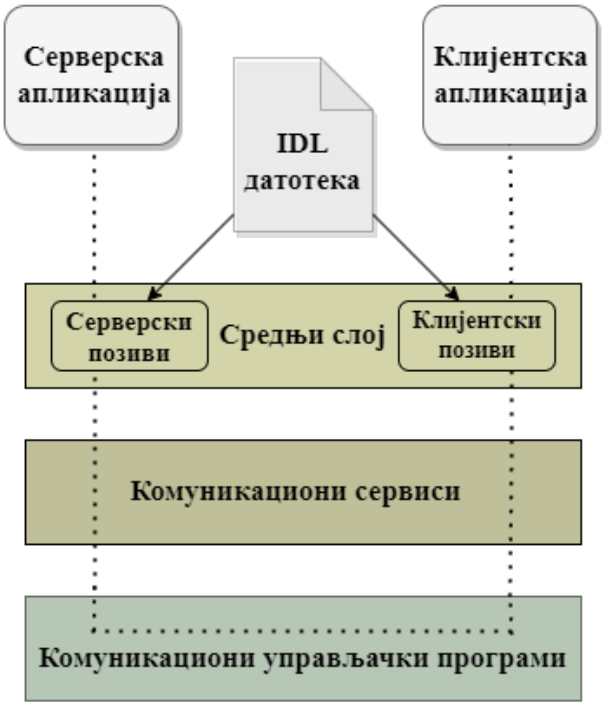
Слика 13: Формат *SOME/IP* поруке за откривање сервиса

фични подаци за овај модул као шту су поља за дужину и саме низове уноса и опција која говоре о којој поруци је реч (захтев за потражњу или нуђење сервиса) и пружају разне додатне информације попут идентификатора инстанци сервиса, задатог трајања *TTL* и слично.

SOME/IP је имплементацијски подржан од *AUTOSAR* и *COVESA* конзорцијума кроз већ поменуте *ara::com* и *CommonAPI* модуле средњег слоја. Начин на који је та комуникација обезбеђена у самим модулима прати исте принципе. С обзиром на то да сврха средњег слоја представља раздвајање имплементације корисничких апликација од имплементације комуникационих механизама, примена аутоматизованог генерисања програмског кода који се користи да ова два аспекта интегрише је очигледна. На основу интуитивних језика за описивање комуникационих спрега као што су поменути *ARXML* и *FIDL* генерише се програмски код независтан од имплемнетације апликације као и имплементације комуникационог механизма, односно протокола. Овај програмски код служи да апликацији обезбеди позиве који омогућавају сервис-клијент комуникацију. Најчешће се генеришу две одвојене компоненте, једна коју користи сервер и друга коју користи клијент као што је приказано на слици 14. У истом маниру функционише и међупросецна комуникација путем *binder* механизма у *Android* оперативном систему користећи *AIDL* за дефинисање спрега од којих се генеришу компоненте потребне за остваривање комуникације између клијента и сервиса.

2.4 Генеративно програмирање и развој вођен моделима

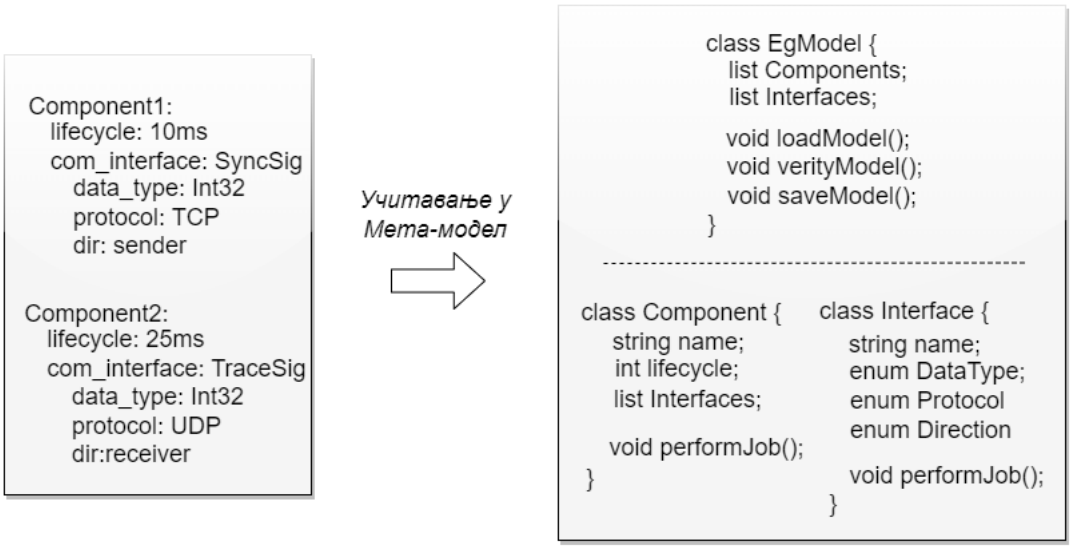
Аутоматизација у виду генерисања програмског кода је тренд широко примењен у аутомобилској индустрији као последица потребних стандардизација, потребе за олакшаним развојем који омогућава усмеравање фокуса на имплементацију као и октривање инова-



Слика 14: Пример рада језика за дефинисање спреге

тивних приступа у погледу аутоматизације процеса у паметним возилима. Генеративно програмирање за почетак скреће пажњу са инжењеринга јединствених компонената и система на скупове сродних система и компонената који се могу повезати у одређени домен. Затим, као други корак уводи могућност коришћена генератора за максимизацију аутоматизације у развоју [142]. Оваква аутоматизација најчешће неће давати компоненте у потпуности спремне за коришћење, већ је потребно додати одређену количину ручног рада, али свакако олакшава читав процес [143]. Системи груписани у породице, односно домене најчешће су описани језицима специфичним за конкретан домен (енгл. *Domain Specific Language - DSL*). *DSL* језици се користе за дефинисање и описивање ентитета, њихових стања и односа међу њима, имају специфицирану синтаксу, семантику и нотације и често нуде механизме и алате за проверу грешака и визуализацију описаног [144]. Примери ових језика су *HTML*, *XML*, *VHDL*, *SQL* и многи други. За разлику од генеративног програмирања које се бави функционалностима посматрајући породице система, развој вођен моделом (енгл. *Model Driven Development - MDD*) је универзалнији приступ чија парадигма треба да буде примењена на у потпуности различите системе и агрегације система [145].

Основни постулат развоја вођеног моделом је постојање мета-модела. Мета-модел треба да садржи податке о конкретном ентитету који представља модел одређеног система. Могу се као пример узети неки имагинарни модели у којима су дизајниране компоненте и њихова својства попут животног циклуса и спрега које представљају зависности од других компонената. Те спреге могу додатно бити описане типовима података који се преносе од компоненте до компненте и врстом протокола који се користи. Ови ентитети који чине модел могу бити представљени кроз мета-модел који се састоји од класа које одговарају

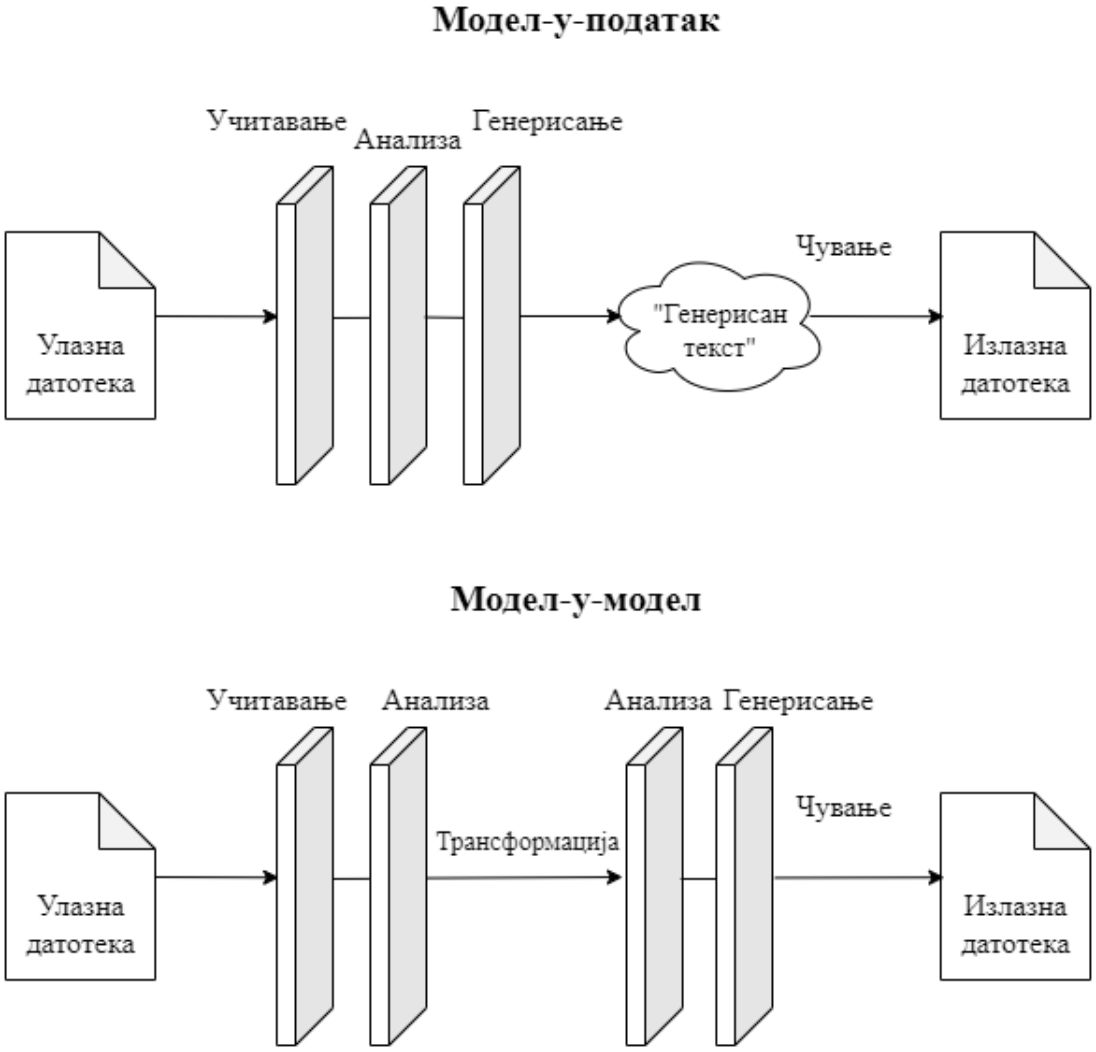


Слика 15: Пример учитавања модела

ентитетима из модела и кроз поља која описују својства и релације међу овим класама као што је приказано на слици 15.

Најпознатији оквир за развој, рад и коришћење мета-модела је *Eclipse Modelling Framework - EMF*. Он омогућава учитавање, интерпретацију и валидацију дизајнираног модела, односно података из модела у мета-модел као и обрнуто, чување података из мета-модела у конкретан модел неког система [146, 147]. Такође, рад са мета-моделима омогућава знатно једноставније превођење из једног језика у други. Први корак у превођењу је семантичко мапирање одлика два језика, а потом следи и имплементационо мапирање претходно дефинисаних релација кроз један или више мета-модела. Постоје две врсте превођења које је могуће оствари коришћењем мета-модела. То су 'модел у текст' и 'модел у модел' превођење као што је илустровано на слици 16.

'Модел у текст' [148] или како се још назива и 'модел у програмски код' [149] превођење представља процес којим се од функционалне целине одређеног језика у виду модела (било да је реч о моделу програма, моделу система и слично) путем мета-модела изравно настаје ток података (текст или програмски код) у неком другом циљаном језику. Овај процес подразумева првобитно учитавање модела језика који се преводи у мета-модел. Подаци из учитаног модела се екстрахују и интерпретирају и на основу анализираног семантичког мапирања једног језика у други ствара се модел програма или система који је представљен у другом језику. 'Модел у модел' превођење започиње на исти начин као и 'модел у текст', међутим, ток података у виду текста, односно програмског кода у другом језику не настаје изравно из учитаних података путем мета-модела. Након што су подаци интерпретирани, односно модел програма или ситема је учитан у мета-модел једног језика, врши се трансформација из учитаних класа података у класе података који одговарају мета-моделу другог језика на основу анализираног семантичког мапирања међу језицима [150, 151]. Потом се подаци представљени класама мета-модела другог језика чувају, односно настаје текст или програмски код у форми реалног модела програма или система приказаног другим језиком. Предност 'модел у текст' превођења је бржа имплементација



Слика 16: Превођења у развоју вођеном моделом

и погоднији је за једносмерне трансформације, док је 'модел у модел' превођење погоднije за двосмерне трансформације јер је ефикасније за развој и одржавање.

2.4.1 Модели језика за дефинисање спрега у програмској подршци аутомобилског система

За потпуно разумевање и корелацију међу језицима за дефинисање спрега који се најчешће примењују у програмској подршци *ADAS* и *IVI* домена (*ARXML, FIDL, AIDL*) треба разумети и семантику модела који се овим путем стварају. Кратак опис сваког од језика је дат у наставку, а у поглављу 6 за верификацију решења је дата и њихова граматика дефинисана у оквиру формалне верификације (граматика *ARXML* језика није дата у целости из разлога што су покривени само делови језика за опис комуникационих спрега, а не они делови којима се дефинише управљање компонентама и слично).

2.4.1.1 *AUTOSAR XML* Први, а истовремено и најсложенији језик за дефинисање сучења од наведених је *AUTOSAR XML - ARXML*. Иако је више од језика са дефинисање спреге јер служи за описивање компонената програмске подршке и њиховог понашања на нивоу читаве електронске контролне јединице, за потребе овог решења релевантан је само део *ARXML* модела којим се дефинишу комуникационе спреге међу компонентама. Свака компонента програмске подршке да би комуницирала са другим компонентама потребно је да има дефинисану спрегу за комуникацију. Свака спрега носи две информације, како преноси податке и шта су тачно подаци које преноси.

Како се врши пренос одређује који тачно од описаних комуникационих механизама се користе (догађај, метод и поље). Кроз једну спрегу може бити остварено више различитих комуникационих мезанизама, али тип података који се преноси мора бити јединствен и непроменљив. Подржани типови података су:

- примитивни типови (означени, неозначени, целобројни, децимални, логички)
- низови - низови било ког типа дефинисани стандардом *C++* програмског језика
- вектор - вектори било ког типа дефинисани стандардом *C++* програмског језика
- знаковни типови - дефинисан стандардом *C++* програмског језика, може бити представљен *UTF-8* или *UTF-16* форматом
- структура - сложен тип података који у себи садржи друге сложене или просте типове
- унија - дефинисана стандардом *C++* програмског језика
- набројиви тип - дефинисан стандардом *C++* програмског језика
- референца - указује на тип или другу референцу која се рекурзивно такође своди на тип

Спреге се у *ARXML* моделу дефинишу засебно а затим се укључују у опис компонената у виду инстанци тако описане комуникационе спреге. Да би инстанца комуникационе спреге била функционална, путем *ARXML* модела се дефинишу и параметри за дистрибуцију односно конкретну примену путем одговарајућег протокола. На примеру *SOME/IP* протокола то би на пример били идентификатори за методе, групе догађаја и догађаје, итд.

2.4.1.2 *Franca IDL - FIDL* Језик за дефинисање спрега на бази *Franca* радног оквира је по доступности конфигурационих параметара за подршку комуникације преко *SOME/IP* протокола сличан *FIDL* језику, с тим да је битна разлика у томе што *FIDL* не служи за опис понашања компонената у оквиру читаве електронске контролне јединице, већ само за опис спрега. Као и код *ARXML* језика, дефинишу се компоненте и њима се придружују инстанце дефинисаних комуникационих спрега. У овом случају, такође, спрегама се описује о ком комуникационом механизму је реч и који тип података учествује у преносу. Сви типови наведени у *ARXML* језику су подржани и од стране *FIDL* норматива. Међутим,

још једна од разлика је то што *FIDL* сам по себи не садржи конфигурационе параметре потребне за дистрибуцију, односно конкретну примену путем одговарајућег протокола као што је случај код *ARXML* језика. За ову сврху неопходно је користити *Franca Deployment - FDEPL* језик. *FDEPL* је продужетак претходно описаног језика за дефинисање спреге, такође базиран на *Franca* радном оквиру који служи да дефинише параметре примењиве искључиво за дистрибуцију конкретног протокола с обзиром на то да је поред *SOME/IP* протокола подржан и *D-Bus*.

2.4.1.3 *Android IDL - AIDL* С обзиром на то да *Android* језик за дефинисање спреге не подржава *SOME/IP* комуникациони протокол и механизме, већ *binder*, самим тим и моделовање спреге изгледа нешто другачије. На првом месту, не постоји концепт компонената програмске подршке којима би се придружила инстанца одређене дефинисане спреге, дефинише се искључиво комуникациони канал. Спрега за комуникацију је, такође, описана дефинисањем комуникационог механизма и типа податка који се преноси. Једини доступан комуникациони механизам је метода коју клијент позива да добије податке од сервиса и у којој смер аргумената које садржи може бити улазни (кроз њега клијент задаје параметар методи која се извршава на сервису), излазни (кроз њега добавља резултат натраг од сервиса) и улазно-излазни (може се користити за оба). Метода која нема повратну вредност и ниједан излазни или улазно-излазни аргумент назива се једносмерна метода (енгл. *Oneway*). Прости типови који су подржани овим језиком су целобројни означени (*byte, int, long*), децимални (*double, float*), логички (*bool*) и знаковни (*char* у *UTF-16* и *string* у *UTF-8* и *UTF-16* знаковној представи) типови, док неозначени типови нису подржани. Сложени типови који су подржани су низови, листе (које су представљене као вектори дефинисани стандардом *C++* програмског језика, односно листе у програмским језицима *Java* и *Kotlin*), мапе са паровима кључ-вредност и набројиви типови. Замена за структуре и уније су такозвани *Parcelable* ентитети специфично имплементирани за дефинисање сложених типова који могу да садрже било које друге типове у *Android* оперативном систему.

2.5 Безбедносни и сигурносни аспекти

Питања безбедности и сигурности у возилима као превозним средствима високих захтева су изузтено битна, стога захтевају посебну анализу. С почетка, аутомобилски систем је био енкапсулиран систем на ког нису утицали други уређаји и ентитети па је фокус у развоју био искључиво на перформансама, цени и поузданости у виду детерминистичког, предодређеног понашања. Већ доста пута поменуто детерминистичко понашање је управо оно што чини већи део безбедносног аспекта. Да би се обезбедило безбедно решење, читав систем треба да буде подвргнут анализи и променама пратећи постулате из два главна стандарда [152,153]. У зависности од тога који је циљни *safety integrity level* потребно је верификовати разне ствари, од принципа управљања пројектом [154] и анализе штете, алата у којима се врши имплементација (чак и алате за уређивање текста), преко способности оперативних система да испуне захтеве у реалном времену, до контролисане интерференције компонената међусобно. Пример је употреба *Microsar* оперативног система који испу-

Табела 2: Напади који су угрозили сигурносне механизме и безбедност аутомобила

Нападач и година	**Нападнут произвођач**	**Циљана компонента и врста напада**	**Исход**
Аутори рада [158] 2016	Honda и Hyundai	Напади блокирања услуга (CAN магистрала)	Отказивање магистрале
Аутори рада [159] 2015	MobilEye C2-270 и LiDAR ibeo LUX 3	Лажирање идентитета и оптерећивање протока (камере и лидари)	Отказивање компонената за перцепцију околине
Keen Security Lab 2016 [160]	Tesla Model S	Удаљени приступ (Wi-Fi)	Потпуна контрола возила
Amat Cama и Richard Zhu, 2019 [161]	Tesla Model 3	Приказ претраживача на инфо-забавном систему	Измена кода фирмвера
William Hatzer и Arjun Kumar, 2017 [162]	Hyundai	Удаљени приступ (Wi-Fi)	Угрожени приватни подаци и стављање возила у погон
Charlie Miller и Chris Valasek, 2015 [163]	Jeep	Удаљени приступ UConnect инфо-забавном систему	Изложеност управљачког и кочионог система
Charlie Miller i Chris Valasek, 2013 [163]	Ford и Toyota	OBD	Контрола читавог возила

њава све захтеве класичне *AUTOSAR* платформе направљене између осталог да омогући остваривања различитих нивоа безбедности, док *Android* не може да има *safety integrity level* јер не може да пружи гаранцију за извршавање задатака у реалном времену и унапред дефинисаном редоследу [155]. Стога се не може употребити у *ADAS* домену, већ одговара потребама инфо-забавног домена који не захтева једнаку стриктност и чији откази не могу довести до тако штетних последица. Исти принцип се примењује и на компоненте физичке архитектуре, потребно је имати посебне микроконтролере за омогућавање највиших нивоа безбедности [156] путем детерминизма и поузданости у извршавању, док је за остале компоненте програмске подршке неопходно омогућити диригован распоред, предодређено време извршавања и спреге за комуникацију са другим компонентама [1]. Са друге стране, традиционални приступ енкапсулације аутомобилског система је довео до изостанка имплементације сигурносних механизама у коришћеним комуникационим протоколима, све док, услед отежане интеграције, није дошло до екстензивног развоја комуникације унутар и ван возила. Овиме је аутомобилски систем постао изложен нападима са других уређаја [157]. Примери напада који су демонстрирали рањивост аутомобилског система у протеклим годинама су приказани у табели 2.

Иако грешке или неовлашћени упади у инфо-забавни домен сам по себи не могу изазвати смртне последице, његова комуникација са остатком система, у овом случају *ADAS* доменом и потенцијална интрузија кроз корисничке апликације инфо-забавног домена је оно што може довести и до таквих сценарија. Из тог разлога, треба обезбедити поменуту слободу од нежељеног уплитања (енгл. *Freedom from interference*) без обрзира на то да ли се домени налазе на истој физичкој архитектури [164] или не. За омогућавање овога, на првом место је потребно обезбедити сигурносне механизме у оквиру најкоришћенијих комуникационих канала [165]. Постоје радови [166] и неки у којима је приказано сигурносно решење за конкретну примену *SOME/IP* протокола [167, 168] користећи сертификате

имплементиране од стране произвођача у сам уређај и асиметрично шифровање за размену кључа који служи за шифровање конкретних информација које су за слање. Мана овог приступа је то што подразумева реимплементацију постојећих решења *SOME/IP* протокола на нижим нивоима, а не на нивоу програмске спреге, смањује прилагодљивост и динамику и не представља решење за дистрибуиране нападе ради блокирања услуга (енгл. *Distributed Denial of Service - DDoS*).

2.6 Постојећа решења за доменску међуповезаност и генеративно програмирање

Постоји много радова који препознају потребу за остваривањем међудоменске комуникације између система који пружа напредну помоћ возачу и инфо-забавног домена. Неки истраживачи [169] то анализирају кроз потребу за повезивањем инфо-забавног домена путем комуникационих магистрала и спрега карактеристичних за домен за напредну помоћ возачу као што је поменута *CAN* магистрала. Тај приступ поседује бенефите у брзој и поузданој размени малих контролних података, међутим овиме је ограничена количина података, а самим тим и контекст коју могу да размењују домени, односно којима могу да располажу корисничке апликације у инфо-забавном домену. Додатно, подразумева читав развојни циклус јер је потребно обезбедити подршку физичке архитектуре, а затим и подршку кроз све нивое стека програмске подршке с обзиром на то да *CAN* магистрала није примарна за инфо-забавни домен, а често није ни подржана по основи. Из тог разлога, тренутна путања развоја магистрала за комуникацију описана у секцији 3.3 за овакву улогу практично предлаже *Ethernet* на првом месту због протока и доступности. То је, такође, препознато од стране аутомобилске заједнице па постоји решење које омогућава конверзију [170] имеђу *CAN/CAN-FD* и *SOME/IP* протокола који је првобитно представљен као носилац међуплатформске комуникације између апликација у класичној и адаптивној платформи [171]. Са друге стране, неки су покушали да приступе проблему поседовања информација јединствених за *ADAS* домен у инфо-забавном домену пресликавањем имплементације како специфичних тако и генералних алгоритама. Тако на пример радови [172, 173] имплементирају надгледање возача диркетно у аутомобилским, али и класичним корисничким инфо-забавним системима попут мобилних уређаја, док са друге стране аутори у [174] решавају изазов имплементације и извршавања алгоритама детектовања, праћења објеката и слично. Овакви приступи су редундантни и представљају велико оптерећење у виду заузећа ресурса и времена извршавања за инфо-забавни домен који иницијално није предвиђен за такве фунцкионалности. Рад [175] сведочи о напорима да се чак читав стек програмске подршке адаптивне платформе *AUTOSAR* стандарда прилагоди захтевима инфо-забавног домена са фокусом на стварању подршке за графички модул и корисничка сучеља и да се на тај начин униформише програмску подршку у аутомобилском систему.

Поред покушаја усвајања комуникационих магистрала и имплементирања компонената програмске подршке, постоје и решења која за циљ имају да омогуће доступност удаљених ресурса физичке архитектуре [176, 177] који припадају домену за пружање на-

предне помоћи возачу. Примарни дељени ресурси су сензори, тачније камере с обзиром на то да се у аутомобилској индустрији не користе камере које се повезују путем универзалне серијске магистрале (енгл. *Universal Serial Bus - USB*) као што је типичан случај у потрошачкој електроници, већ посебним спрежним пролазима попут гигабитне мултимедијалне серијске везе (енгл. *Gigabit Multimedia Serial Link - GMSL*). Међутим, иако идеја употребе удаљене камере решава непотребну дупликацију истоветних ресурса, при реализацији комуникације неопходно је узети у обзир контекст података који се шаљу и на основу тога потенцијално искористити различите комуникационе протоколе и механизме који су погодни за конректну примену. Стога, остваривање великог тока података као што је случај за податке који долазе са камере није најбоље извршавати путем *SOME/IP* [176] или основног *Ethernet* [177] протокола, на првом месту због непостојања механизама за поуздано усклађивање и руковање сегментима јединичних оквира података са камере који долазе путем вишебројних узастопних порука и које је потребно правилно уклопити у једну целину. *Google* компанија одговорна за развој и одржавање *Android* оперативног система такође увиђа потребу за проширивањем слоја за апстракцију ресурса физичке архитектуре специфичним модулима за возила попут камера за окружење возила, осветљења и слично [178], али и покушајима за стварање програмске спреге која пружа акцелерацију извршавања модела неуронских мрежа [179] која представља лаку верзију онога што већ постоји у типичним *ADAS* системима.

Хетерогени састав аутомобилског система је изазов који је у развоју прогрмске подршке тешко адресирати и представља главну препреку за брзу имплементацију и интеграцију иновација. Рад [180] презентује платформу за програмску подршку која нуди решење за прилагодљив развој хетерогених система узимајући примарно у обзир захтеве домена за напредну подршку возачу, али и захтеве инфо-забавног домена, телематике и других електронских контролних јединица. Представљена је абстракција над различитим јединицама физичке архитектуре и оперативним системима, омогућавајући најмању могућу алгоритамску зависност од истих. Међутим, абстракција међупроцесне и међупроцесорске размене података, као и контекст коришћења различитих комуникационих канала нису били у фокусу, поготово када је реч о употреби механизама који су специфично намењени за одређени систем као што је ситуација за *binder* и *Android*.

У овако поменутим хетерогеним системима, средњи слој програмске подршке игра велику улогу у одвајању апликативног од нивоа системских сервиса и коришћења ресурса физичке архитектуре. Постоје бројни радови који предлажу различите начине имплементације програмских платформи и средњих слојева који пружају повољно окружење за функционалности модерних возила (превасходно тежећи ка омогућавању извршавања задатака напредне помоћи возачу и аутономне вожње у реалном времену) [181–184]. Међутим, овај механизам решавања проблема хетерогености стварањем нових платформи за програмску подршку није довољан. Потребно је покрити и аспект њихове међусобне интеграције имајући у виду да јединствен средњи слој који одговара захтевима свих домена тренутно још увек не постоји. Компанија *CARIAD* имплементирањем своје *VW.OS* платформе [185] покушава да оствари једно овакво решења које би интегрисало популарне постојеће системе и модуле средњих слојева [186,187]. Са друге стране, произвођач аутомо-

била *Renault* у сарадњи са *Google* компанијом покушава да створи програмску платформу за возила нове генерације [188] која ће објединити све електронске контролне јединице у систему, њихову међусобну комуникацију и једноставно вршење удаљеног ажурирања, али и додатно омогућити размену података са облаком. Најављени су чак и помаци у физичкој архитектури од стране *Qualcomm* компаније као једног од главних произвођача чипова за аутомобилску индустрију (поготово када је реч о инфо-забавним системима). *Qualcomm* планира да у 2024. години објави чип који ће објединити инфо-забавне и функције домена за напредну помоћ возачу [189].

Бивши *GENIVI*, а данашња *COVESA* у раду са истраживачким центрима компанија *Itemis* и *ITK engineering* представили су интеграцију *ara::com* и *CommonAPI* модуле средњег слоја [190] за које постоји и аутоматизовано решење названо *FARACON* [191]. Наведеним решењима компаније *COVESA* је допринео и истраживачки центар аутомобилске куће *Renault* који је дефинисао свој развојни пут и додатне изазове и бенефите у преласку на сервисно-оријентисану архитектуру као одговор на хетерогеност система [192]. Овиме указују на могућност усредсређивања на дефинисање самог посла који извршава нека компонента и начина којима је она спрегнута са остатком система уместо дефинисања података фокусирајући се на формат самих података на комуникационом медијуму. Недостаци у овим решењима су представљени у непоптпуном мапирању постојећих типова података и додатних компонената за само извршавање, као и у недостатку испитивања каснијег пружања података корисничким апликацијама. Поред овога, рад [192] је указао и на недостатке у виду сигурности и безбедности који у наведеним решењима нису, а треба да буду додатно испитани јер у покушају остваривања веће динамичности и флексибилности, поузданост чији је основни извор била статичност и предодређеност система изостаје. Још једна од уочених предности коју је могуће остварити применом сервисно-оријентисане архитектуре је интензивнија употреба језика за дефинисање спреге и могућност њиховог семантичког мапирања као првог корака у остваривању компатибилности и коначно интеграције различитих система. У раду [193] је сарадњом компаније *Continental* са универзитетима Регензбурга и Западне Бохемије спроведена анализа мапирања, односно превођења језика за дефинисање спреге у сервисно-оријентисаним архитектурама која узима у обзир и *Android*. Међутим, ово је само први корак у интеграцији, након тога је неопходно одредити конкретну примену испитаних делова који настају оваквим превођењем, поготово на случају примене *Android* система који принципе сервисно-оријентисане архитектуре подржава на нешто другачији начин него што је типично предвиђено *AUTOSAR* платформом.

3 Одабир погодног комуникационог протокола

У овом поглављу отпочиње опис решења представљеног дисертацијом. Први задатак је селекција технологије погодне за улогу остваривања међудоменске комуникације између *ADAS* и *IVI* система. При одабирању комуникационе технологије потребно је најпре извршити одабир комуникационог канала и механизама за остваривање комуникације, а са тим и дефинисање коришћеног протокола. При вршењу ових одабира треба пажњу усмерити првенствено на контекст података који се шаљу, тј. њихову улогу и порекло компонената са којих долазе, као и доступност физичких медијума у доменима чију међусобну комуникацију омогућавамо.

3.1 Одређивање комуникационог канала подобног за међудоменску комуникацију

Приликом одабира технологије преноса података потребно је узети у обзир следеће факторе: физичке карактеристике система, захтевани проток, поузданост и контекст података који се преносе. Све разматране технологије преноса су описане претходно у потпоглављу 2.3.

Главна предност дељене меморије је њена брзина, која долази од непосредног приступа подацима. Иако овом карактеристиком значајно предњачи у односу на друге начине размене података, примена дељене меморије је могућа пре свега у случају када се два процеса која комуницирају налазе на истом језгру. Њена примена је могућа и у случајевима када је систем на чипу такав да је приступ истим меморијским локацијама дозвољен за различита језгра на којима се процеси извршавају. У случају када се процеси налазе на физички различитим уређајима, јасно је да овај канал за комуникацију није могућ, а то је управо случај који је тренутно присутан у аутомобилској индустрији. Из овог разлога се дељена меморија не користи за учитавање података са сензора и слање података на актуаторске јединице, што је такође проблем чије решавање предложени приступ треба да

омогући разменом података између компонената програмске подршке као и компонената физичке архитектуре.

Од традиционалних магистрала у аутомобилима, по питању контекста података који би се преносили нема ограничења. Што се тиче физичке доступности, ограничења не постоје, међутим, захтевају додатни развој инфо-забавних уређаја тако да прихватају рад са овим магистралама. *LIN* магистрала је, како је већ речено, осмишљена да буде јефтина магистрала, па из тог разлога не нуди значајне перформансе које би биле погодне за употребу у инфо-забавним решењима, па је то додатни разлог за њено одбацивање. *CAN* и *FlexRay*, са друге стране, представљају изузетно поуздане магистрале у смислу гаранције да ће податак бити достављен. Међутим, проток је оно што представља препреку, јер не дозвољава широку употребу канала, уколико би канал био коришћен од пуно различитих компонената програмске подршке. Употреба напредних сензора који шаљу велике токове података би такође била онемогућена. Остале наведене серијске магистрале имају једноставно другачију улогу, намењене су за емитовање кратких сигнала којима се конфигуришу периферијски уређаји и не могу бити коришћене као главна комуникациона осовина између *ADAS* и *IVI* домена. *PCIe* је комуникациона магистрала која предњачи по питању протока који је омогућен њеним коришћењем и из генерације у генерацију поспешује механизме за поузданост. Међутим, ограничена је дужином од око пола метра коју може да досегне њен физички медијум, што може представљати проблем у спајању домена у самом возилу.

Имајући у виду претходно наведене факторе, *Ethernet* представља погодан комуникациони медијум за пренос података свих контекста, било да је реч о подацима које размењују компоненте програмске подршке или подацима са сензора и других јединица физичке архитектуре. Са сталним порастом протока, *Ethernet* покушава да испрати потребе свих случајева употребе који се такође развијају, постављајући сваки пут све веће захтеве. Модерни сензори, попут лидара, радара и камера, генеришу изузетно велике токове података као што је предочено у потпоглављу 2.1.2. Ово непосредно утиче и на количину података која се размењује између компонената програмске подршке, чији се број такође повећава. Доступност је још једна повољност за *Ethernet*, из разлога што је већ подржан у оба домена и највећи број инжењера располаже добрим познавањем и вештином у његовој употреби, што је такође битан аспект овог решења које покушава да допринесе широкој заједници. Међутим, није довољно само одредити *Ethernet* као комуникациони медијум, већ и стратегију на који начин ће комуникациона архитектура бити формирана. У овом раду неће бити узимани у обзир елементи физичке архитектуре, у виду постојања и топологије скретница, тј. уређаја за дистрибуирање података (енгл. *Ethernet switches*), већ је фокус на дизајну на нивоу програмске подршке. Стога, сервисно-оријентисана архитектура је одабран приступ како би дистрибуција решења и његова модуларност добили потпун смисао. Следећи корак је одређивање погодних комуникационих механизама.

Табела 3: Поређење протокола за имплементацију сервисно-оријентисане архитектуре

	SOME/IP	**DDS**	**MQTT**	**HTTP**	**WebSockets**
Транспортни протокол	UDP/TCP	UDP/TCP	TCP	TCP	TCP
Механизам комуникације	Претплата и одговор на захтев	Претплата	Претплата	Одговор на захтев	Претплата и одговор на захтев
RPC подршка	Уграђена подршка	Два пара пошиљаоца и претплатника	Два пара пошиљаоца и претплатника	Уграђена подршка	Уграђена подршка
Шифровање	Није подржано	SPI	TLS	TLS	TLS
Успостава везе	Без / Са успоставом везе	Без / Са успоставом везе	Са успоставом везе	Без успоставе везе (REST)	Са успоставом везе
AUTOSAR подршка	Класична и адаптивна платформа	Делимично адаптивна платформа	Није подржан	Делимично адаптивна платформа	Није подржан
Имплементација за Android	Доступна	Доступна	Доступна	Доступна	Доступна
Механизам откривања сервиса	Уграђена подршка	Уграђена подршка	Није подржан	Није подржан	Није подржан
QoS	Није подржан	Уграђена подршка	Уграђена подршка	Није подржан	Није подржан

3.2 Одређивање протокола и механизма за омогућавање сервисно-оријентисане парадигме

Начини на које је могуће добавити информацију коју пружа одређени сервис су разнолики. *SOME/IP* протокол је својим стандардом дефинисао употребу три механизма (метод, догађај и поље) којима је омогућен синхрони упит, али и асинхрона комуникација. Механизми су претходно описани у потпоглављу 2.3. Ово је први од фактора који га кандидују за погодно решење поред тога што је специфично осмишљен за употребу у аутомобилској области и што је првобитно и створен за потребе међуплатформске комуникације, односно повезивања две *AUTOSAR* платформе (класичне и адаптивне).

Сумирано поређење *SOME/IP* са другим распрострањеним протоколима који би се могли применити у овој улози дато је у табели. Иако су сви други протоколи далеко распрострањенији у сервисно-оријентисаним архитектурама, механизми комуникације које они подржавају као и начин рада их чине мање погодним од *SOME/IP* протокола. По питању комуникационих механизама једино *WebSockets* подржава оба типа комуникације, тј. и претплаћивање на податак и одговор на захтев. *Data Distribution Service - DDS* и *Message Queuing Telemetry Transport - MQTT* врше комуникацију на бази претплаћивања у мрежи равноправних корисника (енгл. *Peer-to-peer*) или преко посредника/агента (енгл. *Broker*) респективно, док у случају употребе *Hypertext Transfer Protocol - HTTP* протокола подаци могу бити добављени само на захтев. Наравно, у случају претплатничког приступа могуће је имплементирати позивање удаљене методе, тј. добављање одговора на захтев путем два пара претплатника и корисника који објављује садржај, док је најближи начин

за опонашање претплатничког приступа путем одговора на захтев учесталих провера стања. Поред тога, *MQTT, HTTP, WebSockets* почивају искључиво на *Transmission Control Protocol - TCP*, док *SOME/IP* и *DDS* могу бити имплементирани и над *User Datagram Protocol - UDP* протоколом који не остварује везу и није поуздан, али омогућава бржу трансмисију. Такође, једино је имплементација SOME/IP протокола у потпуности интегрисана од стране стандардизованих платформи као што је *AUTOSAR*. Подршка за *DDS* је у протеклим годинама омогућена искључиво од стране адаптивне платформе, али не у потпуности, и интеграција је и даље у току. Исто важи и за *HTTP* чија је употреба на помолу кроз *ara::rest* модул, док за *MQTT* нема информација о расположивој подршци. Ово би додатно продужило развој решења јер би на првом месту захтевало имплементацију самог протокола и његову интеграцију у постојеће платформе. Са друге стране, једино *DDS* и *MQTT* имају уграђене механизме за шифровање и осигуравање квалитета услуге. *HTTP* и *WebSocket* имају верзије са уграђеним шифровањем, али без осигуравања квалитета услуге, док *SOME/IP* нема никакве уграђене механизме који би осигурали квалитет или заштиту података.

Из свих приказаних разлога, *DDS* је издвојен као могућ супарник *SOME/IP* протокола, па је за њега урађена додатна процена. *DDS* својим стандардом пружа већу конфигурабилност и контролабилност, али захтева и сложенију програмску спрегу. Најзначајнија ставка коју нуди *DDS* применом *Real Time Publisher Subscriber - RTPS* протокола огледа се у двадесет и три профила за квалитет услуге. *DDS* пружа и већу модуларност и семантичку поделу међу учесницима користећи моделе учесника у комуникационом домену (енгл. *Domain*) на највишем нивоу, а затим учеснику придружене претплатнике и пошиљаоце у оквиру истог домена на средњем нивоу, да би на најнижем нивоу били ентитети за читање и писање података који су придружени одговарајућем претплатнику или пошиљаоцу и везани су за јединствену тему (енгл. *Topic*). Анализирајући постојећа решења може се уочити да доступне имплементације, подржавају и различите програмске спреге и платформе (могућност типичне имплементације стека у *C++* програмском језику за платформе *Linux* дистрибуције, али и *Java* спреге за *Android* оперативни систем додате изнад поменуте *C++* спреге). Међутим, избор корисничке спреге у овом решењу не игра одлучујућу улогу, из разлога што ће она бити реализована тако да се прилагоди моделу сервиса који примљеним подацима снабдева апликације у инфо-забавном домену, па је коришћење чистог имплементационог стека стандарда најефикаснија опција за избегавање непотребних вишеструких слојева и омотача око функција од значаја. Имплементирани су и једноставни примери и у табели 4 су дата времена извршавања истоветних функционалности применом ова два протокола. Одабране функционалности за упоредбу два протокола су добављање података на претплату (описани механизам догађаја) и добављање података кроз одговоре на захтев (метод и поље за добављање). Механизми поља за постављање и обавештавање су слични догађају где је очекиван сличан исход за оба протокола па ти примери нису имплементирани. Са друге стране, за добављање података на захтев примери за оба механизма су имплементирани јер је ту очекивана највећа разлика имајући у виду да *DDS* нема уграђену подршку за ове механизме. Примери нису оптимизовани (тј. не представљају крајње решење већ служе само у сврху поређења и процене

Табела 4: Поређење кашњења на примерима *SOME/IP* и *DDS* протокола

	SOME/IP[ms]	DDS[ms]
Догађај (податак на претплату)	7.54	9.39
Метод (податак на захтев)	46.71	65.32
Поље за добављање (податак на захтев)	43.2	128.54

понашања ова два протокола). Јасно се може видети да је у сврху коришћења механизма за добављање података на захтев *DDS* значајно спорији. Овиме је коначно закључено да ће *SOME/IP* протокол бити употребљен за остваривање међудоменске комуникације између инфо-забавног домена и домена за напредну помоћ возачу. Међутим, треба имати у виду да ни један протокол није имплементиран тако да савршено одговара свим случајевима употребе, а да циљано решење ове тезе обухвата размену разноликих података. Сходно томе, постоји неколико главних ограничења у овом приступу које смо покушали да савладамо.

Први проблем практичне примене *SOME/IP* протокола је тај што инжењери, посебно они који раде на развоју програмске подршке у инфо-забавном домену, нису обучени и упознати са *SOME/IP* стандардом. Други проблем је поменути недостатак механизама за сигурност, а трећи проблем је што протокол као такав не садржи механизме којима се могу преносити велики токови података чији су редослед и времена пристизања битни, као што је случај за репродукцију аудио или видео сигнала.

Први проблем је превазиђен тиме што комуникација путем овог протокола није корисничким компонентама омогућена искључиво путем програмске спреге специфичне за *SOME/IP* протокол. У *ADAS* домену комуникација је замаскирана кроз позиве модула средњег слоја (*ara::com*) као што ће бити приказано у потпоглављу 4.1.1. На страни *IVI* домена корисничке компоненте могу да користе решењем дату спрегу која је специфична за протокол, док је у случају *Android* оперативног система *SOME/IP* комуникација интегрисана у његов доменски механизам за међупроцесну комуникацију - *binder* [194] што је приказано у потпоглављу 4.1.2. Како би употреба *Android binder* механизма била омогућена било је неопходно превазилажење различитости у парадигмама комуникације описане *AIDL* сервисима и *SOME/IP* протоколом. Подршка за случај *Android* оперативног система је овим решењем у потпуности аутоматизована. Компоненте које добављају податке преко *SOME/IP* протокола су такође генерисане тако да могу лако бити интегрисане и у различите механизме за међупроцесно прослеђивање података до корисника попут дељене меморије (енгл. *Shared memory*), редова порука (енгл. *Message queues*), канала за податке (енгл. *Pipes*) за случај *Linux* дистрибуција, или на пример специфичних редова порука у случају QNX оперативних система. Међутим, због велике разноликости у механизмима ови случајеви нису аутоматизовани, већ је процес генерисања заустављен на стварању комуникационих компонената које садрже једноставну спрегу за размену података и које је потребно накнадно у форми библиотека повезати са извршном датотеком самог сервиса

који прослеђује податке до корисничких апликација на жељени начин. Други проблем је превазиђен тако што су механизми који обезбеђују сва три нивоа сигурности (поверљивост, интегритет и аутентичност) такође имплементирани овим решењем. Ова имплементација је дата у потпоглављу 4.1.3. За трећи проблем је у потпоглављу 4.2 приказана имплементација посебне програмске спреге за канал који служи за размену великих података. Спрега је једноставна за коришћење и кроз свега неколико позива, клијентска и серверска страна могу да размењују велику количину података путем *UDP, TCP, RTP* и *RTSP* протокола без потребе за њиховом имплементацијом.

4 Реализација комуникације применом сервисно-оријентисане парадигме

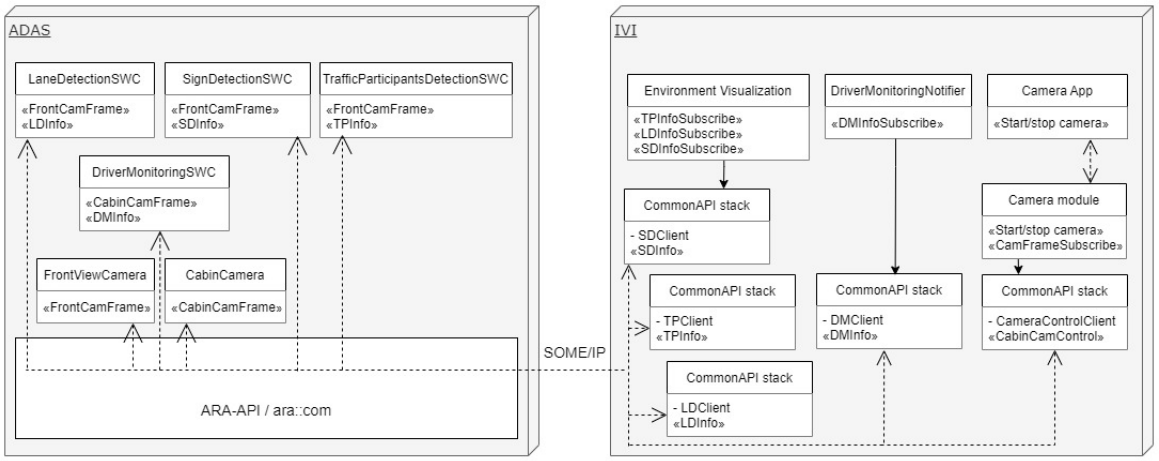
У овом поглављу биће представљена читава комуникација имплементирана овим решењем. Решење је подељено у два дела где оба користе парадигме сервисно-оријентсане архитектуре. Први представља комуникацију путем *SOME/IP* протокола. Други представља засебну програмску спрегу која служи као допуна на постојећу *SOME/IP* комуникацију са циљем да надомести његов недостатак омогућавањем размене великих токова података.

4.1 Реализација комуникације путем *SOME/IP* протокола

У овом делу рада биће описана реализација имплементираног решења за остваривање комуникације између домена за напредну подршку возачу и инфо-забавног домена путем *SOME/IP* протокола. Опис оствареног решења биће подељен по томе како је текла реализација у сваком од два домена појединачно. Сама аутоматизација није покривена овим поглављем, већ ће бити презентована у наредном поглављу засебно.

4.1.1 Имплементација на страни домена за напредну подршку возачу

Како је већ речено, у домену за напредну подршку возачу апликације се извршавају на врху стека програмске платформе. Модул средњег слоја (*ara::com*) по стандарду подржава *SOME/IP* протокол, па је из тог разлога имплементација сервиса на овој страни прилично једноставна и прати стандардизовани приступ. Један од циљева овог решења је да минимално утиче на постојећу програмску архитектуру, представља минималан терет по питању перформанси и захтеваних ресурса и да буде конфигурабилно, односно да подржи више приступа које је могуће одабрати и пружи јасно објашњење који од њих је боље користити у одговарајућем случају.



Слика 17: Дистрибуирани приступ имплементираног решења

Приликом развоја решења, разматране су два могућа архитектурална приступа: дистрибуирани и централизовани. Разлика у њиховој функционалности претежно произилази из разлике у компонентама које се налазе у домену за напредну помоћ возачу. Код дистрибуираног решења, пренос корисних података одвија се изравно са компонената програмске подршке *ADAS* домена ка *IVI* домену, као што је приказано на слици 17. Са друге стране, слика 18 визуализује централизовани приступ, који подразумева постојање посредничке програмске компоненте у *ADAS* домену чија ја једина улога да преузме жељене податке из остатка домена и проследи их на *IVI* страну.

4.1.1.1 Дистрибуирани приступ Непосредна комуникација програмских компонената од интереса са инфо-забаним доменом остварена је тако што је у самим компонентама потребно извршити позиве за слање догађаја, обавештавајућих или поља за постављање вредности, односно имплементирати функције које ће бити прозване са стране инфо-забаног домена у случају добављања података на захтев помоћу метода или поља за добављање. Овде је важно напоменути да компоненте *ADAS* домена у свим случајевима користе објекте сервисне стране модула средњег слоја, сем у случају постављања вредности поља. У случају постављања вредности поља, компоненте на *IVI* страни се понашају као *SOME/IP* сервиси, док *ADAS* страна у улози клијента шаље захтеве за постављање вредности и на тај начин преноси корисну информацију.

Дакле, у случајевима када са *ADAS* компонената подаци одлазе ка *IVI* страни у виду догађаја, потребно је позвати методу *EventsSkeleton* класе:

- *void OfferService()* - неопходна да би сервис био видљив, односно да бо претплата клијената могла бити успешну. Уколико је потребно прекинути видљивост, тј. изложеност сервиса за то је задужен позив *void StopOfferService()* методе

Након што је сервис изложен потребно је позвати следеће методе шаблонске класе *EventDispatcher* наследнице *DispatcherBase* класе:

- *ara::core::Result<ara::com::SampleAllocateePtr<T>> Allocate()* - заузима меморију и враћа показивач на део меморије у ком ће моћи да буду уписани подаци за отпремање

- *ara::core::Result<void> Send(ara::com::SampleAllocateePtr<T> data)* - врши слање корисног податка ка свим претплаћеним клијентима

Уколико је обавештавајуће поље жељени механизам за размену података, није потребно позивати методе за слање, већ је довољно само изменити вредност поља методом:

- *ara::core::Result<void> Update (const_reference data)* - мења вредност поља над којим је позвана и даље позива слање обавештења кроз средњи слој ка клијентима

Све претходно наведеневе методе се не прозивају над објектима наведених класа, већ кроз *Skeleton* компоненту укључивањем генерисаних датотека које нуде програмску спрегу ка средњем слоју.

Са друге стране, уколико је потребно одговорити на захтеве за добављање података са *IVI* стране, потребно је имплементирати методе које ће бити прозване са удаљених клијената:

- *ara::core::Future<Skeleton> METHOD_NAME (data) override* - метода која имплементира наслеђену методу компоненте *Skeleton*. У оквиру ове методе је потребно сместити жељени податак за размену у *std::future* и *std::promise* механизма *C++* програмског језика чиме је обезбеђено асинхроно добављање података

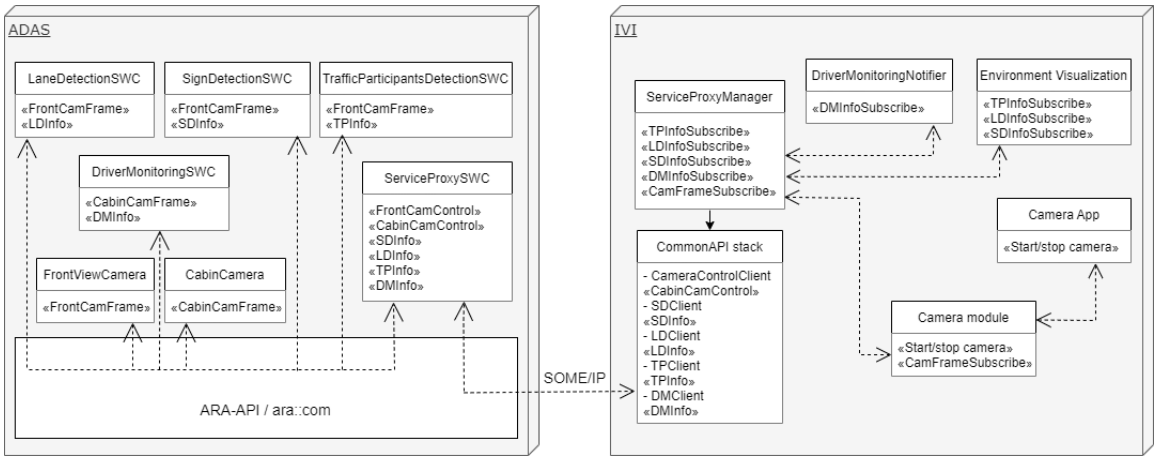
- *ara::core::Result<void> RegisterGetHandler(std::function<ara::core::Future<FieldType>()> getHandler)* - позивом ове методе над пољем компоненте *Skeleton* региструје се функција *getHandler* која ће бити прозвана на захтев удаљеног клијента за добављање вредности поља и коју је потребно имплементирати на исти начин као што је претходно објашњено за механизам методе

У случају коришћења поља за постављање вредности, користе се нешто другачије методе и рад над *Proxy* компонентом:

- *ara::core::Result<ara::com::FindServiceHandle> StartFindService(ara::com::FindServiceHandler<Proxy::HandleType> handler, ara::com::InstanceIdentifier instanceId)* - метода помоћу које се врши откривање сервиса, једини механизам којим клијент може открити постојање сервиса. Није неопходна за коришћење поља за постављање вредности

- *ara::core::Future<FieldType> Set(const FieldType& value)* - метода коју је потребно позвати како би предложена вредност поља била постављена. Повратна вредност је потврда да је вредност постављена

Предност дистрибуираног приступа је очигледно брзина јер је комуникација непосредна. Са друге стране, имајући у виду да комуникациони канал који спаја ова два домена треба да садржи напредније сигурносне заштите које омогућавају све три одлике (поверљивост, интегритет и аутентичност), дистрибуирани режим би захтевао интеграцију ових механизама унутар самих *ADAS* компонената. Такође, уколико је случај коришћења такав да клијентска апликација из *IVI* домена одређену информацију треба да добије на захтев,



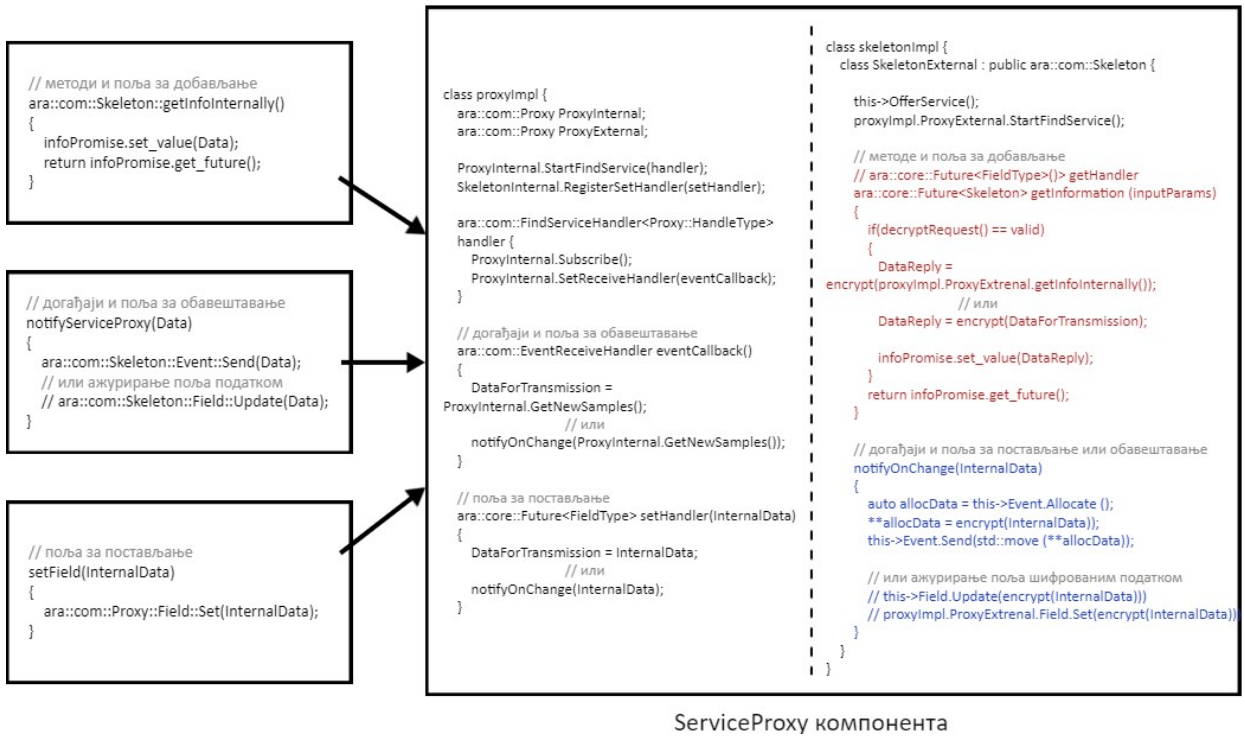
Слика 18: Централизовани приступ имплементираног решења

неопходно би било имплементирати функционалност методе која одговара на позвани захтев и враћа одговарајући податак. С обзиром на то да су ове компоненте развијане од стране различитих компанија и истраживачких центара и да се њихово извршавање мери у микро или милисекундама поменути механизми би додатно оптеретили компоненту што може бити непожељан сценарио.

4.1.1.2 Централизовани приступ Централизовани притуп, за разлику од дистрибуираног, поседује јединствену посредничку компоненту која интерно комуницира са свим компонентама чији су подаци од интереса за пренос и ове податке прослеђује ка *IVI* домену. Назив компоненте која представља овакву спону је *'ServiceProxy'*.

Дакле, централизовани приступ би био начин да наведени проблеми дистрибуираног приступа буду решени на рачун увођења нешто већег кашњења у решење. Централизовани приступ омогућава реализацију сигурносних механизама у посредничкој комуникацији која би представљала филтер потенцијалне малициозне комуникације са *IVI* доменом чиме програмске компоненте које имплементирају алгоритме за напредну подршку возачу не би биле непосредно угрожене. Поред тога, посредничка компонента омогућава мапирање различитих механизама, односно даје могућност пријема информација од стране *ADAS* компонената у виду догађаја као најмање захтевних за пошиљаоца и тако релаксира њихово извршавање, док прослеђивање може бити омогућено ка *IVI* страни кроз механизам методе или поља за добављање података на захтев. Принцип мапирања *SOME/IP* механизама који је имплементиран у централизованом приступу је дат на слици 19, а поређење ова два приступа по питању времена извршавања и оптерећења на процесору је дато у поглављу 6 за верификацију решења.

Принцип мапирања који се може видети на слици 19 има једно ограничење - не подржава мапирање између унутрашњих метода или поља за добављање вредности на догађаје, поља за обавептавање или поља за постављање вредности ка *IVI* домену. Разлог је то што ова компонента представља искључиво спону која извршава задатке реагујући на побуду без управљачке моћи, односно логике за одлучивање када ће информација бити пренета. Фактори одлучивања када ће догађаји, обавештавајућа поља или поља за постављање вредности бити позвани могу бити било шта (граничне вредности, временски циклуси,



Слика 19: Мапирање механизама у посредничкој компоненте

насумични догађаји, итд) и из тог разлога су немогући за потпуну аутоматизацију, а могу бити додати ручно након генерисања.

Дакле, спољашњи догађаји, обавештавајућа поља и поља за постављање вредности могу бити мапирани искључиво на унутрашње догађаје, обавештавајућа поља и поља за постављање вредности у било ком пару тако што унутрашњи меганизам побуђује извршавање спољашњег. У наставку је дат пример који врши мапирање унутрашњег обавештавајућег поља и спољашњег догађаја. Када дође до промене вредности поља дефинисаног у некој интерној *ADAS* компоненти путем механизма за обавештавајуће поље долази до слања информације о промени вредности ка претплаћеној *ServiceProxy* компоненти. Када претплаћена *ServiceProxy* компонента добије обавештење, она одмах извршава слање примљене информације путем механизма за догађај ка *SOME/IP* клијенту који се налази у *IVI* домену.

Са друге стране, спољашње методе или поља за добављање вредности које прозива клијент на *IVI* страни могу бити мапиране на било који унутрашњи механизам. Уколико са *IVI* стране у *ServiceProxy* долази позив методе или поља за добављање вредности који је мапиран на унутрашњи метод или поље за добављање информације на интерној *ADAS* компоненти, захтев ће бити само пропагиран до интерне компоненте која располаже информацијом од интереса. То значи да се у телу прозване методе или руковаоца захтевима за добављање вредности поља само изврши позив мапиране методе или добављања поља са унутрашње стране и да се добијена информација пропагира назад до клијента на *IVI* страни. Уколико са *IVI* стране у *ServiceProxy* долази позив методе или поља за добављање вредности који је мапиран на унутрашњи догађај, обавештавајуће поље или поље за постављање вредности који долазе са интерне *ADAS* компоненте, информација се преноси

посредством заједничког бафера. Унутрашњи механизми по потреби попуњавају бафер подацима које стижу са *ADAS* компонената на *ServiceProxy* тако да се у баферу увек налази актуелна информација. Када са *IVI* стране дође захтев, податак је већ спреман и последња вредност ће бити пружена *SOME/IP* клијенту.

Све методе наведене у дистрибуираном приступу су имплементиране на идентичан начин и у спољашњим спрегама посредничке програмске компоненте (*ServiceProxy*). Међутим, услед мапирања има новина које су махом позиви унутрашњи спрега представљени наредним методама:

- *ara::core::Future<Skeleton> METHOD_NAME (data) override* - метода коју је потребно прозвати над *Proxy* инстанцом унутрашње спреге у оквиру посредничке компоненте како би било започето извршавање одговарајуће удаљене методе на страни *ADAS* компоненте која садржи податке од интереса

- *ara::core::Result<void> RegisterSetHandler(std::function<ara::core::Future<FieldType> (const FieldType& data)> setHandler)* - позивом ове методе над инстанцом поља компоненте *Skeleton* унутрашње спреге региструје се функција *setHandler* која ће бити прозвана на захтев удаљеног клијента (у овом случају друге *ADAS* компоненте) за постављање вредности поља и коју је потребно имплементирати тако да сачува долазни податак који ће касније бити искоришћен или одмах позове неки од три обавештавајућа механизма.

- *ara::core::Result<ara::com::FindServiceHandle> StartFindService(ara::com::FindService Handler<Proxy::HandleType> handler, ara::com::InstanceIdentifier instanceId)* - позивом ове методе над *Proxy* компонентом унутрашње спреге региструје се функција *handler* која ће бити прозвана на промену стања инстанце одређеног сервиса (који се у у овом случају налази на страни друге *ADAS* компоненте). Ово стање може бити дојава да је сервис видљив и доступан и у том случају *handler* је потребно имплементирати тако да настави претплату на одговарајући податак.

- *ara::core::Result<void> Subscribe(size_t)* - позивом ове методе над инстанцом догађаја *Proxy* компоненте унутрашње спреге посредничка програмска компонента даје сигнал да започиње претплату на податке који стижу путем обавештавајућих механизама са сервиса, тј. других *ADAS* компонената.

- *ara::core::Result<void> SetReceiveHandler(ara::com::EventReceiveHandler handler)* - позивом ове методе над инстанцом догађаја *Proxy* компоненте унутрашње спреге региструје се функција *handler* која ће бити прозвана на послати податак са инстанце одређеног сервиса (друге *ADAS* компоненте) и коју је потребно имплементирати тако да сачува долазни податак који ће касније бити искоришћен или одмах позове неки од три обавештавајућа механизма.

- *template <typename F> ara::core::Result<size_t> GetNewSamples(F&& f, size_t maxNumberOfSamples = std::numeric_limits<size_t>::max())* - позивом ове методе над инстанцом догађаја *Proxy* компоненте унутрашње спреге извршено је преузимање података пристиглих догађајем са одговарајућег сервиса.

ServiceProxy компонента се састоји из три дела који се налазе у различитим изворним датотекама: *executor*, *adapter* и *skeletonImpl* и-или *proxyImpl*. Као што је већ речено, *ServiceProxy* компонента не поседује никакву логику у виду манипулације које утиче на вредност прослеђиваних информација или временског тренутка када ће информације бити прослеђене.

Executor служи да контролише извршавање компоненте, врши иницијализацију, постављање сигнала који су спрегнути са распоређивачком компонентом која контролише извршавање свих других компонената у стеку (*ara::exec*) како би било могуће извршити терминирање компоненте на захтев распоређивача. Након тога је позивом методе *ara::core::Result<void> ExecutionClient::ReportExecutionState(ExecutionState state) const noexcept* над *ara::exec::ExecutionClient* компонентом задато да је посредничка компонента у стању извршавања. Овде је извршена и иницијализација компоненти за праћење стања и исписа (*ara::log::Logger*) позивом методе *Logger& CreateLogger(ara::core::StringView ctxId, ara::core::StringView ctxDescription, LogLevel ctxDefLogLevel) noexcept* задајући један од нивоа исписа *(Off, kFatal, kError, kWarn, kInfo, kDebug, kVerbose)*. У основи је задат ниво који искључиво приказује проблематична стања, док је по потреби могуће смањити или повећати ниво детаља.

Очигледно је да се механизми комуникације различито користе у случају имплементације клијентске, односно имплементације сервисне стране. Из тог разлога постоје две компоненте *proxyImpl* и *skeletonImpl*. Механизми комуникације који користе *Proxy* компоненту средњег слоја су имплементирани тако што су позвани непосредно над *Proxy* компонентом која је у виду композиције садржана као поље компоненте *proxyImpl*. Са друге стране, механизми који користе *Skeleton* компоненту средњег слоја су имплементирани тако што компонента која их имплементира мора да наследи *Skeleton*. Како би било могуће имплементирати виртуелне методе из генерисаних *Skeleton* компонената сваког од спрежних пролаза, неопходно је било имплементирати класу која наслеђује одговарајући *Skeleton*. Све ове изведене класе су садржане у *skeletonImpl* датотеци и у њима је имплементирано већ објашњено мапирање.

Adapter служи да креира све инстанце објеката наслеђених класа из *skeletonImpl* и *proxyImpl* компоненти чиме започиње извршавање механизама и оживљавају сви спрежни пролази. Није било могуће имплементирати све функционалности у *Adapter* компоненти јер би онда она морала да наслеђује *Skeleton* компоненте за све спрежне пролазе чиме би постојало вишеструко наслеђивање, које није дозовољено по *MISRA* правилима кодовања.

Из табела у поглављу 6 за верификацију којима су представљена мерења кашњења које уводи пренос података користећи *SOME/IP* механизме могуће је видети како су разлике између дистрибуираног и централизованог приступа релативно мале. У случајима централизованог приступа додатних $\sim$ 2ms када је реч о методи или пољу за добављање вредности, односно $\sim$ 1ms када је реч о догађајима, обавештавајућим пољима или пољима за постављање вредности.

Табела 5: Поређење *ara::com* и *CommonAPI* средњег слоја

Категорија	**Поткатегорија**	**ara::com**	**CommonAPI**
CPU		0.03%	0.03%
Меморија	величина програмског стека	86 MB	76 MB
	величина клијентске апликације	2.2 MB	300 kB
Кашњење	иницијализација клијента	1074 ms	1899 ms
	добављање података на захтев	8.711 ms	8.761 ms
	добављање података на претплату	3.959 ms	3.962 ms
Зависности		vsomeip boost RapidJSON DLT*	vsomeip

*diagnostic, log и trace модули

4.1.2 Имплементација на страни инфо-забавног домена

Што се тиче имплементације на *IVI* страни, први део истраживања је био усмерен ка одлуци који од модула средњег слоја који имају подршку за *SOME/IP* протокол користити, *ara::com* или *CommonAPI*. Након њихове анализе, мерења и поређења спроведених на *Nvidia Jetson AGX* платформи са резултатима датим у табели 5 одлучено је да *CommonAPI* представља боље поклапање. Највећи разлог је то што је *ara::com* модул средњег слоја ипак дизајниран тако да буде део веће платформске целине па повлачи за собом минималне зависности од дијагностичарских компонената за праћење и бележење промена и стања. Ово није случај са *CommonAPI* модулом средњег слоја као што се може видети по величини компајлираних библиотека (коришћењем истог компајлера и компајлирањем за исту платформу) што га чини једноставнијим. Из тог разлога, први део решења у инфо-забавном домену представља имплементирање блиблиотека *CommonAPI* клијената који преко *SOME/IP* протокола комуницирају са одговарајућим сервисима тако да их је могуће увезати у било који програм који се извршава.

4.1.2.1 *CommonAPI SOME/IP* клијент *CommonAPI* клијент је имплементиран тако да омогућава манипулацију *SOME/IP* механизмима најпре стварањем алијаса одговарајућег именског простора за приступ генерисаним функцијама, нпр. *CommonAPIClient* и објекта *CommonAPI::Runtime*, а затим позивањем следећих функција за иницијализацију радног окружења као што је приказано на слици 20.

У случају када *CommonAPI* клијент представља заиста клијентску страну у питању су следеће функције:

- *bool initializeProxyInterface (int32_t ID)* - главни задатак је да изврши стварање *Proxy* објекта који одговара одређеној инстанци сервиса и над којим ће касније бити прозиване клијентске методе. За конструисање *Proxy* објекта је позвана функција *std::shared_ptr<CommonAPIClient::InterfaceProxy<>> buildProxy<CommonAPIClient::Interface> (std::string domain, std::string instanceID,*

```
// уколико је  клијент заправо клијентска страна спреге
#include "CommonAPI_generatedFiles"

namespace CommonAPIClient = v0_1::lava::ServiceInterfaces;

static std::unordered_map<int32_t,
std::shared_ptr<CommonAPIClient::InterfaceProxy<>>> clientsMap;

CommonAPI::Runtime  g_runtime;

bool initializeProxyInterface (int32_t ID) {
   proxyClient =
g_runtime.buildProxy<CommonAPIClient::InterfaceProxy> (LOCAL,
ID, "CommonAPIIterafceClient");
   clientsMap.insert({ID, spClient});
}

bool deinitializeProxyInterface (int32_t ID) {
   proxyClient = clientsMap.find(ID);
   proxyClient->second.reset();
}

 // поље за добављање вредности
 //  int8_t getFieldFromInterface(int32_t ID, OutputArg)
int8_t MethodFromInterface(int32\_t ID, InputArgs, OutputArg)  {
   proxyClient = clientsMap.find(ID);
   OutputArg = proxyClient->second.callSomeipMethod(inputArgs);
}

// поље за обавештавање
// int64_t subscribeFieldFromInterface(int32_t ID, void
(*listener)(FieldDataType))
int64_t subscribeEventFromInterface(int32_t ID, void
(*listener)(EventDataType)) {
   proxyClient = clientsMap.find(ID);
   proxyClient->second.Event.subscribe(listener);
}

int8\_t unsubscribeEventFromInterface(int32\_t ID, uint32\_t\&
functionListener Number) {
   proxyClient = clientsMap.find(ID);
   proxyClient->second.Event.unsubscribe();
}
```

```
// уколико је  клијент заправо сервисна страна спреге
#include "CommonAPI_generatedFiles"

class StubImpl : CommonAPI::Stub {}

static std::unordered_map<int32_t,
std::shared_ptr<StubImpl<>>> clientsMap;

CommonAPI::Runtime  g_runtime;

bool initializeStubInterface (int32_t ID) {
  StubImpl stubClient;
  g_runtime.runtime->registerService(LOCAL, ID, stubClient);
   clientsMap.insert({ID, stubClient;
}

bool deinitializeStubInterface (int32_t ID) {
   stubClient = clientsMap.find(ID);
   delete stubClient->second.;
}

// поља за постављање вредности
int64_t subscribeSetterFromInterface(int32_t ID, void
(*listener)(FieldDataType)) {
   stubClient = clientsMap.find(ID);
   stubClient->second.setSetterListener(listener);
}
```

Слика 20: *CommonAPI* клијент

std::string connection) над *CommonAPI::Runtime* објектом. Овако конструисана инстанца смештена је, под идентификатором коришћеним као кључ, у мапу свих *Proxy* објеката који ће бити направљени и коришћени за размену података са одговарајућом инстанцом сервиса на *ADAS* страни

- *bool deinitializeProxyInterface (int32_ t ID)* - врши ослобађање ресурса радног окружења односно *Proxy* објекта када комуникација више није потребна

Уколико *CommonAPI* клијент представља сервисну страну, тј. уколико спрега поседује поље за постављање вредности, у питању су следеће функције:

- *bool initializeStubInterface (int32_ t ID)* - креира *Stub* објекат класе која наслеђује генерисану *Stub* компоненту. Затим је потребно овако конструисан *Stub* објекат регистровати као сервис позивом функције *bool registerService<CommonAPIClient::Interface> (std::string domain, std::string instanceID, Stub*, std::string connection)* над *CommonAPI::Runtime* објектом. Овако конструисана инстанца смештена је, под идентификатором коришћеним као кључ, у мапу свих *Stub* објеката који ће

бити направљени и коришћени за размену података са одговарајућим клијентом на *ADAS* страни

- *bool deinitializeStubInterface (int32_t ID)* - врши ослобађање ресурса радног окружења и *Proxy* објекта (односно *Stub* објекта уколико постоји што ће бити објашњено у наставку) када комуникација више није потребна

Даље је потребно имплементирати функције за спрегу ка *SOME/IP* позивима

- *int8_t MethodFromInterface(int32_t ID, InputArgs, OutputArg)*- служи за позивање удаљене методе на сервис страни путем *CommonAPIClient::<METHOD_NAME>(InputArgs, CallStatus, OutputArgs)* методе иницијализованог *Proxy* клијентског објекта и прослеђивање добављене вредности са сервисне компоненте

- *int8_t getFieldFromInterface(int32_t ID, OutputArg)* - служи за позивање методе за добављање вредности поља на сервис страни путем *CommonAPIClient::Get()* методе иницијализованог *Proxy* клијентског објекта и прослеђивање добављене вредности са сервисне компоненте

- *int64_t subscribeEventFromInterface(int32_t ID, void (*listener)(EventDataType))* - представља почетак претплатничког режима на одговарајући догађај и чување функције повратног позива која ће бити прозвана и којом ће жељени податак бити прослеђен када догађај стигне са сервис компоненте позивом *CommonAPIClient::subscribe(std::function<EventType>())* методе над иницијализованим *Proxy* објектом

- *int8_t unsubscribeEventFromInterface(int32_t ID, uint32_t& functionListenerNumber)* - представља позивање функција за прекидање претплатничког режима ка одговарајућем догађају позивом *CommonAPIClient::unsubscribe()* методе над иницијализованим *Proxy* објектом

- *int64_t subscribeFieldFromInterface(int32_t ID, void (*listener)(FieldDataType))* - представља почетак претплатничког режима на одговарајуће обавештавајуће поље и чување функције повратног позива која ће бити прозвана и којом ће жељени податак бити прослеђен када догађај стигне са сервис компоненте позивом *CommonAPIClient::subscribe(std::function<FieldType>())* методе над иницијализованим *Proxy* објектом

- *int8_t unsubscribeFieldFromInterface(int32_t ID, uint32_t& functionListenerNumber)* - представља позивање функција за прекидање претплатничког режима ка одговарајућем обавештавајућем пољу позивом *CommonAPIClient::unsubscribe()* методе над иницијализованим *Proxy* објектом

- *int64_t subscribeSetterFromInterface(int32_t ID, void (*listener)(FieldDataType))* - представља почетак претплатничког режима на одговарајуће поље за постављање

вредности и чување функције повратног позива која ће бити прозвана и којом ће жељени податак бити прослеђен када информација стигне са сервис компоненте, метода за отказивање претплате није потребна. У својој имплементацији врши позив *CommonAPIClient::setSetterListener(std::function<FieldType>())* методе над иницијализованим *Stub* објектом

Наравно, могуће је комбиновати клијентске и сервисне спреге уколико компонента са којом комуницира *CommonAPI* клијент поседује вишеструке спрежне пролазе.

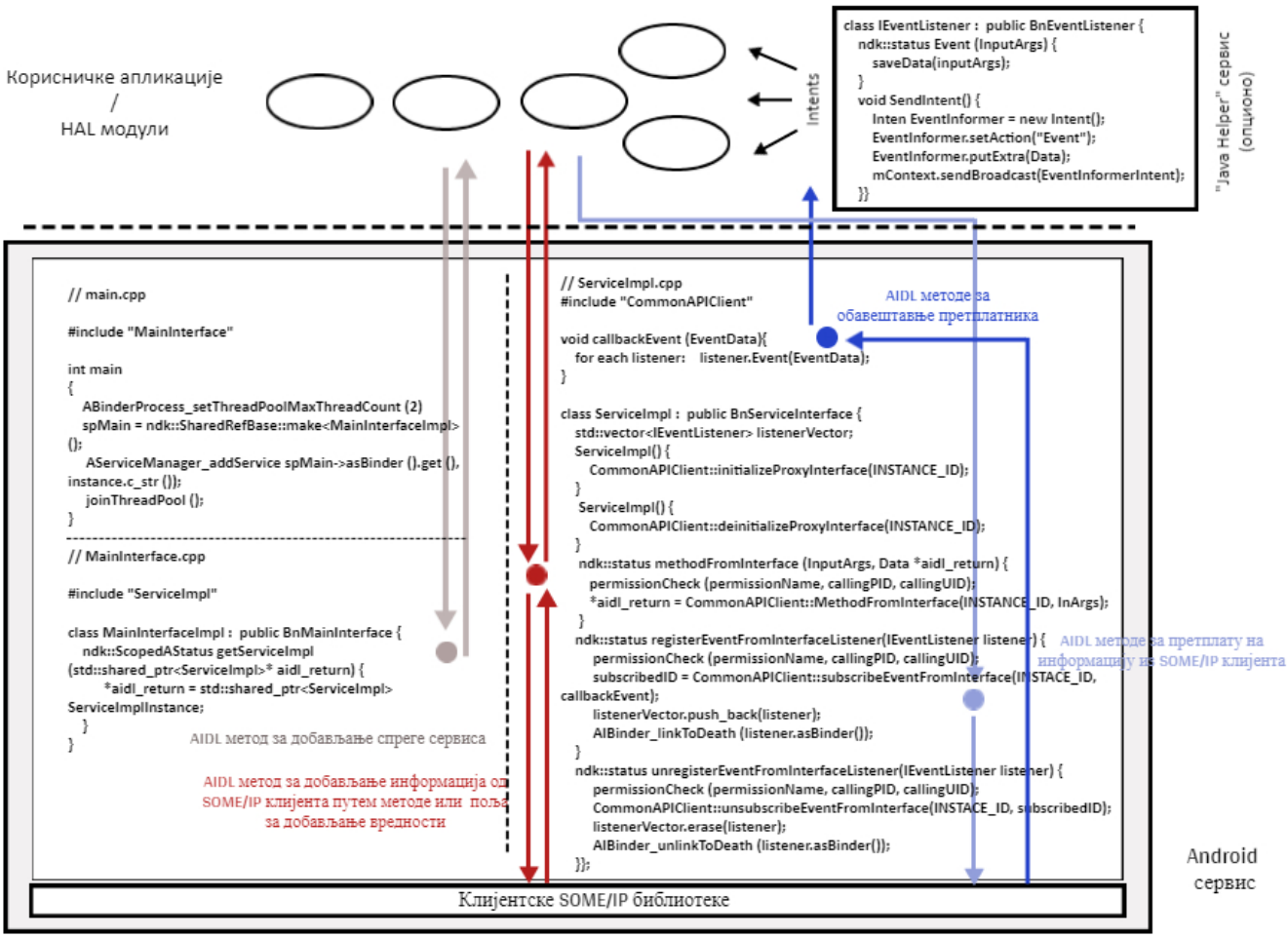
4.1.2.2 Интеграција у *Android OS* С обзиром на специфичност међупроцесне комуникације која настаје од језика за дефинисање спреге у *Android* оперативном систему и сам тренд примене *Android* система, додатни допринос решења биће остварен имплементацијом механизма који омогућава интеграцију једног или више *CommonAPI SOME/IP* клијената у *Android OS*. Такву подршку је могуће остварити на неколио начина:

- непосредно уз апликацију - интегрисати *CommonAPI SOME/IP* клијента у оквиру истог процеса који представља апликација, у изворни код апликације или путем повезаног сервиса кроз *AIDL* спрегу. Предност у овавком случају је брзина, а уколико је искључиво једна апликација заинтересована за информације предност су и ресурси и процес интеграције. Међутим, уколико исту информацију желе да добаве и други модули или корисничке апликације, неопходно је поновити процес што драстично квари перформансе, ресурсе и лакоћу интеграције на нивоу читавог система. Такође, комуникациони механизам зависи од процеса за који је везан што може бити неповољно у случају када сам процес корисничке апликације оде у неко од пасивних стања (*onPause* или *onStop* сигнали), немогуће је примити и чувати релевантне информације са *ADAS* домена и искористити их када процес апликације буде поново активан.

- непосредно уз модул апстракције слоја физичке архитектуре - интегрисати *CommonAPI SOME/IP* клијента у оквиру истог процеса који представља сервис који располаже коришћењем ресурса физичке архитектуре кроз одговарајући модул, у изворни код имплементације модула или путем повезаног сервиса кроз *AIDL* спрегу. Као и у претодној опцији, предност могу бити употреба ресурса и брзина, међутим, уколико исту информацију желе да добаве други модули или корисничке апликације, неопходно је поновити процес што негативно утиче на перформансе и лакоћу интеграције на нивоу читавог система.

- сервис у првом плану - интегрисати *CommonAPI SOME/IP* клијента у оквиру засебног процеса. Овиме се добија нешто веће заузеће, али се зато добија на контролабилности уклањањем зависности од сваког другог процеса. Додатно, овај приступ омогућава ефикасно достављање истих података различитим корисничким апликацијама и коришћеним модулима апстракције физичке архитектуре.

Изабрано решење погодно за *Android OS* представља стварање сервиса покренутог у првом плану који ће интегрисати претходно описаног *CommonAPI* клијента и кроз методе *AIDL* спреге пружити његове функционалности корисничким апликацијама које желе



Слика 21: *Android C++* сервис

да имају приступ ресурсима из *ADAS* домена. Подржани су *Java*, *Kotlin* и *C++* језици у којима је створен сервис, а разлике у њиховим перформансама ће бити адресиране у поглављу 6.

Како би подршка за *Android OS* била успешно имплементирана било је потребно превазићи разлике у парадигми сервисно-оријентисане комуникације омогућене *SOME/IP* протоколом и *Android binder* механизмом који се користи у *AIDL* методама. Две главне разлике представљају највеће изазове. Прва произилази из тога што *AIDL* не подржава концепт компоненте, већ само опис комуникационе спреге. Друга је последица тога што *binder* механизам подржава размену података искључиво путем захтева које креира клијент.

SOME/IP подржава вишеструку имплементацију исте комуникационе спреге у виду различитих инстанци путем параметара садржаних у *ARXML* или *FDEPL* датотекама и могућност придруживања ових инстанци једној или већем броју компонената. Међутим, као што је већ речено, *AIDL* језик у основи не подржава параметре којима би се оваква дисјунктност могла пренети до корисника, а неопходно је пружити корисницима на *IVI* страни могућност да интерагују са произвољним инстанцама сервиса из *ADAS* домена. За то је, као што је демонстрирано на слици 21, потребно направити:

- главну датотеку у којој је број нити које одређују колико корисни-

ка може истовремено да приступа сервису постављен на два функцијом *ABinderProcess_setThreadPoolMaxThreadCount(2)*. Одлучено је поставити број нити на два како би у теоријском смислу било омогућено истовремено додавање новог корниска, као и послуживање постојећег (иако у реалности није могуће контролисати који посао ће нити прихватати). Затим је сервис главне *AIDL* спреге додат у компоненту за управљање сервиса функцијом *status_t AServiceManager_addService (const sp<IBinder>& service, const String16& name)*, да би на самом крају била позвана функција *joinThreadPool()* која обезбеђује да сервис неће бити прекинут пре времена.

- *MainInterfaceImpl* главну комуникациону спрегу описану *AIDL* језиком која наслеђује генерисани део сервиса *BnMainInterface* и укључује остале комуникационе спреге описане истим језиком тако да се *binder* објекат који омогућава комуникацију путем сваке појединачке спреге добија као повратна вредност метода садржаних у главној *AIDL* спрези као што је приказано на слици 21.

- *ndk::ScopedAStatus getServiceImpl (std::shared_ptr<ServiceImpl>* aidl_return* - метода коју позива корисничка апликација и која као повратну вредност враћа статус извршавања, а кроз аргумент *aidl_return* даје објекат конкретног сервиса над којим је потребно позивати методе за остваривање *SOME/IP* саобраћаја. Уколико постоје спреге истог имена, биће разликоване на основу пакета задатог у *.aidl* датотеци који у себи садржи идентификатор инстанце

- *ServiceImpl* класа која имплементира конкретну спрегу са *ADAS* страном и наслеђује генерисани део сервиса *BnServiceInterface* одговарајуће *AIDL* спреге. Свака од ових спрега одговара конкретној инстанци комуникационог сервиса на *ADAS* страни и у себи садржи интегрисане *CommonAPI* позиве који омогућавају комуникацију баш са том конкретном инстанцом сервиса задатом кроз пакет спреге. На овај начин, свака корисничка апликација путем одговарајућих *AIDL* метода главне комуникационе спреге добија *binder* објекат којим је описана комуникациона спрега са жељеном инстанцом сервиса на *ADAS* страни и даљим позивањем метода добијене спреге управља разменом података.

- при конструктору сервиса за сваку специфичну *AIDL* спрегу, позвана је функција *CommonAPIClient::initializeProxyInterface(INSTANCE_ID)* како би одговарајући *CommonAPI* клијент био иницијализован. Такође, у деструктору је позвана функција *CommonAPIClient::deinitializeProxyInterface(INSTANCE_ID)* за уништавање одговараућег клијента.

Решавање другог проблема подразумева мапирање и пружање сваког од описаних *SOME/IP* механизама кроз *AIDL* методе специфичне комуникационе спреге позване од корисничких апликација као што је приказано на слици 21.

- У случају када корисничка апликација потражује ресурс са *ADAS* стране који стиже путем *SOME/IP* методе или поља за добављање вредности, позиви ових

SOME/IP механизама су изложени корисничким апликацијама кроз позиве *AIDL* метода из одговарајуће спреге, као што је на слици приказано примером имплементиране методе *ndk::status methodFromInterface (InputArgs, Data *aidl_return)* у којој је повратна вредност статус извршавања, а подаци добијени путем *SOME/IP* позива изложених кроз *CommonAPI* клијента су кориснику дати последњим аргументом. Овако имплементиран *AIDL* метод у себи најпре има проверу дозволе приступа сервису путем управљачке компоненте *Permission Manager* и методе *int32_t APermissionManager_checkPermission(const char *permission, pid_t pid, uid_t uid, int32_t *outResult*). Уколико је резултат провере позитиван настављено је ка одговарајућем *SOME/IP* позиву из *CommonAPI* клијента

- Међутим, за остваривање догађаја и преосталих врста поља, неопходно је имплементирати по две *AIDL* спреге за жељени *SOME/IP* механизам – једну којом корисничка апликација на сервису кроз аргумент *AIDL* методе *ndk::status registerEventFromInterfaceListener(IEventListener listener)* проследи и тако региструје *binder* објекат друге *AIDL* спреге *IEventListener*. Приликом позване регистрације, *Android* сервис на исти начин најпре проверава дозволе приступа, а затим уколико ни један корисник до сад није претплаћен, путем *CommonAPI* клијента извршава претплату своје функције *void callbackEvent (EventData)* на одговарајући сервис из *ADAS* домена. Када пристигне један од обавештавајућих механизама (догађај, поље за обавештавање и поље за постављање) долази до позивања претплаћене функције која даље позива метод друге *AIDL* спреге (*ndk::status Event (InputArgs)*) који је имплементиран у корисничкој апликацији. Метод *ndk::status Event (InputArgs)* ће бити прозван за сваког корисника чији је *binder* објекат регистрован у сервису, а као аргумент је прослеђен податак пристигао из *ADAS* домена. Додатно, при регистрацији је имплементирана и функционалност *"Link to death"* која омогућава сервису да добије сигнал уколико је корисник који држи регистровани *binder* објекат завршио процес или уништио објекат са своје стране чиме га може избацити из складиштених ослушкивача.

- Корисник у сваком тренутку може затражити заустављање претплате на обавештавајући *SOME/IP* механизам аналогно регистрацији позивом методе *ndk::status unregisterEventFromInterfaceListener(IEventListener listener)* која ће бити имплементирана на начин да избаци регистровани *binder* објекат који има улогу ослушкивача из вектора регистрованих, чиме корисник неће бити обавештен на наредне пристигле податке. С обзиром на то да више метода које могу бити извршаване у различитим нитима приступају овом вектору, приступ њему је заштићен механизмима *std::mutex*, *std::unique_lock* и *std::condition_variable* из стандардне библиотеке *C++* језика.

Такође, руковање пристиглим подацима на претплаћеног *SOME/IP* клијента је вршено на два начина у зависности од тога да ли је активиран квалитет услуге или не:

- Уколико није активан квалитет услуге, све методе које позива *Android C++* сервис и које су имплементиране од стране корисничких апликација над регистрованим

ослушкивачима су *oneway* типа и прозване су секвенцијално у for петљи као што је приказано на слици 21

- Уколико је активан квалитет услуге, методе нису *oneway* типа, односно нису извршаване асинхроно, већ је неопходно чекање на њихову терминацију и повратну информацију која потврђује да је корисничка апликација обрадила прослеђени податак. За ову сврху је имплементиран посебан распоређивач.

Распоређивач долазећих обавештавајућих механизама је имплементиран како би осигурао правилан пренос информација без блокаде, уз минимално кашњење и додатно могућност приоритизације корнисничких апликација које су у листи заинстересованих за истоветан податак. Компонента је предстаљена под називом *EventHandlingManager - EHM* и функционише на начин описан у наставку:

- Приликом регистровања ослушкивача од стране корисничких апликација или *HAL* модула се проверава приоритетна вредност процеса из ког долази захтев. Ова приоритетна вредност се назива *nice value* или *niceness* и коришћена је од стране *Linux* језгра приликом распоређивања извршавања процеса. Има распон вредности од -20 до 19, где је -20 најприоритетнија, а 19 најмањег приоритета. Ова вредност се придружује сваком ослушкивачу приликом његовог чувања у вектор у самом *Android* сервису.

- За сваки пристигли догађај или поље за обавештавање или постављање вредности, *EHM* ствара објекат од сваког појединачног претплаћеног ослушкивача. Поред самог ослушкивача, овај објекат садржи и информацију коју је потребно проследити као и показивач на функцију у којој ће корисничка апликација бити позвана. Затим следи рачунање коефицијента приоритета.

- Рачунање коефицијента приоритета (P_{coef}) је извршено на основу времена приспећа (T_{ar}) података са *ADAS* домена, првобитно убележене вредности приоритета додељене корисничкој апликацији - *niceness* (N_{coef}), информације о броју претходно добијених обавештења која још увек нису достављена крајњем кориснику (R_{obj}) и фактору учесталости (F_{occ}) пристизања овог обавештења по следећој формули.

$$\begin{aligned} P_{coef} =& T_{ar} + N_{coef} - K_{ro} * R_{obj} - F_{occ} \\ T_{ar} =& T_c - T_{ref}[ms] \end{aligned} \tag{1}$$

Учесталост представља бројач који се рачуна кроз читаво извршавање сервиса са периодом постављања нулте почетне вредности од 1 секунд. Разлог за то је немогућност утврђивања учесталости за сваки обавештавајући механизам унапред. Исти период је коришћен и за поновно рачунање времена приспећа. Време приспећа представља разлику времена забележеног у тренутку доласка обавештавајућег механизма (T_c) и референтног времена (T_{ref}) забележеног у сервису које се ажурира на претходно поменути период. Фактор времена приспећа је изражен у милисекундама. Тежински

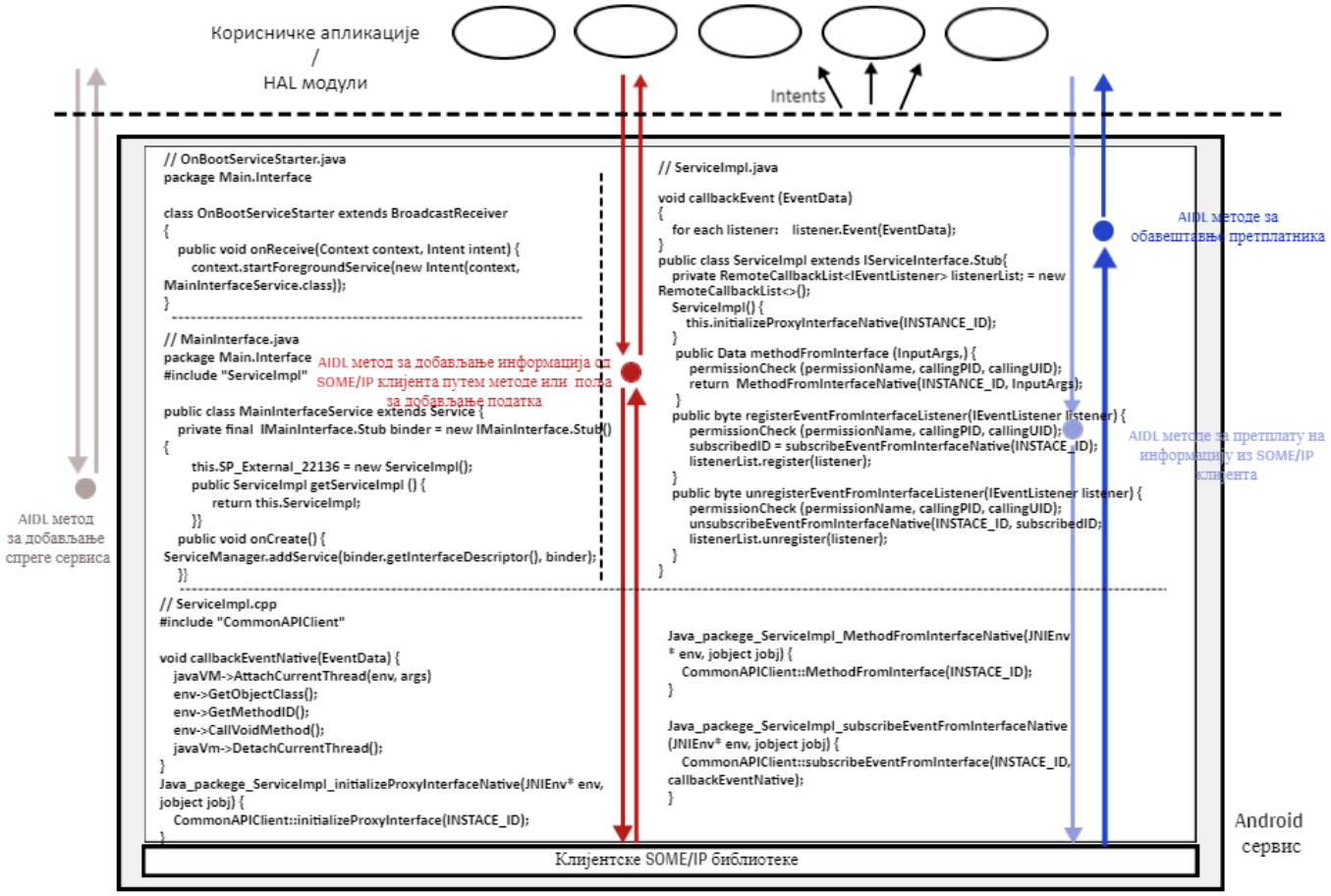
коефицијент за број преосталих необрађених информација (K_{ro}) је 40 како би могао у потпуности да поништи *niceness*.

- Након што је објекат пристиглог претплатничког података (кроз догађај или поље) створен и коефицијент приоритета израчунат, извршено је његово отпремање у мапу где је аутоматски сортиран на основу израчунатог коефицијента приоритета. Са друге стране, руковалац за складиште нити (енгл. *Thread Pool*) упошљава слободне нити извршавајући у њима позиве функција чији су показивачи сачувани у сортираним објектима из мапе узимајући редом, најприоритетнији објекат догађаја, односно поља. Руковалац за складиште нити је имплементиран тако да поред позива *AIDL* метода које се извршавају у корисничким апликацијама прати и време њиховог извршавања. Уколико време извршавања метода на страни корисника прелази 10ms, ова нит бива откачена (енгл. *Detached*) и корисник је сматран опасним за рад сервиса и бива избачен из вектора претплаћених корисника на тај одређени податак. Овиме сервис захтева од корисника да на својој страни изврши искључиво преузимање одговарајућег података, како би утицај имплементације корисничких функционалности на сервис био минимизиран, а ефикасност рада сервиса у виду брзине опслуживања клијената увећана.

Као што је речено, *Android* сервис је имплементиран у различитим језицима (*C++*, *Java* и *Kotlin*) како би разлике у перформансама биле испитане и како би подршка у интеграцији за различите случаје коришћења била омогућена. Постоје две основне разлике у функционалности *C++* и *Java/Kotlin* сервиса, као што се може приметити на слици 22, а то је обавезно коришћење *Java Native Interface - JNI* слоја (у случају *Java/Kotlin* сервиса) и могућности прослеђивања података који долазе путем *SOME/IP* механизама ка више заинтересованих клијената.

Java и *Kotlin* сервиси су приказани на слици 22 и реализовани су готово на истоветан начин као што је то ситуција за *C++* сервис. Овде је, такође, присутна главна *AIDL* спрега која корисничким апликацијама пружа објекте конкретних спрега којима је изложен приступ одговарајућим инстанцама сервиса на *ADAS* страни позивањем истовентних метода као што је случај у *C++* сервису јер настају од истих *.aidl* датотека. Разлике у имплементацији су следеће:

- Да би било омогућено покретање *C++* сервиса при покретању саме платформе довољно је додати команду за његово покретање у *init.rc* скрипти која контолише подизање почетних процеса. Са друге стране, у случају *Java* и *Kotlin* сервиса је у оквиру почетне *OnBootServiceStarter.java* датотеке (која представља пандан main.cpp датотеци) имплементирана класа која проширује *BroadcastReceiver* што јој омогућава да реагује на *Intent* сигнал који означава покретање платформе. Хватање одговарајућег *Intent* сигнала је извршено методом *public void onReceive(Context context, Intent intent)* која извршава покретање сервиса у првом плану функцијом *final void startForeground (int id, Notification notification))* којој је прослеђен *Intent* конструисан тако да врши побуду класе *MainInterfaceService* у којој је имплементирана главна *AIDL* спрега



Слика 22: *Android Java* сервис

- Класа *MainInterfaceService* проширује системску класу *Service* која при покретању врши додавање сервиса помоћу *ServiceManager* компоненте као што је то рађено у *C++* сервису. Међутим, у овој класи је имплементирано и конструисање *binder* објекта на основу генерисаног сервисног (*Stub*) дела главне спреге како би било омогућено имплементирање метода које као повратну вредност враћају инстанце циљаних *AIDL* спрега

- Класа која имплементира циљану *AIDL* спрегу је логички поново имплементирана на исти начин, тако што проширује генерисани сервисни део спреге чиме је омогућена имплементација свих потребних метода које излажу *SOME/IP* механизме. Међутим, није било могуће одмах позивати аналогне методе из *CommonAPI* клијента јер је он написан у *C++* програмском језику па је било неопходно укључити употребу *Java Native Interface - JNI* спреге. *JNI* је имплементиран тако што је на називе циљаних метода из *CommonAPI* клијента додат суфикс *"Native"*. Дакле, при позивима метода из одговарајућих *AIDL* спрега, такође се најпре врши провера дозвола, али се потом *CommonAPI* клијент не позива непосредно, већ позивима *JNI* спреге. *JNI* спрега је имплементирана у *C++* програмском језику тако што прослеђује задати *INSTANCE_ID* у позиве *CommonAPI* клијента и враћа добијене податке назад

- Поред генералне разлике у коришћењу коју уводи *JNI*, највећа разлика је у реа-

лизацији обавештавајућих механизама. За њихову употребу је било неопходно имплементирати две функције повратних механизама, једну у *Java* делу сервиса, а другу у *C++* делу. Функција *void callbackEventNative(EventData)* која је имплементирана у *C++* делу је првобитно претплаћена *CommonAPI* клијенту, а њеним прозивањем је потребно прозвати компоненте за добављање информација о повратној функцији из *Java* дела коју је потребно прозвати. Добављање информација је извршено позивима *AttachCurrentThread()* и *DetachCurrentThread()* над *javaVM* као и *GetObjectClass()* и *GetMethodID()* над *JNIenv* компонентама које садрже информације о контексту *Java* објекта са којим је извршавање *C++* функција спрегнуто. Кад постоји информација о повратној функцији коју треба позвати врши се њено позивање методом *CallVoidMethod()* прослеђујући пристигли податак са *ADAS* домена до *Java* дела сервиса. Даље је у *Java* делу имплементирано прозивање корисничких апликација над регистрованим ослушкивачима који су у случају *Java* и *Kotlin* сервиса складиштени у *RemoteCallbackList* листи, уместо вектору као што је случај у сервису. *RemoteCallbackList* листа је коришћена због аутоматских механизама за синхронизацију при освежавању стања у случају терминације корисника који су у њој складиштени или додавања нових, што није случај са обичним вектором где је то имплементирано додатно

- У случају *Java* сервиса не постоји коришћење *EHM* компоненте, већ је при активираном квалитету услуге извршавање праћено путем асинхроних *future* механизама који у *Java* програмском језику омогућавају терминацију нити уколико корисник премаши допуштено време извршавања

- *Kotlin* сервис је имплементиран на истоветан начин као и *Java* уз једину разлику што за обавештавање корисника није коришћен *future* механизам, већ механизам корутина (енгл. *Coroutines*) које су у случају коришћња *Kotlin* програмског језика повлашћен приступ за асинхрону комуникацију

Додатно, *Java* и *Kotlin* сервиси пружају могућност преношења информације до корисничких апликација путем механизама који се у *Android* оперативном систему назива *Intent*, а који у *C++* имплементираном сервису није доступан. *Intent* омогућава дифузно емитовање садржаја и побуђивање корисничких апликација прослеђивањем објекта за размену података са намером да покрене одређену активност из другог процеса. Искуствено је утврђено (детаљи у поглављу за верификацију) да је непосредно слање ка свакој апликацији путем *binder* механизма у случају када има више од 50 обавештавајућих задатака (нпр. 50 корисника претплаћених на 1 догађај или 10 корисника претплаћених на 5 догађаја) у случају *C++*, односно више од 10 у случају *Java* сервиса лошије у виду перформанси него коришћење *Intent* механизма. Стога, како би се постигла потпуна прилагодљивост решења сервиси су имплементирани тако да се механизам слања података пристиглих *SOME/IP* догађајем, пољем за обавештавања или пољем за постављање пребаци са прозивања регистрованих ослушкивача у виду *binder* објеката на дифузију *Intent* објеката. У случају *C++* сервиса имплементирана је помоћна компонента у *Java* језику (*Java Helper* на слици 21) која се понаша као корисничка апликација и којој се подаци ша-

љу само у случају када број корисника пређе дефинисану границу. Унутар ове компоненте је имплементирано преузимање пристиглих објеката кроз имплементирану методу *AIDL* спреге за обавештавајуће механизме, а затим конструисање *Intent* објекта, складиштење потребне информације и напослетку дифузија. У случају *Java* и *Kotlin* сервиса није неопходно постојање помоћне компоненте јер је могуће извршити дифузију, непосредно, из самих сервиса.

Да би сервис испунио захтев за могућност добављања података како са компонената виших слојева програмске подршке тако и са ресурса физичке архитектуре *ADAS* домена, неопходно је остварити могућност отпремања података ка корисничким апликацијама и модулима апстракције физичке архитектуре истовремено што утиче на стварање и распоређивање самог сервиса на систему. На пример, *Java* и *Kotlin* сервиси могу бити додати искључиво путем *ServiceManager* компоненте из *Java* спреге радног оквира. Из тог разлога, они се не могу наћи у */vendor* партицији и комуницирати непосредно са осталим компонентама из партиције из разлога што у овој партицији *ServiceManager* компоненте из Java спреге радног оквира није видљива по основи. Зато их је могуће креирати само у */system* или */system-exp* партцији и неопходно је написати матрицу усклађености (енгл. *Compatilibity matrix*) која омогућава *binder* механизму да успешно разреши комуникацију међу компонентама различитих партиција на систему. У случају *C++* сервиса ово није случај и они могу бити додати у обе (*/vendor* и */system*, односно */system-exp*) партиције по потреби.

4.1.3 Сигурносни механизми

По питању имплементације сигурносних механизама које је потребно укључити у решење неопходно је било осмислити начин за извршавање правилног шифровања података узимајући у обзир следеће факторе:

- питање ресурса, кашњења и сложености које ће сигурносни механизми увести у решење (подразумева и испитивање коришћења симетричног и асиметричног шифровања)
- да ли је само шифровање довољно?
- начин на који је могуће спрегнути имплементацију сигурносних механизама у оквиру *IDL* језицима дефинисане комуникационе спреге (неусаглашеност типова описаних у језицима за дефинисање спрега и типова које је могуће шифровати, промена у величини податка, додатне ознаке потребне за обезбеђивање правилног рада сигурносних механизама)
- одабир погодног алгоритма који ефикасно нуди имплементацију захтева из претходне три тачке
- начин чувања кључа (генерисање у времену извршавања, чување у меморији, чување диркетно у променљивој извршне датотеке уз маскирање и слично)

Постоје три аспекта увођења сигурности: поверљивост, интегритет и аутентичност. Поверљивост подразумева шифровање података на начин да буду немогући за тумачење за све ентитете сем оних за које су подаци намењени. Појам поверљивост не захтева нужно да шифровани подаци ни не стигну до неовлашћених ентитета или да на путу до дестинационих ентитета остану непромењени, већ је довољан услов за поверљивост то да сви други ентитети који не спадају у оне којима је податак намењен у случају приступа подацима не могу успешно да их протумаче, што их чини безначајним за све друге учеснике сем оних којима је намењено (а то су они који имају одговарајући кључ за правилно дешифровање). Међутим, није довољно обезбедити саму поверљивост из разлога што учесници у мрежи могу опонашати изворне податке ка дестинационим ентитетима и на тај начин их збунити, навести на погрешну реакцију или у потпуности пореметити њихов рад. Поред тога, може постојати ентитет који пресреће изворне податке, мења их (без обзира што не може да их протумачи) и тако измењене прослеђује до дестинационих ентитета (енгл. *Man-in-the-middle*). На тај начин се може произвести неправилно понашање, чак иако је податак потекао са исправног изворишта. Интегритет је појам који пружа гаранцију да је податак непромењен од момента кад је послат са извора, док је аутентичност особина која подразумева обезбеђивање механизма којим се утврђује ко је пошиљалац података. Дакле, за обезбеђивање све три особине није довољно само шифровати/дешифровати податке помоћу једног или више кључева, већ је неопходно у послате податке интегрисати јединствену ознаку пошиљаоца (енгл. *Message Authentication Code - MAC*) и иницијализациони вектор (енгл. *Initializing Vector - IV*).

Типови података који се овим решењем преносе су дефинисани језицима за дефинисање комуникационих спрега, тачније ознакама датим у *ARXML* моделима (претходно је објашњено да је то и генерални начин развоја аутомобилског ситема). Ово даље значи да типови података који ће бити размењивани могу бити произвољни, међутим, то утиче неповољно на имплементацију сигурносних механизама. Разлог је то што је за исправну имплементацију сигурносних механизама потребно у пренетим подацима имати и наведене *MAC* и *IV* вредности, што је немогуће извршити за све дефинисане типове јер су они дефинисањем предвиђени искључиво за пренос корисне информације (почевши од простих, не би било могуће убацити их у целобројни тип (енгл. *Integer*), итд). Из тог разлога је решењем одређено да дефинисани подаци који се размењују између *ADAS* и *IVI* домена морају бити такве структуре, да садрже један низ карактера и два вектора целобројног типа како би у њима били чувани шифровани подаци, али и вредности за *IV* и *MAC*. Да би корисна информација била правилно уметнута у структуру првобитно је на *ADAS* страни извршена серијализација корисног податка који је потребно доставити у *IVI* домен. Након серијализације, овако добијена секвенца бајтова се може исправно шифровати и поред *MAC* и *IV* ознака послати као целина.

Да би сигурносни механизми били ефикасни потребно је да минимално утичу на сложеност интеграције решења, кашњење и оптерећење система на ком се извршава. Из тог разлога је одабрано коришћење симетричног шифровања које је једноставније, а за саму имплементацију се користе функције из *OpenSSL* библиотеке. Као алгоритам је одабран *AES-GCM* 128 из следећих разлога:

- Уколико би се користили алгоритми попут *ECB*, не би било иницијализационог вектора и ознаке за аутентикацију поруке. Без њих је могуће подржати једино поверљивост података, није могуће обезбедити интегритет и аутентичност података. Додатно, *ECB* шифрује сваки блок података независно, што значи да ће излаз шифровања за истоветне податке увек бити исти, што је неприхватљиво.

- Алгоритми попут *CBC* имају иницијализациони вектор, међутим, немају ознаку за аутентикацију. Тиме омогућавају квалитетније шифровање које се разликује у свакој итерацији, без обзира да ли је податак који се шифрује исти или не, али не пружају могућност утврђивања пошиљаоца, тј. уочавања слања поруке са нежељеног извора.

- Уколико би се користили мањи кључеви (нпр. 32 бита), уређаји данашњице могу да прођу кроз све потенцијалне комбинације кључа док не утврде која је исправна, чиме би сигурност била у потпуности нарушена.

Постоји неколико начина типично коришћених за чување кључева при имплементирању сигурносних механизама. Чување кључева незаштићено у меморији и учитавање по потреби је опасно и зато се користе посебни модули физичке архитектуре (енгл. *Hardware Security Module - HSM*) који обезбеђују део меморије заштићен за коришћење. Такође, маскирање (енгл. *Obfuscation*) променљивих и функција у програму које раде са кључем је чест случај да би се спречила могућност разбијања шифре читањем бинарних датотека програма који се извршавају. Један од најсигурнијих начина је генерисање кључева у реалном времену у току извршавања програма. Међутим, овај приступ доноси потешкоће у виду размене таквог кључа са циљем да се омогући правилно шифровање и дешифровање на удаљеним уређајима. У сврху решавања потешкоћа размене кључа потврдом идентитета учесника у размени података чести је примена дигиталних сертификата. За највиши ниво сигурности захтевају учешће овлашћене треће стране од које је потребно затражити сертификат и уколико је захтев у складу са имплементираним механизмима попут исправног шифровања приватним кључем и података о учеснику или уређају првобитно задатих од произвођача. Након што је идентитет учесника потврђен, одговор, односно сертификат садржи приступне податке (од којих један може бити кључ за симетрично шифровање).

С обзиром на то да начин чувања кључа већински зависи од произвођача аутомобила, решење је имплементирано модуларно, тако да може бити прилагођено било којој од наведених практичних примена. Тренутно, као пример, сам генератор је тај који псеудонаусмично генерише кључ. Са друге стране, сертификати нису употребљавани јер се у пракси на различите начине имплементирају од стране различитих произвођача. Притом, типично се користе у процесу првобитног руковања (енгл. *Handshake*) што не решава у потпуности сигурност каснијег саобраћаја. Сигурносни механизми су имплементирани искључиво у генерисаним програмским компонентама при коришћењу *SOME/IP* функција за размену података (односно изнад *ara::com* и *CommonAPI* модула) како не би захтевали промене на нижем нивоу програмског стека и како би важили без обзира на то да ли се за *IVI* домен подаци даље у систем преносе на основу међупроцесног механизма типичног за оперативни систем. На овај начин се не захтева додатни напор у интеграцији решења.

За шифровање преноса података у решењу се користи симетрично шифровање тако што се функцијама *OpenSSL* библиотеке задаје кључ који је, како је већ речено, претходно генерисан од стране генератора. Иницијализациони вектор се генерише насумично и мења за свако слање како би јединствена комбинација кључа и инцијализационог вектора била обезбеђена. Такође, због одабраног алгоритма, подржани су позиви *OpenSSL* библиотеке којима се производи и *MAC* ознака за потврду идентитета за одговарајући пренос података. Након што су сва три податка (шифровани податак, *IV* и *MAC*) спремни смештају се у структуру и прослеђују одговарјућим позивима *ara::com* модула. Са пријемне стране, подаци пристигли *CommonAPI* позивима се издвајају опет употребом функција *OpenSSL* библиотеке, *MAC* ознака се проверава, а информација дешифрује генерисаним кључем.

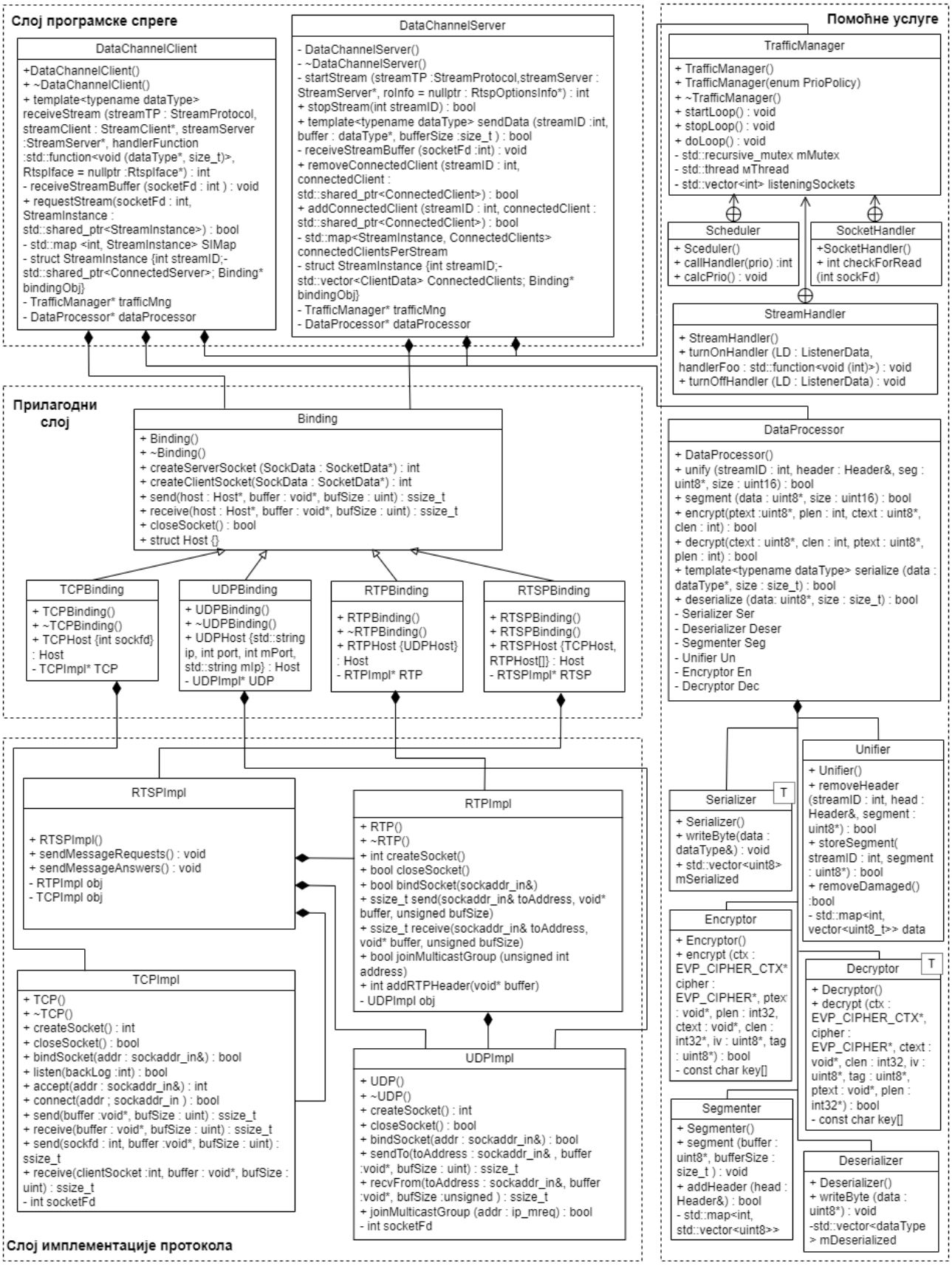
Додатно, у поглављу 6 за верификацију решења дат је удео сигурносних механизама у целокупним перформансама.

4.2 Решење за велике токове података

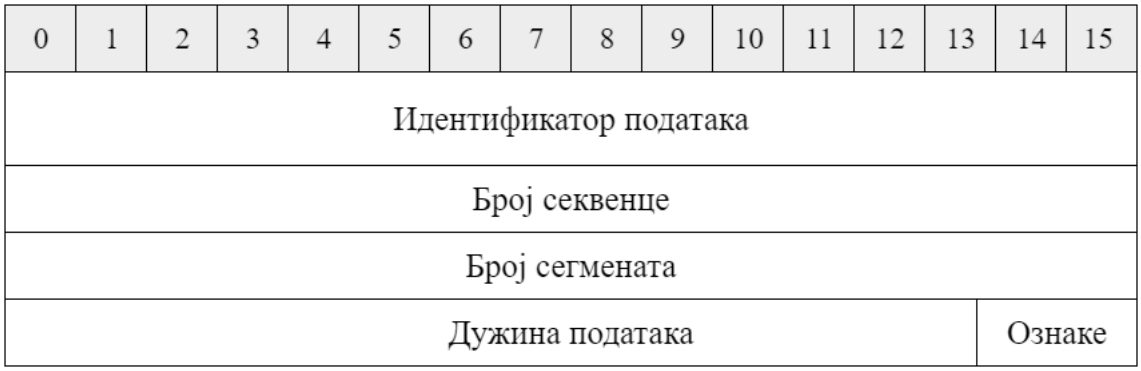
Приликом трансмисије великих токова података чији је редослед од значаја, комуникациони механизми које омогућава *SOME/IP* нису релевантни. Дакле, неопходно је увести додатно усклађивање сегмената великих података који се размењују. Додавање заглавља на постојеће *SOME/IP* заглавље тако да подаци потребни за усклађивање буду пренети и прављење додатног слоја око *SOME/IP* имплементације која делегира, односно рукује удаљеним позивима био би само непотребни продужетак. Из тог разлога је у ову сврху направљен канал за велике податке. У оквиру имплементације овог канала било је потребно: имплементирати транспортне протоколе (1), осмислити и имплементирати управљачку компоненту која диригује слање, пријем и врсту распоређивања саобраћаја (2), укључити одговарајући механизам за обезбеђивање сигурности (3), осмислити изглед и креирати једноставано заглавље којим ће бити омогућена правилна сегментација великих података задатих од корисничких апликација (4), имплементирати серијализацију и сегментацију (5), имплементирати компоненте средњег слоја које ће раздвојити специфичности сваког од коришћених протокола (6) и на крају имплементирати једноставну корисничку спрегу (7).

Канал се састоји из три нивоа, како би корисничка спрега била максимално упрошћена и минимално зависна од коришћеног транспортног протокола као што је приказано на слици 23. Због сложености приказа, дијаграмом су приказане најважније класе, методе и поља. За приватна поља су имплементиране све добављачке методе, међутим, оне нису представљене дијаграмом. На најнижем слоју се налазе имплементације подржаних протокола за размену великих података, а то су: *UDP, TCP, RTP* и *RTSP*. Ниво изнад ових имплементација је прилагодни слој. Прилагодни слој је имплементиран тако да су основне функционалности (слање, пријем и слично) садржане у надкласи коју наслеђују класе у којима су ове функционалности имплементиране и мапиране на одговарајући протокол за пренос. Највиши слој дефинише спрегу са корисником, односно начин на који корисник користи канал за велике податке. Управљачка компонента за слање, пријем и распоређивање саобраћаја пролази вертикално кроз читав програмски стек за слање великих података.

Поред управљачке компоненте, компоненте помоћних услуга у које спадају серијализација (десеријализација), сегментација (унификација) и шифровање (дешифровање) података су такође спрегнуте вертикално са читавим програмским стеком. Сегментација је имплементирана како би се избегла подразумевана фрагментација на нивоу интернет протокола. У једначинама 2 и 3 за одређивање величине сегмента за пренос корисних података размотрени су различити фактори: величина *Ethernet* оквира (F_{eth}), *Ethernet* заглавља (H_{eth}) и поља за контролни збир (N_{fcs}) (1), величина заглавља интернет (H_{ip}), *UDP, TCP* и *RTP* протокола (H_{tp}) (2), величина заглавља додатог од стране овог канала (H_{dca}) како би се омогућила правилна синхронизација сегмената (3), и утицај шифровања на величину података (N_{enc}) (4). У случају коришћења *RTP* протокола заглавље канала није додато јер је синхронизација података овим протоколом већ дефинисана. Стога је дата једначина 2 за рачунање броја бајтова корисног податка сегмента трансмисије путем *UDP* и *TCP*, а једначина 3 путем *RTP* протокола.



Слика 23: Архитектура канала за пренос великих података



Слика 24: Заглавље додато каналом за пренос великих података

$$N_p = F_{eth} - H_{eth} - N_{fcs} - H_{ip} - H_{tp} - H_{dca} - N_{enc} \tag{2}$$

$$N_p = F_{eth} - H_{eth} - N_{fcs} - H_{ip} - H_{tp} - N_{enc} \tag{3}$$

Треутно је подржана дужина стандардног *Ethernet* оквира од 1518 бајтова дефинисаног *IEEE 802.3* стандардом [195], чије заглавље износи 14 бајтова, а планирано је проширење решења првобитно подржавајући *IEEE 802.1Q* стандард [196], а потом и *Jumbo Ethernet* оквире који нису покривени стандардом већ их имплементира произвођач. Заглавље интернет протокола износи 20 бајтова за типичну дужину подржане *IPv4* верзије, а као проширење се планира омогућавање енкапсулације путем *IPv6* пакета. Дужина *UDP* заглавља износи 8, а за *TCP* је 20 бајтова. Заглавље канала износи 8 бајтова и састоји се од ознаке за број секвенце односно сегмента, укупног броја сегмената на које је растављен податак, дужине корисног податка и контролних ознака за први и последњи сегмент као што је представљено на слици 24. У случају преноса података путем *RTP* протокола, не користи се злаглавље канала, већ заглавље дефинисано овим протоколом дужине 16 бајтова. Утврђено је да се шифровањем додаје укупно 28 бајтова на дужину податка који је потребно шифровати.

Циљ имплементираног шифровања јесте да уведе сигурносни механизам у овај канал. Како би сегментација и шифровање били омогућени, најпре је потребно урадити серијализацију података како би подаци били правилно представљени као обичан низ октета у меморији. За серијализацију је коришћена *boost* библиотека, док је за потребе шифровања коришћена *OpenSSL* библиотека. За шифровање је коришћен *AES-GCM* алгоритам са дужином кључа од 128 бита.

Један програм у себи може имати више клијентских и/или серверских компоненти. Свака корисничка апликација, тј. програм, било да је у питању клијентска или серверска страна, позива спрегу овим решењем датог канала. Направљене клијентске или серверске компоненте на највишем нивоу садрже листу учесника са којима комуницирају, која се ажурира током читавог циклуса рада и користе заједничку *TrafficManager* компоненту за управљање саобраћајем.

Корисничка апликација на серверској страни треба прво да креира инстанцу сервера

(*DataChannelServer*), а затим ток података над објектом сервера позивајући функцију *int startStream (StreamProtocol streamTP, StreamServer* streamServer, RtspOptionsInfo* roInfo = nullptr)* која за први параметар прима ознаку набројиве класе транспортног протокола који ће бити коришћен, а као други параметар објекат сервера која садржи адресу интернет протокола и спрежни пролаз. Уколико је коришћен *RTP* потребна је и учесталост смењивања кадрова, тј. оквира (енгл. *Frame rate*), односно за *RTSP* је потребно попунити структуру *RtspOptionsInfo* која садржи информације о опцијама преноса које сервер може да понуди. Две ствари су кључне за стварање тока података: конструисање мрежних утичница које ће сервер користити за тај ток података и остварање везе са компонентом која управља саобраћајем (*TrafficManager*). конструисање утичнице је извршено позивима *int createSocket()* методе из слоја протокола која је прозвана кроз *int createServerSocket (SocketData* SockData)* методу одговарајуће *Binding* компоненте прилагодног слој.

Са друге стране, са компонентом за управљање саобраћајем је спрега непосредна, а објекат ове класе је конструисан кад и прва инстанца клијента или сервера у програму. Након што је направљена утичница при иницијализацији тока података, на серверској страни позвана је метода *void turnOnHandler (ListenerData LD, std::function<void (int))* угњеждене класе *StreamHandler* компоненте *TrafficManager* чиме је серверска утичница регистрована у вектору *std::vector<int> listeningSockets* , који представља низ свих утичница које је потребно надгледати. Поред утичнице, регистрована је и метода повратне спреге која ће бити прозвана када на утичници буде било саобраћаја. *TrafficManager* компонента врши константно ослушкивање у нити *mThread* и проверава стање на утичницама методом *int checkForRead (int sockFd)* угњеждене класе *SocketHandler*. Када на утичници буде примећен саобраћај, позива се *void receiveStreamBuffer (int socketFd)* повратна метода имплементирана у серверској компоненти, која даље повлачи читање пристиглог податка функцијама у имплементацији протокола (о читавој овој путањи ће бити више детаља у наредним пасусима када буде описиван рад клијентске компоненте). Серверска компонента проверава да ли је примљени податак уствари порука *"CONNECT REQUEST"*, односно да ли је *"DISCONNECT REQUEST"*. Уколико јесте, долази до позива метода које служе за управљање листом ентитета клијената којима је потребно слати податке, додајући их методом *bool addConnectedClient (int streamID, std::shared_ptr<ConnectedClient> connectedClient)*, односно избацујући их методом *bool removeConnectedClient (int streamID, std::shared_ptr<ConnectedClient> connectedClient)* респективно. Ове методе су изложене кориснику и могу, такође, бити позване из самог програма.

Повратна вредност методе за стварање тока указује да ли је инстанца тока података направљена или није. Уколико није - биће негативна, уколико јесте - вредност је једнака идентификатору утичнице и представља идентификатор тока података за слање података јер једна серверска апликација може садржати више активних токова података на које шаље исте или различите податке.

Након што је иницијализација завршена, потребно је да серверски програм позове шаблонску методу *template<typename dataType> bool sendData (int streamID, dataType* buffer, size_t bufferSize)*. Ова функција најпре даље прозива серијализацију методом *bool serialize (dataType* data, size_t size)* из компоненте за обраду података *DataProcessor* која даље

позива *void writeByte (dataType& data)* методу шаблонске угњеждене класе *Serializer*. У овој методи, позвани су оператори за серијализацију ">>"из *boost* библиотеке којом се прослеђени тип података серијализује у низ карактера спреман за даље слање. Затим следи шифровање позивом методе *bool encrypt (EVP_CIPHER_CTX* ctx, EVP_CIPHER* cipher, void* ptext, int32_t plen, void* ctext, int32_t* clen, uint8_t* iv, uint8_t* tag)* која даље у компоненти за шифровање *Encryptor* прозива методе *bool Init(), bool Update(), bool getParams()* из *OpenSSL* библиотеке. Прва и друга метода служе за шифровање података за слање, а трећа за додавање *MAC* ознаке. Након шифровања података, следи позивање методе *bool segment (uint8_t* data, uint16_t size)* која позива истоимену методу *void segment (byte_t* buffer, size_t bufferSize)* из (*Segmenter*) компоненте која извршава сегментацију. *Segmenter* даље креира *std::map<int, std::vector<uint8_t>>* мапу парова где је вредност вектор података за слање на чијем је почетку додато заглавље методом *bool addHeader (Header& head)*, а кључ његов редни број. Након што су подаци овако припремљени, из *DataChannelServer* компоненте је за сваког од клијената из листе позвана метода *ssize_t send(void* buffer, unsigned bufSize)* над показивачем на објекат прилагодног слоја *Binding* који чува одговарајући објекат изведене класе прилагодног слоја за специфичан протокол који је одабран.

Прилагодни слој позива одговарајућу методу за слање из транспортног слоја у зависности од тога који је протокол коришћен чиме се подаци отпремају на мрежну утичницу (енгл. *Network socket*).

Са друге стране, клијентска апликација креира инстанцу клијентске компоненте (*DataChannelClient*) и над њом позива шаблонску функцију *template<typename dataType> int receiveStream (StreamProtocol streamTP, StreamClient* streamClient, StreamServer* streamServer, std::function<void (dataType*, size_t)> handlerFunction, RtspIface* RtspIface = nullptr)* за стварање пријемног тога података. При позиву ове функције, од клијентске апликације је захтевано да зада тип пријемног податка који одговара типу који је иницијално послат са сервера како би десеријализација била успешна. Први параметар функције прима ознаку транспортног протокола који ће бити коришћен, други параметар је објекат клијента који садржи адресу интернет протокола и порт, трећи је објекат сервера, а четврти функција повратног позива (енгл. *Callback*) која ће бити прозвана сваки пут кад податак стигне са сервера на мрежну утичницу. Пети параметар није захтеван и он садржи функцију повратне спреге која представља спрегу којом ће *RTSP* клијент моћи да добије податке из одговора *RTSP* сервера и кроз њега пошаље нови захтев потребан за успешну успоставу сесије размене података. Као у случају серверске апликација, при стварању инстанце тока података на клијентској страни је на исти начин најпре конструисана мрежна утичница, а потом спрега са компонентом за управљање саобраћајем (*TrafficManager*) и уколико је функција успешно извршена повратна вредност представља идентификатор тока, тј. идентификатор активне мрежне утичнице.

Као што је већ речено, *mThread* нит управљачке компоненте током читавог рада проверава стање на мрежним утичницама свих регистрованих клијената. Када поменута *checkForRead* ослушкивачка метода региструје да податак постоји, започиње пропагацију, али овај пут до клијентске компоненте, позива најпре повратни метод *void*

receiveStreamBuffer (int socketFd). Овај метод помоћу идентификатора утичнице прослеђеног као параметар позива, кроз *Binding** инстанцу над конкретном компонентом прилагодног слоја позива методу *ssize_t receive(void* buffer, unsigned bufSize)* која даље позива одговарајућу функцију за читање са мрежне утичнице - *ssize_t receive(int clientSocket, void* buffer, unsigned bufSize)* у случају *TCP* тј. *SOCK_STREAM* утичница, односно *ssize_t recvFrom(sockaddr_in& toAddress, void* buffer, unsigned bufSize)* у случају *UDP* тј. *SOCK_DGRAM* утичница. Унутар ових најнижих метода програмског стека се позивају *recv()* и *recvfrom()* системске функције респективно.

Када је податак прочитан са мрежне утичнице помоћу компоненте за обраду података (*DataProcessor*) следи извршавање унификације добијених сегмената методом *bool unify (int streamID, Header& header, uint8_t* seg, uint16_t size)*. Ова метода позива даље позива методе угњеждене класе *Unifier* у којој се врши унификација. Најпре метода *bool removeHeader (int streamID, Header& head, uint8_t* segment)* уклања заглавље и издваја корисне податке из њега. На основу ових података подешене су вредности у мапи *std::map<int, vector<uint8_t>> data* која складишти све примљене податке. Кључ мапе представља редни број читавог примљеног податка (у случају преношења видео поруке то би био редни број оквира у читавој видео поруци), а вредност је вектор података који чине читав оквир (у случају преноса путем *RTP* протокола мапа има само један пар у чијем вектору се налазе подаци од више целина јер само *RTP* заглавље не садржи податке о редном броју целине, већ искључиво о секвенци сегмента). С обзиром на то да је величина података за слање унапред непозната, са сваким пристиглим сегментом се мапа мења. Податак о редном броју из заглавља детерминише кључ под којим ће бити промењена вредност, а секвенца пристиглог сегмента у читавом податку одређује нову величину вектора под одговарајућим кључем. Величина вектора се задаје на основу целобројног умношка секвенце и максималне величине корисног податка добијене претходно приказаним једначинама. На овај начин ће остати вишак заузете меморије, тј. меморије која није искоришћена, али гарантована је целовитост података. Метода *bool storeSegment()* смешта податке у одговарајући део вектора лоцираног на основу података из заглавља. Када пристигне сегмент који носи информацију о завршавању података једне целине, метода *bool removeDamaged()* проверава да ли је читав вектор попуњен. Уколико није, биће одмах обрисан из мапе. Биће обрисани, уколико постоје у мапи сви парови са кључевима мањим од тренутног за који је пристигао последњи сегмент. У случају *RTP* протокола, разамтрана је и временска ознака коју је носио први сегмент целине податка у свом заглављу. Ова ознака је упоређена са временом пристизања последњег сегмента и уколико је њихова разлика већа од задате фревенције слања податак ће, такође, бити одбачен.

Уколико је податак успешно састављен позвано је дешифровање читавог податка методом *bool decrypt (EVP_CIPHER_CTX* ctx, EVP_CIPHER* cipher, void* ctext, int32_t clen, uint8_t* iv, uint8_t* tag, void* ptext, int32_t* plen)* на исти начин као што је то случај у шифровању, узимући за ctext адресу почетног елемента вектора састављених података. Дешифровање врши компонента *Decryptor* опет користећи поменуте три методе из *OpenSSL* библиотеке. Након дешифровања следи десеријализација методом *bool deserialize (uint8_t* data, size_t size)* шаблонске класе *Deserializer* оператором «<" *boost* библиотеке

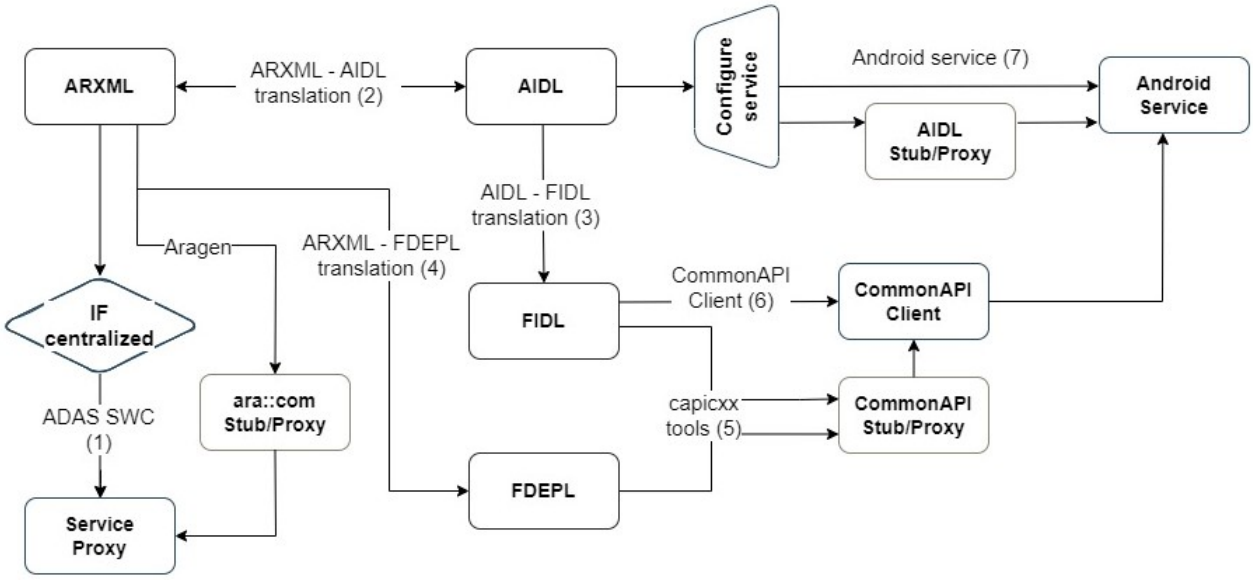
у тип података који је клијент навео при конструисању објекта иснтанце тока података. Када је податак спреман биће позвана и сама повратна функција коју имплементира корисничка апликација, тј. програм. С обзиром на то да се податак који је пристигао налази у меморији коју контролишу компоненте наше програмске спреге, неопходно је одмах по приспећу преузети податак у меморијски блок којим управља сама корисничка апликације. Податак ће бити допступан докле год путем исте утичнице не пристигне нови који га преписује. Уколико се у уређају у било ком моменту деси да нема меморије за заузимање у сврху пристизања нових података или у току припремања података за слање или за прослеђивање кориснику, комплетан податак ће бити одбачен, докле год се алокације меморије не изврши успешно.

5 Аутоматско генерисање средњег слоја комуникације

Описана потреба за одвајањем имплементације апликација на врху програмске подршке од нижих модула и платформских сервиса реализована је путем дефинисања стандардизоване програмске спреге. Комуникациони стек је управо најбољи пример стандардизоване програмске спреге. Корак даље у дефинисању комуникационе спреге је могућност њеног описивања путем модела направљених у неком од језика за дефинисање спреге (*ARXML*, *FIDL* и *AIDL*) описаним у теоријским основама. На овај начин, корисницима је омогућено да сами дефинишу комуникационе спреге међу компонентама, а да се из описаних модела изгенеришу делови средњег слоја комуникације који омогућавају да се путем позива дефинисане спреге изврши комуникација.

Из ових разлога, идеја аутоматизације овог решења долази из потребе да се омогући аутоматско генерисање читаве програмске подршке за размену података на основу комуникационих спрега дефинисаних у почетним *ARXML* моделима. Смер генерисања може бити и супротан, тј. могуће је за почетну тачку користити и *AIDL* модел уз који би било неопходно описати *SOME/IP* податке у *.fdepl* датотеци, али како је већ речено тај смер није представљен радом. С обзиром на то да прогамска подршка за пренос токова великих података има непроменљиву спрегу и као таква се непосредно укључује у процесе који учествују у комуникацији и која не зависи ни од једног језика за дефинисање спреге, није потребно генерисати овај део решења. Стога се аутоматизација своди искључиво на генерисање подршке за сервисно-оријентисану комуникацију путем *SOME/IP* протокола.

Процес аутоматског генерисања дат је на слици 25. Оваква поставка циљаног процеса решења се даље разлаже на захтеве који су специфично везани за сваку од фаза генерисања и за које је дефинисана архитектура решења за превођење и тиме реализацију сваке фазе као што ће бити приказано у наставку. Након тога, из сваке појединачне архитектуре су издвојене целине и имплементационе јединице (функције) које је потребно имплементирати како би захтеви били испуњени. Поред слике 25, табела 6 представља мапирање основних ентитета кроз различите језике над којим је базирано решење приказано



Слика 25: Фазе генерисања решења

Табела 6: Мапирање основних елемената *ARXML*, *FIDL* и *AIDL* модела

ARXML	**FIDL**	**AIDL**
ARPackage	FidlPackage	AidlPackage
ServiceInterface	ServiceInterface	AidlInterface
ClientServer Method	Method	AidlMethod
Argument	Argument	Argument
Field:Setter	Attribute: Write-only	3x AidlMethod и 2x AidlInterface
Field:Getter	Attribute: Read-only	AidlMethod
Field:Notifier	Attribute: Notifier	3x AidlMethod и 2x AidlInterface
Event	Broadcast	3x AidlMethod и 2x AidlInterface
Primitive types	Primitive types	Primitive types
Consts	Consts	Consts
Structure	Structure	Parcelable
Enum	Enum	Enum

у наставку по фазама.

На почетку је неопходно и дефинисати зависности од других решења и предуслове које је потребно испунити за саму реализацију. Да би генерисање било започето, неопходно је имати описани модел *ADAS* система, односно компонената које постоје у овом домену. Додатно, у модел сваке од компонената неопходно је убацити дефиницију комуникационе спреге путем које ће бити обављена размена жељених података. Уколико се користи централизовани приступ, односно централизовано решење, неопходно је додати и модел којим се описује *ServiceProxy* компонента. Поред тога, уколико је одабрано коришћење централизованог решења неопходно је задати и датотеку формата *.json* у којој треба описати мапирање излазних комуникационих спрега *ServiceProxy* компоненте са циљним комуникационим спрегама компоненти чији се подаци преносе. Након што су дефинисане почетне тачке, могуће је извршити генерисање. Ови поступци су нешто што мора бити извршено

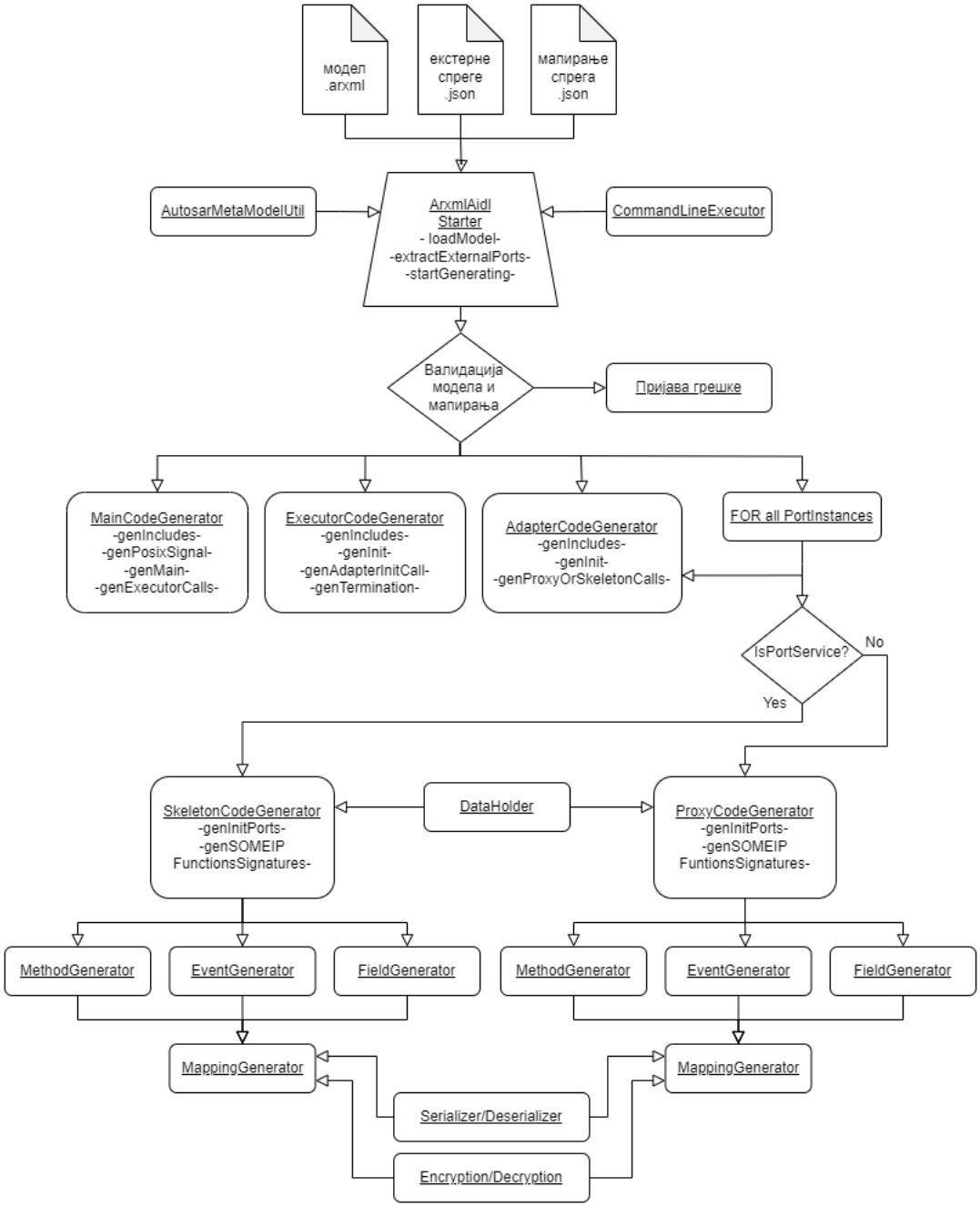
од стране истраживачких и инжењерских центара произвођача аутомобила, јер су једино они ти који могу управљати садржајем компонената у овако енкапсулираним системима као што је *ADAS* и доношењем одлука који подаци могу бити изложени осталим доменима. Са друге стране, комуникационе спреге на *IVI* страни су једноставније за управљање јер је, налик на ситуацију у потрошачкој електроници, тежња усмерена ка томе да и апликације у *IVI* домену буду јавно доступне и да могу бити инсталиране као део програмске подршке.

Један од корака у остваривању аутоматски генерисане програмске подршке за пренос података који сваки произвођач аутомобила изводи на свој начин пратећи *AUTOSAR* стандард подразумева између осталог и генерисање модула средњег слоја за комуникацију (*ara::com*) тако да омогући размену података путем свих спрега за комуникацију дефинисаних *ARXML* моделом. Различите компаније (*Vector*, *Elektrobit*, итд) нуде и сертификују ове услуге, пружајући гаранцију за добијене функционалности, и одговарају на захтеве следећи стандардизацију описану *AUTOSAR* документацијом. За потребе овог решења, коришћен је *aragen* алат дат у оквиру демонстраторског пакета читавог Adaptive *AUTOSAR* стека, који даје пример тога како модули треба да буду имплементирани да би били у складу са стандардом. Након што је *aragen* пружио генерисане делове модула средњег слоја и омогућио коришћење дефинисаних комуникационих спрега, потребно је у циљаним програмским компонентама *ADAS* домена ручно имплементирати позиве који управљају слањем података, док у случају централизованог решења, започиње и генерисање *ServiceProxy* компоненте.

5.0.1 I фаза - генерисање *ServiceProxy* компоненте

Целокупно генерисање, па тако и генерисање *ServiceProxy* компоненте, извршено је коришћењем развоја вођеног моделима и *Eclipse Modelling Framework - EMF* оквира. *ARXML* мета-модел, неопходан за овакав начин генерисања, преузет је од artop организације и коришћен у овом истраживању. Архитектура генератора прве фазе је дата на слици 26. Делови *ARXML* модела од значаја су они који садрже информације за описивање комуникационих спрега у компоненти и сав генерисани програмски код настаје од њих. Обавезни улазни аргументи за генератор ове фазе су *.arxml* датотеке које садрже информације о компоненти која прослеђује податке и *.json* датотеке у којима је наведено које су спољашње спреге, тј. спреге од интереса за генератор, као и мапирања унутрашњих на спољашње спрежне пролазе, док је назив саме компоненте опциони улаз (подразумевани назив је *ServiceProxy*).

Из архитектуре се може видети како је најпре у покретачкој компоненти ове фазе извршено учитавање модела *ServiceProxy* компоненте из *.arxml* датотека, на основу претраге по називу. При учитавању конструисано је неколико листа од значаја: листа за пријемне спрежне пролазе, листа за пошиљалачке спрежне пролазе и листа параметара за сваку конструисану инстанцу спрежног пролаза. Поред тога, извршено је учитавање мапирања из *.json* датотеке и конструисани су парови спрежних пролаза путем којих се подаци прослеђују на *IVI* домен. При учитавању мапирања се врши и валидација: да ли заиста наведени спрежни пролази за мапирање постоје, да ли су у складу са могућим сцена-



Слика 26: Архитектура генерисања посредничке компоненте на *ADAS* страни

ријима мапирања, да ли типови података одговарају. Уколико су сигурносни механизми активирани, у валидацији се проверава и да ли је тип сваког податак који се размењује са *IVI* страном дефинисана сигурносна структура. Затим, након покретачке компоненте следи извршавање генераторских компонената за генерисање програмског кода за сваку појединачну датотеку. Генерисање програмског кода представља стварање низа карактера који се на крају чува у излазну датотеку, што представља описани модел-у-текст принцип.

Најпре се у целости ствара датотека у којој је *main()* функција и датотеке заглавља и изворног кода за извршилачку и адаптер компоненту. Називи настају додавањем суфикса *"Executor"* и *"Adapter"* на назив посредничке компоненте из модела. Потом се за сваку инстанцу спољашњег пролаза у адаптер компоненти генерише изворни код за ини-

цијализацију компоненте која имплементира сервисни, односно клијентски део, као и саме сервисне и клијентске позиве. Називи датотека заглавља и изворног кода за сваку од ових компонената појединачно се добијају тако што се на назив спрежног пролаза дода суфикс *"skeletonImpl"* за сервисну компоненту, док је у случају клијентске компоненте овај наставак *"proxyImpl"*.

Програмски код главних компонената које учествују у размени података се генерише из три дела. Најпре се, у првом делу, генерише код пролазећи кроз све *SOME/IP* механизме спреге за коју се генерише компонента. У зависности од тога да ли је инстанца спреге за коју се генерише код пријемна или пошиљалачка страна, генерише се употреба различитих објеката за остваривање комуникације из *ara::com* модула (*Proxy* или *Skeleton*). Још једном је корисно напоменути да је у случају спољашње спреге која се понаша као пријемна страна тј. клијент и унутрашње спреге која се понаша као пошиљалац тј. сервис дозвољена употреба искључиво поља за постављање вредности. Тако се за унутрашњу спрегу и долазне догађаје, поља за обавештавање и поља за постављање вредности стварају функције повратне спреге које се прозивају у *ServiceProxy* компоненти сваки пут када друге *ADAS* компонете шаљу податке. У тим функцијама се генерише и код за преузимање и чување пристиглих података. За позиве добављања података на захтев путем метода и поља за добављање вредности у унутрашњој комуникацији се генеришу искључиво позиви одговарајућих метода којима се на тај начин податак и добавља. Са друге стране, за механизме обраде захтева пристиглих са *IVI* стране се за генерисање припремају методе, док се за отпремање одлазних догађаја, поља за обавештавање и поља за постављање вредности генеришу позиви одговарајућих функција. У овако имплементираном првом делу генерисања клијентских и сервисних компонената се изгенерисани програмски код још увек не чува у датотеку.

Следећи корак, тј. други део стварања компонената одговорних за размену података је генерисање мапирања. Генерисање мапирања представља уметање низа карактера који повезује пријемне и пошиљалачке спрежне пролазе на дефинисани начин, као што је описано у потпоглављу 4.1.1. Тако се за случај обраде захтева који пристижу са *IVI* стране на основу мапирања умеће генерисани позив унутрашње методе или коришћење већ сачуваних података који су путем неког од обавештавајућих механизама дошли у *ServiceProxy* компоненту. Такође, уколико подаци долазе и одлазе путем неког од обавештавајућих механизама, након преузимања и чувања података са унутрашњих инстанци се умећу генерисани позиви спољашњих догађаја, поља за обавештавање или постављање вредности. Кад је генерисање мапирања завршено, још увек се припремљени низови карактера, тј. програмски код не генерише у датотеку јер следи трећи део.

У трећем делу се проверава да ли су сигурносни механизми активирани. Уколико јесу, у низ карактера умеће се и њихова имплементација. Ово подразмева првобитно генерсање низа карактера којим се врши серијализација користећи *boost* библиотеку и шифровање у случају слања података, односно дешифровање и десеријализација за пристигле податке тако да се типови података уклопе са оним који су дефинисани моделом. Након што је уметање сигурносних механизама извршено (или прескочено уколио се не захтева његово извршавање) генерисани програмски код бива сачуван у одговарајуће датотеке.

У случају дистрибуираног приступа ова фаза се прескаче.

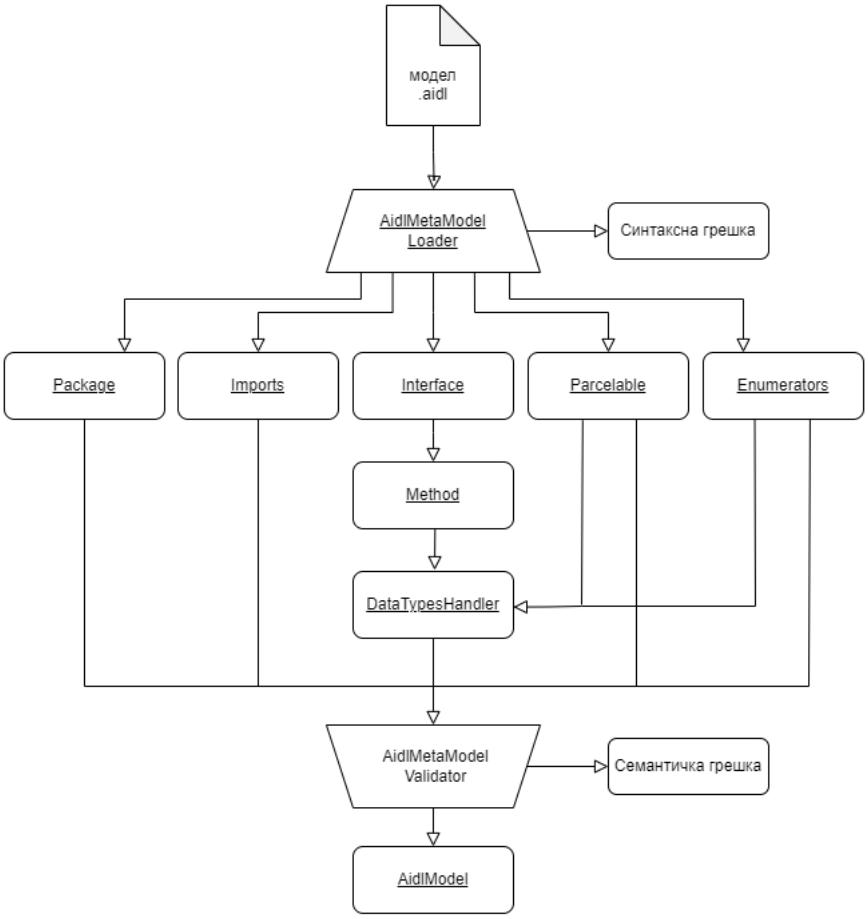
5.0.2 II фаза - превођење између *ARXML* и *AIDL* модела

Друга фаза је превођење из *ARXML* у *AIDL* модел принципима модел-у-модел превођења. Овај генератор је имплементиран на јединствен начин, без обзира на то да ли је реч о дистрибуираном или централизованом приступу. Да би било могуће остварити генерисање у овој фази, предуслов је био имплементирати *AIDL* мета-модел.

5.0.2.1 *AIDL* мета-модел Како би превођење било извршено по модел-у-модел принципу, неопходно је имати имплементиране мета-моделе за оба језика. *ARXML* мета-модел је већ написан у оквиру artop решења и слободан за коришћење студентима у истраживачке сврхе. Верзије шема које подржава описани мета-модел су 00049. Са друге стране, за *AIDL* језик не постоји доступан мета-модел. Зато је он имплементиран коришћењем *Xtext* радног оквира. *Xtext* омогућава да се кроз једну синтаксу дефинишу правила и услови у којима се ентитети могу појављивати као и њихову правилну лексику. Мета-модел је писан по принципу лабава граматика, стриктна валидација (енгл. *loose grammar, strict validation*). То значајно утиче на број потребних правила за описивање граматике. На пример, по правилима *AIDL* језика различити типови података смеју, односно не смеју, да се нађу на одређеним позицијама - повратна вредност може да буде било који тип подржан *AIDL* језиком; аргументи и поља унутар *Parcelable* не смеју бити типа void; тип константи може да буде само означен цео број или низ знакова. На овај начин, потребно би било описати 4 различита правила у граматици. Међутим, одлучено је да у граматици постоји само једно правило које проверава да ли је тип уопште подржан и препознат од стране *AIDL* језика, а у валидацији ће бити пријављена грешка за тип који семантички не може да стоји на одређеном месту. Овакав приступ чини решење лакше прилагодљивим за будуће измене. Од *Xtext* описа граматике *EMF* радни оквир формира мета-модел. Улога *EMF* радног оквира је значајна, од *.xtext* датотеке за опис граматике величине 8KB настаје 468KB кода за класе контејнера, 376KB кода за класе парсера, 36KB кода за класе серијализатора и 312KB кода за услужне класе што је укупно 1192KB генерисаног програмског кода.

Дијаграм архитектуре *AIDL* мета-модела је дат на слици 27. Граматиком описаном мета-моделом на почетку се извршава учитавање пакета и свих дефинисаних зависности које се укључују. Затим се учитавају дефинисане спреге, што резултује учитавањем свих метода које су дефинисане овом спрегом и типова података који се у њима користе. Такође, истоветно се учитавају и ентитети *Parcelable* и набројивих типова, најпре њихови називи, а затим и типови података који су у њима дефинисани. Учитавање информација о сваком поменутом ентитету се извршава за сваки присутан елемент који одговара описима ових ентитета. Дакле, у тренутку учитавања се још увек не врши валидација односа ових ентитета унутар читавог модела.

Поред граматике написан је и валидатор којим се проверава и семантички смисао сваког сегмента из граматике *AIDL* модела. Овде се проверава: да *AIDL* има дефинисан назив пакета (1), проверава се да ли постоје друге дефинисане зависности које се укључују



Слика 27: Архитектура учитавања *AIDL* модела у мета-модел

путем import директиве (2) - уколико постоје потребно их је спрегнути, да ли је дефинисан спрежни пролаз, набројиви тип или тип *Parcelable* (3), уколико је дефинисан спрежни пролаз - да ли је то једини спрежни пролаз дефинисан у датотеци и да ли је поред њега дефинсан тип *Parcelable* или набројиви тип (4), уколико је дефинисан тип *Parcelable* - да ли је то једини тип *Parcelable* дефинисан у датотеци и да ли је поред њега дефинсан спрежни пролаз или набројиви тип (5), уколико је набројиви тип - да ли је то једини набројиви тип дефинисан у датотеци и да ли је поред њега дефинсан спрежни пролаз или тип *Parcelable* (6), да ли се у овим дефинисаним зависностима заиста налази оно што се користи, а није дефинисано у самој датотеци (7). Након што валидатор прође без пријављене грешке, може се сматрати да је модел из *.aidl* датотека учитан и све информације су садржане у одговарајућим објектима који представљају модел.

5.0.2.2 Превођење из *ARXML* модела у *AIDL* модел Када постоји *AIDL* мета-модел, могуће је реализовати и превођење *ARXML* модела у *AIDL*. Дијаграм архитектуре је дат на слици 28. Превођење почиње учитавањем *.arxml* датотека. Датотеке се учитавају коришћењем програмске спреге коју пружа *EMF* у оквиру које се извршава парсирање и интерпретација учитаних симбола. Од овако учитаног модела се конструишу објекти који садрже податке описане мета-моделом и који се користе за каснију обраду. Уколико провера за учитани *ARXML* модел укаже на то да је модел празан или да нема податке о

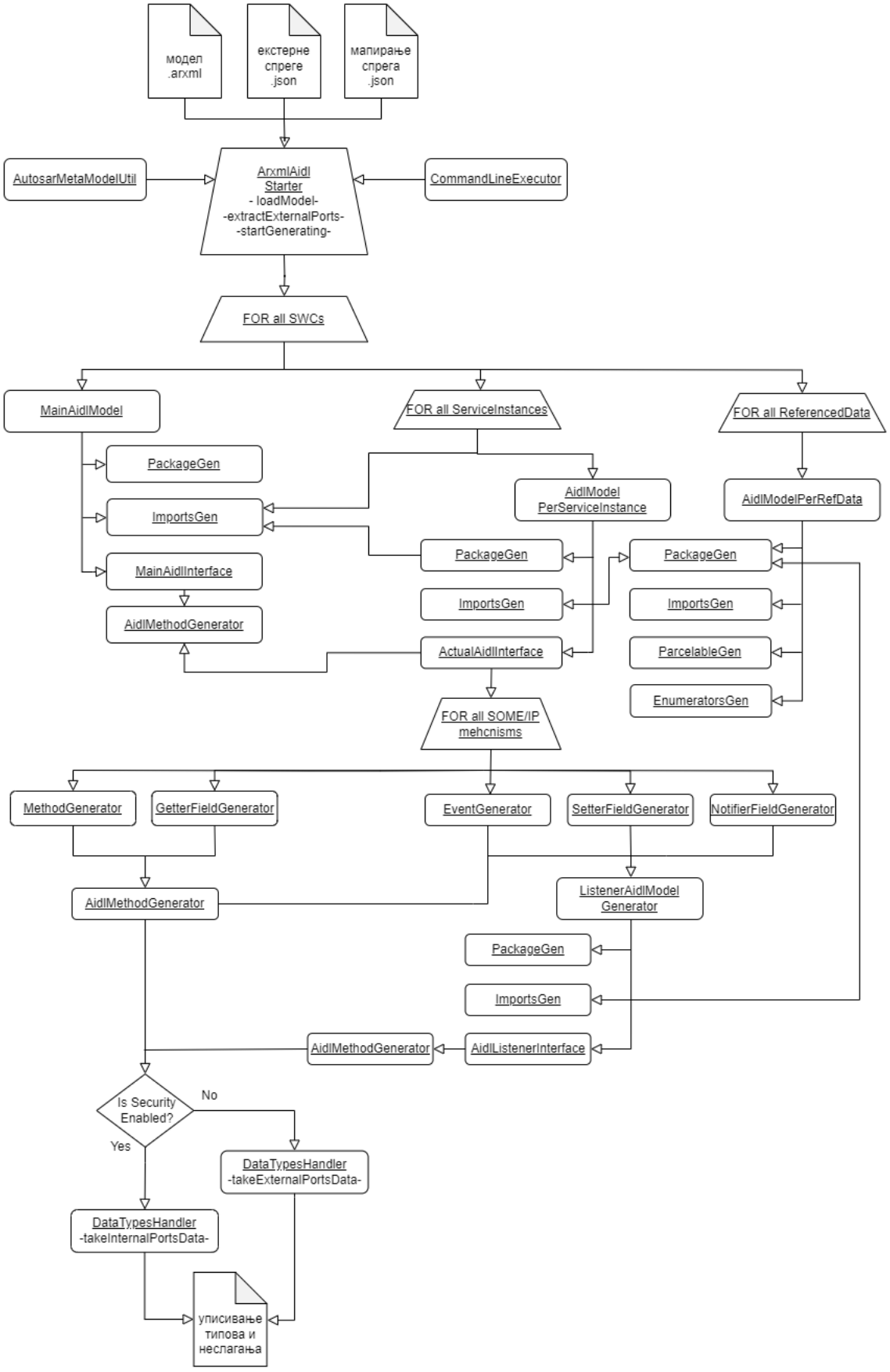
спрежним пролазима, превођење неће започети.

Превођење почиње тако што се из учитаног *ARXML* модела, издвајају и чувају само спрежни пролази компонената који треба да воде ка *IVI* страни. Ово је омогућено помоћним описима датим у датотекама формата *json*. Ово је неопходно како би се омогућило ефикасно генерисање које узима у обзир само спреге од интереса. Уколико не би постојали ови додаци којима се генерише комуникација за жељене спреге, то би било могуће потенцијално урадити на друга два начина. Први је увођење конвенције именовања, што отежава примену решења на постојеће системе. Друго решење се ослања на узимање искључиво пошиљалачких спрежних пролаза за све *SOME/IP* механизме сем за поље за постављање вредности. Код поља за постављање вредности се разматра само пријемни спрежни пролаз. У оба случаја би се гледали само они пролази чије инстанце немају пар са којим су већ спрегнути у *ARXML* моделу јер то значи није направљена да спрега ни са ким у систему). Овакав начин филтрирања спрежних пролаза, иако представља луцидније приступ, није максимално ефикасан, из разлога што може доћи до стварања вишкова. Такође, у централизованом приступу уколико постоји више могућих парова спрега које преносе исте типове података мапирање би било додатно отежано због непознавања контекста информација које спреге преносе.

Након што је модел учитан, над циљаним спољашњим спрежним пролазима се врши превођење при чему се стварају објекти од претходно описаних класа *AIDL* мета-модела. Тако се на почетку, за сваку компоненту програмске подршке која учествује у комуникацији из *ARXML* модела ствара један главни спрежни пролаз (било да је у питању *ServiceProxy* у случају централизованог приступа или неке друге компоненте када је реч о дистрибуираном приступу). За овај *AIDL* модел се конструишу називи пакета на основу назива пакета у ком се налазе програмске компоненте из *ARXML* модела и називи спрега на основу назива одговарајуће компоненте. Након тога се за сваку инстанцу спрежног пролаза стварају три ствари. Прво методе које враћају објекте других спрега из укључених пакета. Затим *AIDL* модел који садржи спрегу насталу од одговарајуће сервисне инстанце из *ARXML* модела и на крају се генерише укључивање (import) ових спрега у главни *AIDL* модел.

Са обзиром на то да у *ARXML* моделу може постојати више инстанци сервиса, тј. спрежних пролаза направљених од једне спреге није довољно дефинисати спреге у *AIDL* моделима искључиво на основу спрега из *ARXML* модела. Зато се свака *AIDL* спрега појединачно преводи на основу одговарајуће инстанце сервиса, тј. спрежног пролаза из *ARXML* модела која је додељена тој специфичној компоненти за коју се трансфомрација врши. Сам назив *AIDL* спреге одговара називу спреге из *ARXML* модела, али се за сваку од генерисаних *AIDL* спрега разликују пакети чији називи долазе од кованице назива пакета програмске компоненте, назива саме компоненте, пакета спреге, назива спреге и ознаке спрежног пролаза из *ARXML* модела. Неопходно је то урадити како би се могла направити дисјункција при коришћењу *CommonAPI* клијената, односно, како би крајњи корисник могао адекватно извршити одабир конкретне инстанце *ADAS* сервиса са којом жели да оствари размену података јер сваки од тих сериса могу настати од јединствено дефинисаног спрежног пролаза али могу преносити податке различитог контекста.

Дакле, када су направљене датотеке за сваку појединачну *AIDL* спрегу са одговара-



Слика 28: Архитектура превођења *ARXML* модела у *AIDL* модел

јућим пакетом у идентичном пару као што су коришћени у главној спрези, следеће што је потребно превести су механизми у оквиру самих спрега. Свака спрега треба да садржи бар један, а потенцијално и више *SOME/IP* механизама. За генерисање метода у *AIDL* спрегама сваки од *SOME/IP* механизама се преводи коришћењем специфичне компоненте имплементираног генератора за ову фазу. Тако су компоненте генератора раздвојене на компоненту за превођење догађаја, методе и поља за обавештавање, добављање и постављање вредности.

Механизми који добављају вредности на захтев су најлакши за превођење. За методу се прво проверава да ли је врста методе "пошаљи и заборави", односно метода која не чека на извршавање и не може да има излазне аргументе и повратну вредност. Уколико јесте, називу методе у *AIDL* спрези се придружује кључна реч *"oneway"*. Сам назив *AIDL* методе се добија из назива *SOME/IP* методе (или поља за добављање). Потом се за превођење самих типова и аргумената улази у посебну компоненту за руковање типовима. У случају када спрега садржи *SOME/IP* догађај, потребно је креирати две методе - за пријављивање и за одјављивање ослушкивача догађаја. Поред тога, неопходно је створити и додатни *AIDL* модел, односно *.aidl* датотеку која ће поседовати спрегу за стварање ослушкивача за тај специфичан догађај. У оквиру ове спреге биће генерисана и метода чији је аргумент управо податак који се преноси догађајем са *ADAS* стране. Овим је омогућен мезанизам за мапирање догађаја представљен у претходном поглављу дисертације. Још једном треба нагласити да се за превођење поља користе исти принципи као и за одговарајући догађај или метод, уколико је поље за добављање вредности пресликавају се принципи методе, а уколико је поље за обавештавање или постављање вредности пресликан је догађај. С обзиром на то да поља, методе и догађаји у *ARXML* моделу могу имати исти назив (биће разликовани придруженом ознаком у *XML* моделу), неопходно је на све методе додати одговарајући суфикс: *"_EVENT"*, *"_METHOD"* или *"_GETTER"*, *"_SETTER"* и *"_NOTIFIER"* како би се касније у превођењу и генерисању *FIDL* модела механизми могли бити разликовати, односно како би *CommonAPI* клијент био у потпуности усклађен са сервисом.

При превођењу у току наилажења на аргументе и на повратне вредности, типови података се такође преводе посебном компонентом. Већина основних типова је мапирана непсоредно из разлога што су подржани у оба језика задржавајући и назив током превођења. Главне разлике су у томе што у *AIDL* језику не постоје неозначени типови и структура података као таква. Неозначени типови се преводе у означене типове једног ранга изнад, док се за структуре позива нова компонента за превођење и стварање засебног *AIDL* модела који садржи *Parcelable* тип са називом који одговара структури. Да би *Parcelable* тип био правилно дефинисан, потребно је креирати нову датотеку која ће изгледати као типична *.aidl* датотека са називом структуре од које настаје. Такође, назив пакета унутар ове *.aidl* датотеке ће одговарати пакету структуре из *ARXML* модела. Може имати своје зависности - друге *Parcelable* променљиве (у чијем случају се све генерише рекурзивним позивима истих функција компонената за њихово превођење), али нема дефинисану спрегу. Уколико је потребно превести највећи неозначени тип, превођење ће кориснику приказати упозорење да може доћи до недефинисане промоције типова

и губитка података.

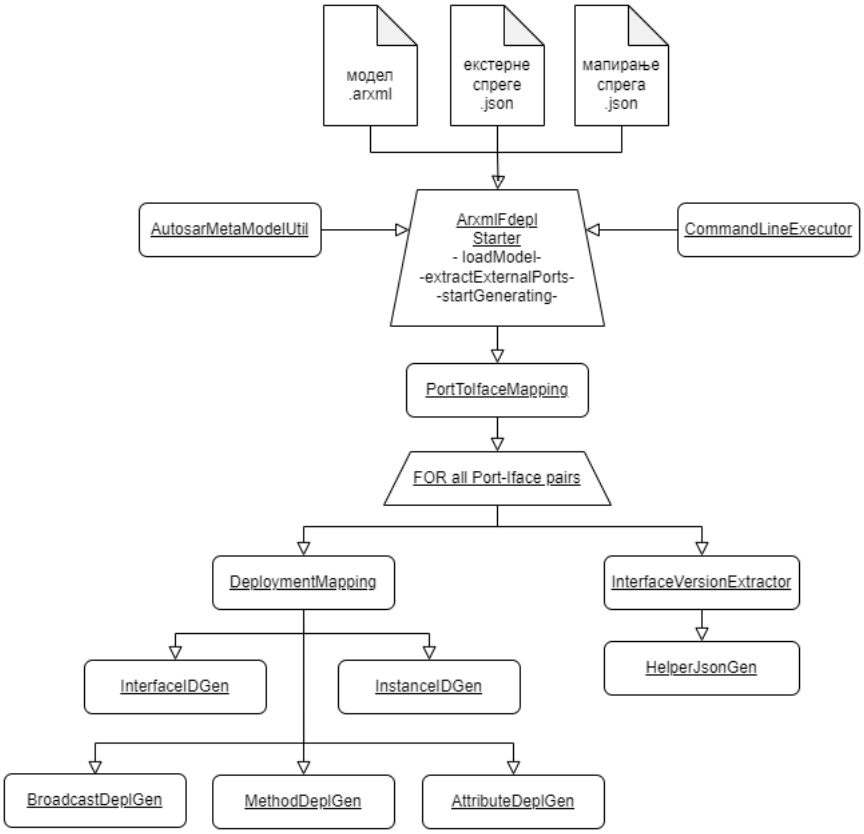
Два конфигурациона параметра такође имају значај у превођењу. То су активација сигурносних механизама и квалитета услуге. Уколико је квалитет услуге активиран, упозорења се промовишу у грешке и није допуштено извршити превођење типова чије коришћење неће бити у потпуности безбедно. Такође, није могуће имати *"oneway"* методе јер се њима не добија статус који потврђује њихово извршавање. Поред тога, активација сигурносних механизама подразумева коришћење сигурносног слоја у размени података при употреби *SOME/IP* механизама. Да би сигурносни механизми описани у потпоглављу 4.1.3 били обезбеђени, неопходно је да типови података у свим излазним спрежним пролазима буду идентичне структуре које се састоје од једног низа карактера и два вектора. За учитани *ARXML* модел ово не прави никакву разлику, сем додатног корака валидације који чини да превођење буде прекинуто уколико неки спољашњи спрежни пролаз који је дат у опису мапирања спрежних пролаза које је потребно изложити *IVI* страни користи тип података другачији од ове структуре. Међутим, с обзиром на то да су сигурносни механизми на *IVI* страни имплементирани у *CommonAPI* клијенту, *FIDL* модел је тај који податке размењује путем дефинисане структуре, док *AIDL* модел треба да садржи спреге које поседују изворне податке *ADAS* сервиса и да их као такве представи клијентима. Зато се, у овом случају, за превођење спрега у *AIDL* моделима користе подаци са спољашњих спрежних пролаза, али за типове података се узимају мапирани унутрашњи спрежни пролази.

Треба напоменути да се након превођења не добијају само *.aidl* датотеке, већ и помоћна датотека у којој су уписана неслагања типова и параметри спреге које треба пренети до *FIDL* модела.

5.0.3 III фаза - превођење између *ARXML* и *FDEPL*

Да би било могуће остварити комуникацију између инстанци сервиса и клијента преко *SOME/IP* протокола није довољно преводити само податке о спрези. Потребно је имати и усклађене параметре комуникације на обе стране. Параметри су, као што је објашњено у претходним поглављима, на *ADAS* страни садржани у *ARXML* моделу, а на *IVI* страни у *FDEPL* моделу, односно *.fdepl* датотеци (изузетак су једино параметри за верзију спреге који су у *.fidl* датотеци). Из тог разлога је потребно подржати и превођење између ова два модела.

Из учитаног *ARXML* модела се издвајају и преводе само параметри за рад конкретне спреге. Најпре се дефинишу парови свих спрежних пролаза си одговарајућих спрега и потом се за сваки од тих парова на основу мапиране припадности одређеној спрези и програмској компоненти покреће генерисање. Затим се креира *.fdepl* датотека са називом који настаје од спреге из *ARXML* модела коју описује. На почетку датотеке се генерише низ карактера, односно текст који представља повезивање са одговарјаућим *.fidl* датотекама на основу назива спреге, а потом се редом позивају компоненте које врше превођење вредности за идентификатор инстанце сервиса, *SOME/IP* верзије, инстанце догађаја, методе, поља за обавештавање, добављање и постављање вредности као што је приказано на дијаграму 29. Сви ови подаци су неопходни за изградњу заглавља *SOME/IP* протокола.



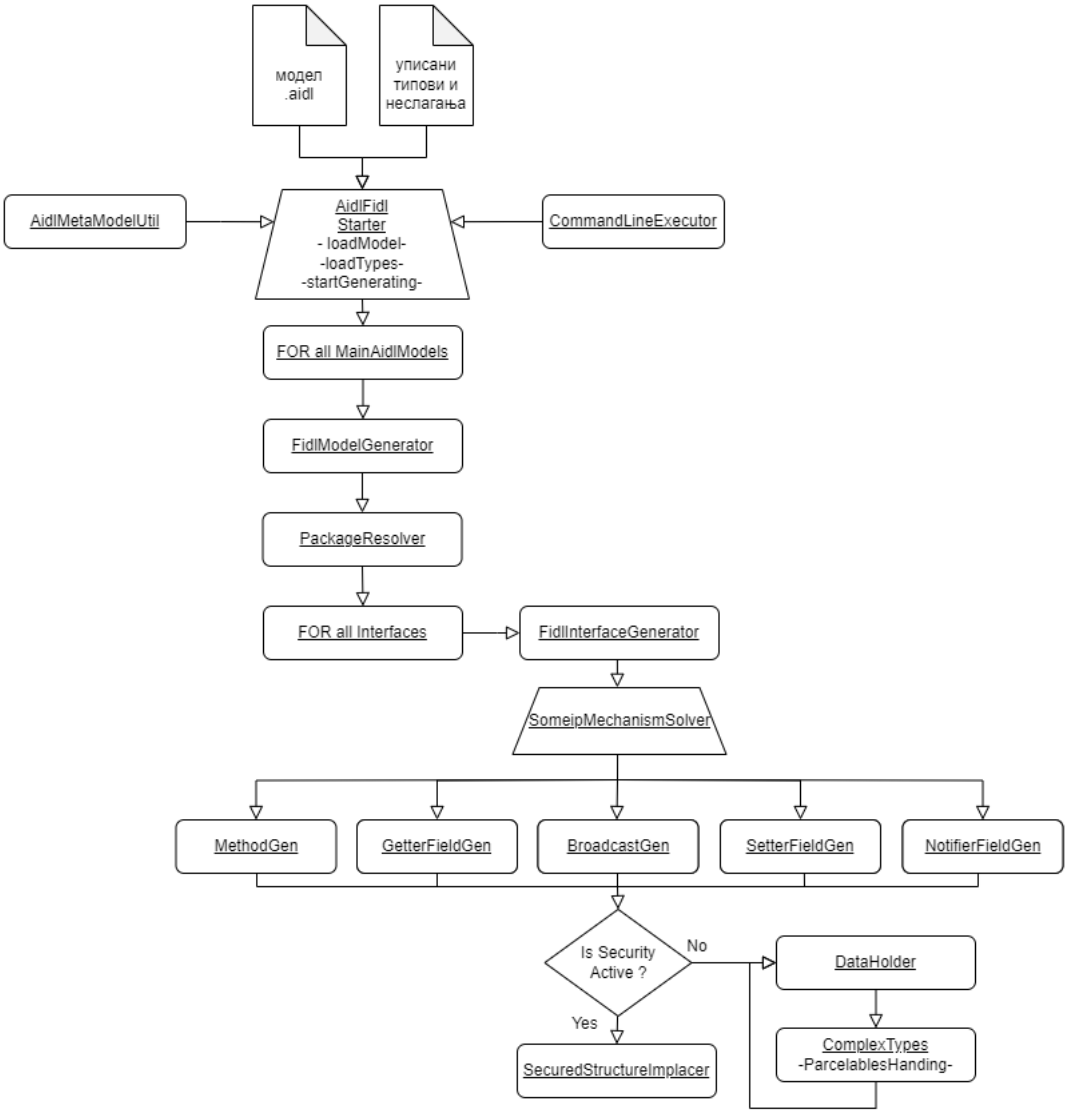
Слика 29: Архитектура превођења *ARXML* модела у *FDEPL* модел

Додатно, поред добијеног *FDEPL* модела, ствара се и помоћна дадотека која садржи параметре који описују верзију спреге који су у случају коришћења *Franca* радног оквира садржани у *FIDL* датотекама.

5.0.4 IV фаза - превођење између *AIDL* и *FIDL* модела

Принцип превођења из *AIDL* у *FIDL* модел је готово исти као и за претходно описану фазу 2. Фазом 2 су добијени релевантни објекти класа *AIDL* мета-модела које описују реалан модел и сада је те информације потребно мапирати у други модел.

Из главне *.aidl* датотеке се извлаче постојеће спреге које је потребно пресликати у *FIDL* модел. На основу имена метода које у главном *AIDL* моделу служе за добављање спрега упарених са одговарјаућом инстанцом спрежног пролаза сервиса и на основу назива пакета у ком се налазе се извлачи закључак шта су заправо инстанце истог спрежног пролаза. Од тога се у *FIDL* мета-моделу креира један објекат који дефинише спрегу са називом који ће одговарати почетном називу из *ARXML* модела. *FIDL* не садржи податке о самим инстанцама, то се задаје кроз *FDEPL*, зато у овом моменту то није ни потребно. Потом се за сваку од спрега пролази кроз њене методе како би се спрежним пролазима у *FIDL* моделу придружили одговарајући *SOME/IP* механизми са издвојеним називом који је, као и за спрегу, одржан у *AIDL* моделу, а одговара *ARXML* моделу. Ово се изводи на основу одговарајућих суфикса *"_EVENT"*, *"_METHOD"*, *"_GETTER"*, *"_SETTER"* или *"_NOTIFIER"* из назива метода.



Слика 30: Архитектура превођења *AIDL* модела у *FIDL* модел

Типови података за *FIDL* су у потпуности усклађени са *ARXML* моделом и неопходно је да буду идентични како би *SOME/IP* сервис и *SOME/IP* клијент могли правилно да комуницирају. Тако да се неусаглашености попут неозначеног типа у *AIDL* моделу решавају тако што се враћају у изворни тип на основу помоћне датотеке направљене током друге фазе генерисања у којој су назначени ови изузеци. Поред тога, сложени типови, односно структуре се конструишу на основу *Parcelable* типова. Уколико су сигурносни механизми активирани, генератор не узима у обзир типове из *AIDL* модела, већ предефинисану структуру. За разлику од *AIDL* модела, сви спрежни пролази и дефиниције свих структура могу бити у једној *.fidl* датотеци. Поред изузетака у превођењу типова, из исте помоћне датотеке се издвајају и параметри за верзију протокола.

Дијаграм превођења је дат на слици 30.

5.0.5 V фаза - генерисање *CommonAPI* средњег слоја

Попут фазе генерисања посредничке компоненте, генерисање преосталих фаза се одвија по принципу модел-у-текст парадигме. Најпре се на основу добијених *.fidl* и *.fdepl* датотека позивају алати за генерисање *CommonAPI* средњег слоја. Алати су део *capicxx-tools* пакета и власништво су *COVESA* (бивши *GENIVI*) компаније и отворени су за употребу као и већина њихових решења.

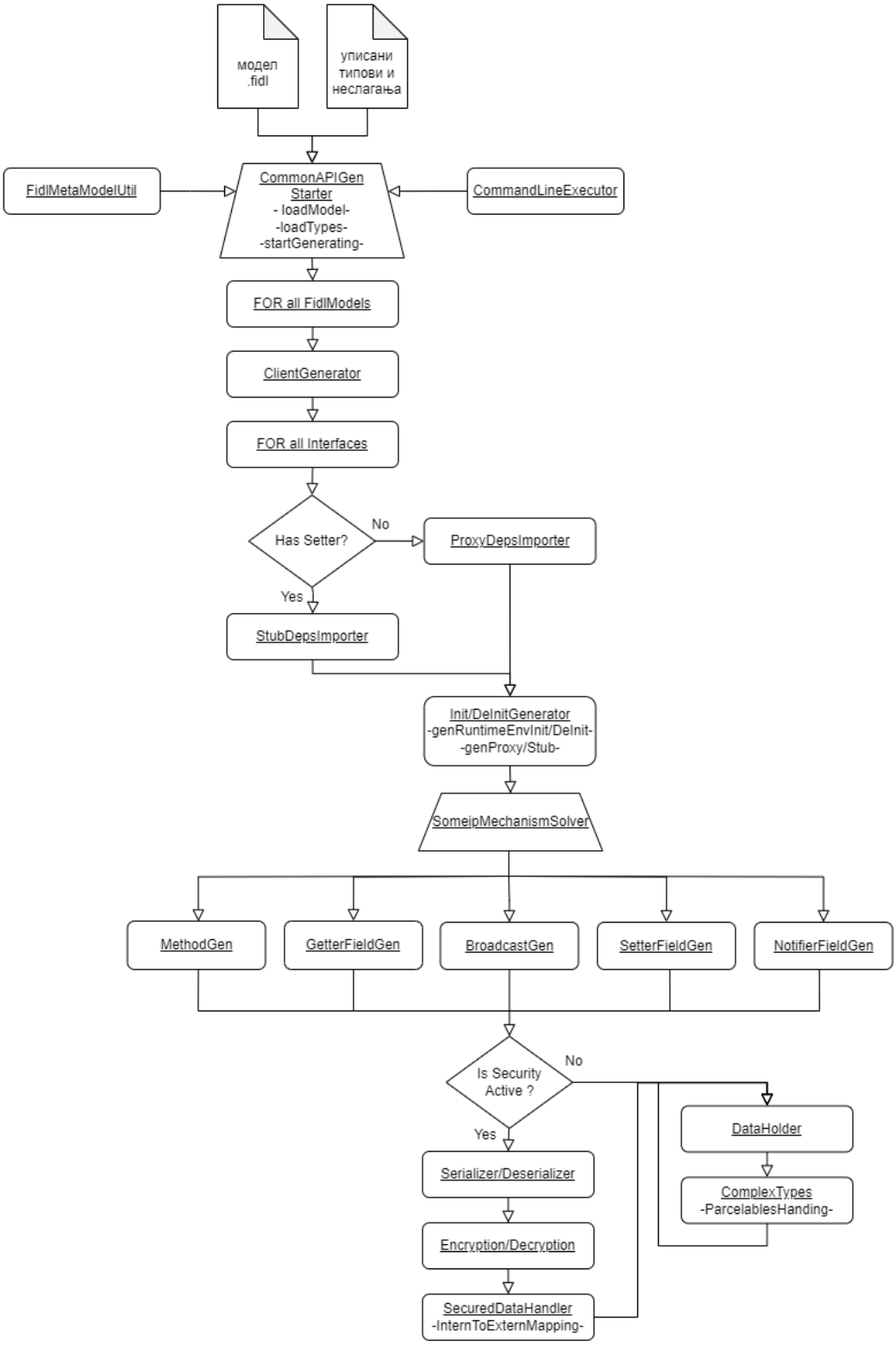
Након покренутог генерисања *capicxx* алатима добијају се делови *CommonAPI Core* и *CommonAPI Binding* компонената (као што је приказано на слици 8 у потпоглављу 2.2.2, а међу њима су најзначајније имплементације *Stub* и *Proxy* класа, које су круцијалне за остваривање комуникације. Објекти ових класа су, поред самог радног окружења, оно што се креира у процесима *SOME/IP* сервиса и клијента респективно и над чим се врше позиви дефинисане спреге овог протокола. У оквиру овог корака, након што су компоненте генерисане, ствара се и *CMake* датотека која је неопходна за превођење добијеног *C++* кода и стварање библиотеке која се може даље увезати у клијентску апликацију. *Stub* компоненте које служе за рад сервисне стране се користе само уколико постоји поље за постављање вредности.

Генерисање *CMake* датотеке извршава компонента *CommonAPIBuilder*. Она поставља предефинсане путање које се користе за превођење (и које могу бити измењене за потребни случај коришћења): *Android NDK*, архитектуру циљане платформе, верзију циљаног *Android* оперативног система, верзију преводиоца, коришћени *C++* стандард и задаје аргументе командне линије за покретање преводиоца. Пре него што компонента *CommonAPIBuilder* генерише *CMake*, она прави потребно окружење за превођење (структура директоријума, променљиве окружења и слично). Након што је *CMake* направљен, дешава се превођење.

5.0.6 VI фаза - генерисање *CommonAPI* клијента

Генерисање *CommonAPI* клијента се врши примарно на основу *FIDL* модела. Међутим, с обзиром на то да се овде генерише спона између *Android* сервиса и *SOME/IP* клијентских компонената (у случају поља за постављање вредности сервисним компонентама) биће потребно користити и помоћну датотеку створену током друге фазе, тј. превођења *ARXML* модела у *AIDL*. Разлог за то је до сад поменуто потенцијално неслагање у типовима података који су размењени путем *SOME/IP* протокола и њихове представе ка *Android* апликацијама.

Генерисање започиње тиме што се за сваки дефинисани спрежни пролаз стварају посебне датотеке заглавља (*.hpp*) и изворног програмског кода (*.cpp*). У датотекама заглавља ће бити генерисане искључиво декларације и укључивања зависности од генерисаних датотека *CommonAPI* модула средњег слоја (производ извршавања *capicxx* алата), док изворни код садржи и дефиниције функција. У једном пару ових датотека се налази руковање свим *CommonAPI SOME/IP* клијентима који врше комуникацију преко одговарајућег спрежног пролаза. Такође, у свакој датотеци изворног кода се у првом делу генеришу називи пакета учитани из *FIDL* модела. Поред тога се генерише постављање именског



Слика 31: Архитектура генерисања *CommonAPI* клијента

простора и мапа *Proxy* компонената, односно клијената на основу назива спреге. Потом следи генерисање функција за иницијализацију и деинизијализацију клијента које примају *int ID* као аргумент који идентификује инстанцу са којом треба комуницирати. Такав клијент се на основу прослеђеног аргумента смешта у мапу у случају иницијализације,

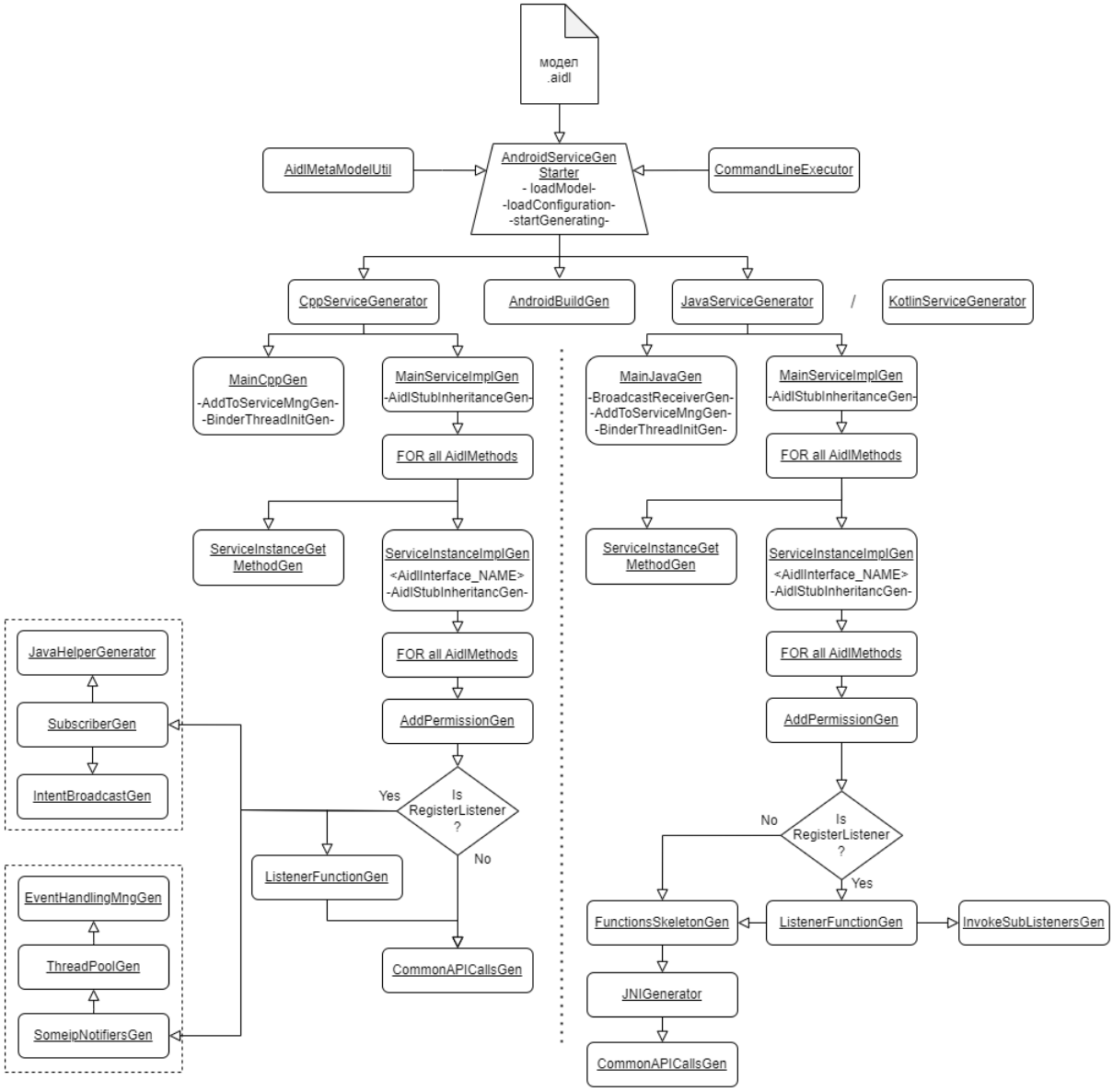
односно уклања из мапе у случају да је позвана деиницијализација. Након тога се генеришу функције које имплементирају у њему дефинисане *SOME/IP* механизме као што је описано у поглављу 4.

Функције које имплементирају сваки од постојећих *SOME/IP* механизама, такође, поседују генерисан *ID* као један од аргумената и на почетку свих функција се првобитно генерише провера да ли овај *ID* постоји у мапи, односно добављање потребног клијента над којим ће бити извршавани *SOME/IP* позиви. Посебна компонента је имплементирана да изврши генерисање за сваки појединачни механизам. Методе и поља за добављање вредности су најједноставнији и захтевају прављење само једне методе која извршава овај задатак. Њен назив је добијен из кованице именског простора, назива методе из модела и назива спреге из модела. За догађај, поље за обавештавање и поље за постављање вредности је неопходно пружити методе за регистрацију и одјаву претплате на одређени механизам. Називи аргумената су идентични као што је дато у моделу.

Током генерисања посебна компонента је задужена и за поставку типова вредности које се користе и размењују, односно компонента која рукује типовима података. Све неусаглашености у типовима података се решавају у програмском коду који генерише ова компонента. Неозначени број у озачени број већег ранга, *String* класа у *char** (из разлога што *string* тип из *C++* стандарда и *string* тип из *NDK* окружења нису једнаки тј. усаглашени по основи), као и руковање сложеним типовима и векторима. Податке у типовима *Parcelable* је потребно мапирати на структуре сложених типова. У оквиру овог мапирања се генеришу конструктори копирања и премештања како би се побринули за то да се подаци правилно копирају и премештају у меморији, како не би дошло само до копирања показивача, нити до остатака низова. Ово се примењује и на случаје угњеждених сложених структура података, тако да се конструктори копије позивају ланчано све до најнижег типа који копира сам податак.

Поред тога, уколико су укључени сигурносни механизми, посебна компонента служи да генерише уметнути програмски код за серијализацију и шифровање одлазних, односно дешифровање и десеријализацију долазних података. Методе које врше генерисање серијализације и десеријализације првобитно генеришу позиве *boost* библиотеке који конструишу контејнер за ток серијализованих/десеријализованих података и затим генеришу позив оператора за серијализацију/десеријализацију. Уколико провером типова података којима се рукује буде утврђено да је реч о сложеном типу, односно структури, за потребе серијализације и десеријализације се генерише и слободна шаблонска функција неопходна за рекурзивно позивање оператора све док не дође до основних типова података које оператор из *boost* библиотеке може да серијализује, тј. десеријализује.

Уколико се у спрежном пролазу утврди да је дефинисано поље за постављање вредности, процес је идентичан, с тим да се при генерисању инцијализацијског и деиницијализацијског дела у оквиру радног окружења врши конструисање сервисних компонената (*Stub*) уместо клијентских (*Proxy*) и њихово смештање, односно манипулисање кроз генерисану мапу.



Слика 32: Архитектура генерисања *Android* сервиса

5.0.7 VII фаза - генерисање *Android* сервиса

Генеирсање *Android* сервиса је највише конфигурабилна фаза у читавом процесу генерисања. Њом је могуће дефинисати: програмски језик у ком ће сервис бити генерисан (*C++*, *Java* или *Kotlin*) као и партицију где ће сервис бити смештен на плочи (*vendor* или *system_exp*). Компоненте које започињу генерисане се разликују на основу тога који програмски језик је одабран.

5.0.7.1 *Android C++* сервис Уколико је одабран *Android C++* сервис онда се генерише неколико сегмената као што се може видети на левој страни дијаграма 32. Најпре main.cpp датотека у којој се иницијализује сервис, додаје у *ServiceManager* и дефинише складиште *binder* нити. Затим се генришу датотеке заглавља и изворног кода које имплементирају сервисну компоненту главне *AIDL* спреге. У оквиру ових датотека конструишу

се *binder* објекти који настају од других *AIDL* спрега подударних са спрежним иннстанцама на *ADAS* страни и генеришу се само дефиниције *AIDL* метода којима се добавља одговарајући *binder* објекат на захтев корисничке апликације.

Након тога се за сваку од ових *AIDL* спрега генеришу посебне датотеке заглавља и изворног кода које у себи садрже назив спреге. У свакој од њих се првобитно генерише наслеђивање одговарјауће *Stub* компоненте и започиње генерисање имплементације метода дефинисаних самом спрегом. Пре генерисања самих метода, генерише се и вектор *binder* објеката ослушкивача за обавештавајући *SOME/IP* механизам. Називи метода остају исти као што је дато у моделу. У методама се редом генеришу делови за проверу дозвола, тј. пермисија (пермисије се постављају по потреби, као низа карактера који ће бити коришћен у генерисаним функцијама из системске компоненте за за управљање пермисијама - енгл. *Permission Manager*). Затим се у методама генеришу позиви одговарајуће функције из *CommonAPI SOME/IP* клијента представљене у претходној фази. У случајима када се генеришу методе задужене за претплаћивање на догађај, поље за обавештавање или постављање вредности генеришу се и слободне функције за сваки од ових механизама које ће представљати претплаћене повратне функције и које ће на пристизање података прозивати регистроване *binder* објекте корисничких апликација.

Иако је реч о *C++* сервису, када постоји неки од обавештавајућих *SOME/IP* механизама, односно када постоји бар једна *"Register...Listener()"* функција, посебна компонента генерише и *Java* код за помоћни *Java* сервис који на назив *AIDL* спреге додаје суфикс *"JavaHelper.java"*. Овај помоћни *Java* сервис се претплаћује на све обавештавајуће механизме једне спреге *C++* сервиса. Међутим, биће искоришћен само уколико број претплатника пређе дефинисани праг - до тад је само складиштен као ослушкивач у *C++* сервису који не буде побуђиван на долазне инфрормације. Генерисан је из три дела и сва три дела се генеришу за сваки пронађени обавештавајући механизам (односно *"Register...Listener()"* функцију). Први је конструисање *binder* објекта за претплату на одговарајући *SOME/IP* механизам, други је позив метода из *AIDL* спреге којим се овако конструисан *binder* објекат региструје на долазеће податке и трећи је генерисање имплементације метода које ће бити прозване над регистрованим *binder* објектом при приспећу података и у којима се стварају одговарајући *Intent* сигнали.

Поред тога, ако је конфигурација за квалитет услуге активна, биће генерисан и руковалац подацима пристиглим путем неког од три *SOME/IP* механизма за обавештавање. Попут помоћног *Java* сервиса и овај руковалац је генерисан из три дела. Први део је окосница компоненте руковаоца у склопу које се генерише и део за додавање претплатника у мапу, тј. пара где је кључ приоритет, а вредност ослушкивач за обавештавајући механизам који је пристигао. Други део генерише иницијализацију и деиницијализацију складишта нити и њиме се генеришу потребни механизми за синхронизацију као и главна функција која управља извршавањем свих нити проверавајући њихове статусе и одбацивајући нити и ослушкиваче из контејнера у којима се чувају уколико се извршавају дуже него што је дозвољено. Трећи је компонента за отпремање догађаја, она генерише функцију која извршава посао прозивајући претплаћену слободну функцију и дајући јој као први аргумент ослушкивач са највећим приоритетом, односно ослушкивач са врха мапе и као остале аргу-

менте пристигле податке. Уколико не постоји активација квалитета услуге, ослушкивачу се редом прозивају у петљи и све произвајуће методе су асинхроне без повратне вредности с обзиром на то да имају *"oneway"* ознаку.

5.0.7.2 *Android Java/Kotlin* сервис Случај генерисања *Java* и *Kotlin* сервиса је готово идентичан. Разлика је, као што је већ објашњено, једино у томе како је реализовано конкурентно отпремање података добијених путем обавештавајућих *SOME/IP* механизама.

У њима је, као и у *C++* сервису, ток генерисања идентичан, с тим да је потребна компонента која генерише *JNI* део сервиса задуженог за позиве *CommonAPI* клијената јер су они креирани *C++* језиком. Са друге стране, није потребно имати помоћни *Java* сервис, јер је једина његова сврха отпремање *Intent* сигнала, а то је сада могуће непосредно из сервиса. То значи да се сад након преласка прага за број претплатника, за обавештавање корнисчких апликација не користе њихови *binder* објекти нити нека додатна екстерна компонента већ се само прелази на генерисани програмски код који ствара и дифузно отпрема *Intent* сигнале.

Дакле, уместо генерисања main.cpp датотеке и посебних датотека заглавља и изворног кода, генерише се *.java* датотека назива који настаје од назива одговарајуће прогграмсе компоненте на *ADAS* страни и који је садржан у главној *AIDL* спрези. У овој датотеци биће генерисана класа истог назива као и у случају *C++* сервиса (назив компоненте и суфикс *"Impl"*) која проширује системску класу *Service*. За покретање процеса који извршава методе дефинисане унутар ове класе задужена је друга генерисана датотека назива *"SomeipOnBoot.java"* која проширује *BroadcastReceiver* и у којој је генерисано реаговање на побуду сигнала емитованог приликом подизања система на плочи. Генерисан одговор на прихватање тог сигнала је покретање свих сервиса насталих од главних *AIDL* спрега у првом плану.

У главном сервису су, такође, генерисане методе које на захтев добављају *binder* објекте *AIDL* спрега преко којих се заиста врши комуникација.

Свака *.java* датотека која имплементира овакву *AIDL* спрегу садржи имплеметацију свих потребних метода. У методама је такође најпре генерисана провера пермисија као и у случају *C++* сервиса, а затим се стварају позиви *JNI* функција. У оквиру имплементације ових функција је генерисано позивање одговарајућих функција из *CommonAPI* клијената. За добављање података путем обавештавајућих *SOME/IP* механизама у овом случају је потребно генерисати две повратне функције, једну у *JNI*, другу у *Java* делу сервиса. *JNI* повратна функција се претплаћује на механизам, па се из ње прозива обавештавајућа функција написана у *Java* делу сервиса која прозива регистроване *binder* објекте клијената. Да би у *JNI* делу, приликом регистровања и претплате путем *CommonAPI* клијента, било могуће прозвати функцију из *Java* класе битно је на почетку *JNI* функција генерисати чување инстанце радног окружења (*JavaVM*) како би се из ње могли добавити идентификатори класе, методе и поља одговарајућег *Java* објекта из сервиса који је потребно обавестити.

Додатно, за сваки од сервиса, независно од програмског језика у ком су писани, су

посебном компонентом генерисане и датотеке за превођење у *.bp* формату као и датотеке потребне за управљање *SELinux* механизмима (*.te*)

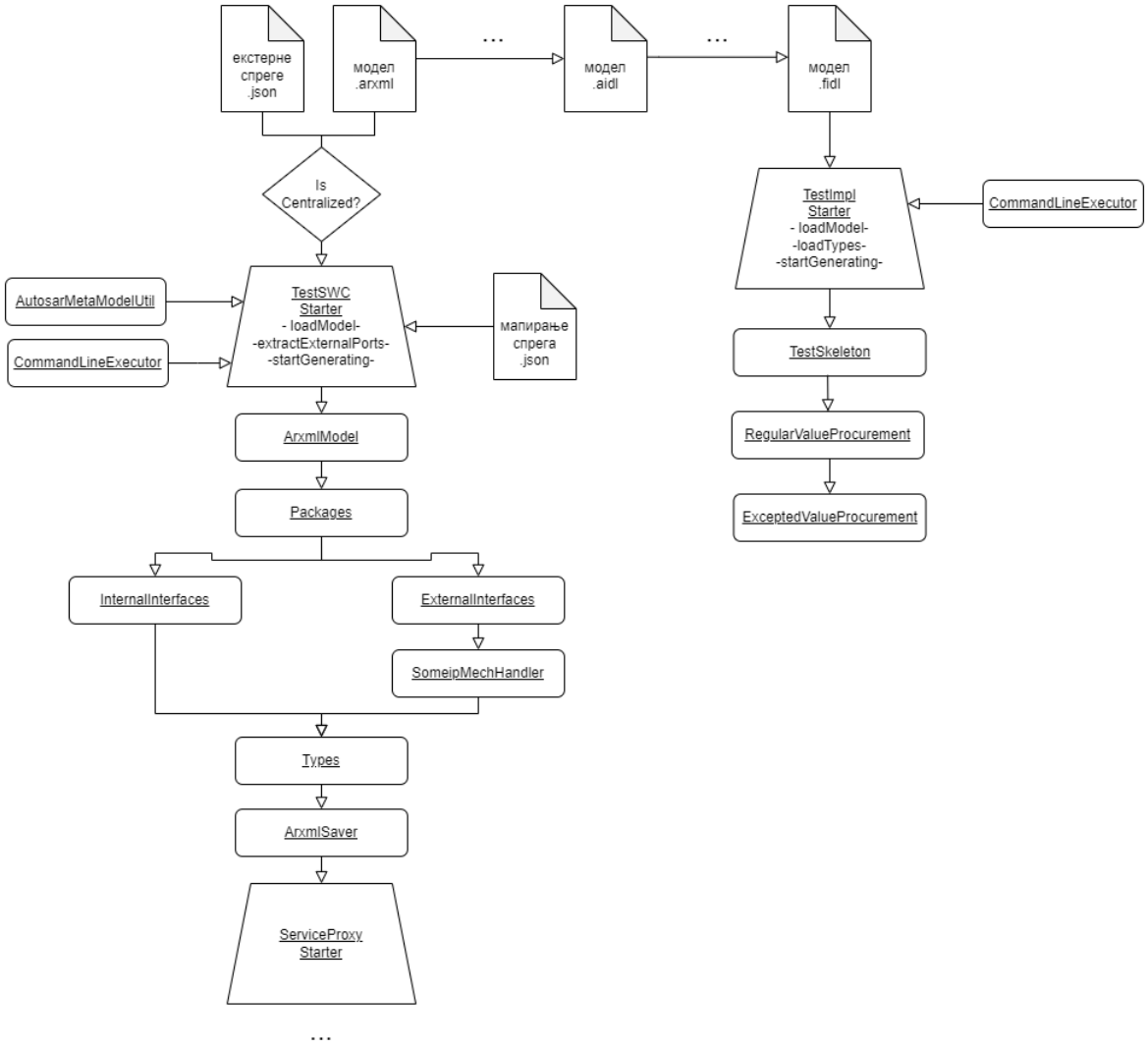
5.0.8 VIII фаза - генерисање тестова за комуникацију

Поред генерисања програмских компонената које имплементирају решење и врше размену података, имплементирано је и генерисање тестова који служе да верификују комуникацију међу генерисаним компонентама, односно помогну у проналажењу проблема уколико подаци не пристижу у жељеном маниру. Генерисање тестова се значајно ослања на претходне фазе генерисања и по потреби их понавља. Тестови се извршавају на страни *IVI* домена, с обзиром на то да ту пристужу подаци.

На *ADAS* страни промене изазване фазом генерисања тестова постоје једино у случају централизованог приступа. У централизованом приступу генерише се додатна, тестна програмска компонента чија је сврха да се путем ње врши потврда размењиваних података. Генерисана тестна компонента ће имати исте унутрашње спреге као и посредничка (*ServiceProxy*) компонента, али ће за спољашње спреге користити искључиво поље за добављање као најједноставнији механизам за употребу и метод. Метод је коришћен само у случају мапирања унутрашњих метода зато што пољем за добављање вредности није могуће задати улазне аргументе. У случају унутрашњег поља за постављање вредности, тестна компонента не мапира механизам ка споља, односно није могуће користити тестну компоненту у процесу верификације. Разлог за то је што је за унутрашњу комуникацију у случају поља за добављање вредности *ServiceProxy* компонента сервисна страна, а циљана *ADAS* компонента клијент (јер клијент шаље захтев за постављање вредности на сервис) и један клијент преко истог спрежног пролаза не може да шаље захтеве на различите сервисе. Модел тестне компоненте се најпре генерише по узору на модел посредничке компоненте, стварањем објекта *ARXML* модела, а потом реплицирањем улазних спрега и мапирањем на одговарјуће спољашње механизме и типове (овај корак је неопходно допунити ручно како би сви параметри комуникације били исправни и компонента правилно распоређена у извршавању). Након што је модел тестне компоненте створен, позивима прве фазе генератора се генерише и њена имплементација на идентичан начин као и за посредничку *ServiceProxy* компоненту. Затим се модел тестне компоненте даље преводи и од њега се генерише *CommonAPI* тестни клијент.

На *IVI* страни је неопходно најпре генерисати костур тестова (потпис функције, макрое из коришћеног *gTest* окружења којима се врши иницијализација, проверавају очекиване вредности и пријављују резултати), а потом се генеришу позиви којима се добављају корисне информације путем генерисаних *CommonAPI SOME/IP* клијената. Додатно, генерише се податак који је очекивана вредност теста, односно вредност са којом је потребно упоредити пристигле вредности. У случају централизованог податка, тестови нуде могућност провере пристиглих података упоређивањем са, такође генерисане, *SOME/IP* спреге са тестном компонентом.

Сам начин тестирања решења је описан у поглављу 6 за верификацију, док је овде дат искључиво начин генерисања који се може видети на слици 33.



Слика 33: Архитектура генерисања тестова за комуникацију

6 Верификација и резултати

Процена решења датог овом дисертацијом је подељена у две велике целине: експериментална процена функционалности односно учинковитости извршавања и испитивање рада алата за генерисање који ће бити детаљно описани у овом поглављу. Решење је примењено на неколико референтних платформи са спецификацијама датим у табелама 7 и 8 како би показало независност од саме физичке архитектуре: *Telechips TCC805x (Dolphin3)* и *Qualcomm SA8155P* на страни инфо-забавног домена и *Alpha AMV* и *Nvidia Jetson AGX Xavier* на страни домена који пружа напредну подршку возачу.

Табела 7: Особине платформи на страни инфо-забавног домена

	Qualcomm SA8155P	**Telechips TCC805x (Doplhin3)**
Централна процесорска јединица	8 x Kryo ARMv8	ARM Cortex-A72 и ARM Cortex-A53
Радна меморија	LPDDR4X 12GB	LPDDR4X 16GB
Графичка процесорска јединица	Adreno640	Imagination PowerVR
Јединица за дигиталну обраду сигнала	6 x Hexagon	-
Мрежни пролази	10/100/1000 Base-T	10/100/1000 Base-T Ethernet AVB
Остале спреге	I/O, USB2, USB3.1, PCIe, I2S, I2C, DisplayPort	USB2.0, USB3.0, DisplayPort, CAN, LIN, UART, MIPI, JTAG
Оперативни систем	Android Automotive 11, Android Automotive 12	Android Automotive 10, Android Automotive 11

Табела 8: Особине платформи на страни домена који пружа напредну подршку возачу

	Alpha AMV (3 x TDA2x SoC)	**Nvidia Jetson AGX Xavier**
Централна процесорска јединица	2 x ARM Cortex-A15 и 4 x ARM Cortex-M4	Nvidia Carmel ARMv8.2
Радна меморија	DDR3 1.5GB	LPDDR4X 8GB
Графичка процесорска јединица	SGX544	Nvidia Volta (512 CUDA језгара)
Јединица за дигиталну обраду сигнала	2 x C66x	-
Мрежни пролази	10/100/1000 Base-T	10/100/1000 Base-T
Остале спреге	MIPI CSI-2, PCIe, CAN, UART, HDMI, JTAG	MIPI CSI-2, PCIe, CAN, HDMI, USB2.0, USB3.1, UART, SPI, I2C, I2S, GPIO
Оперативни систем	- Linux + Adaptive AUTOSAR (SoC спрегут са IVI доменом) - TI-RTOS (SoC за алгоритме)	Linux + Adaptive AUTOSAR

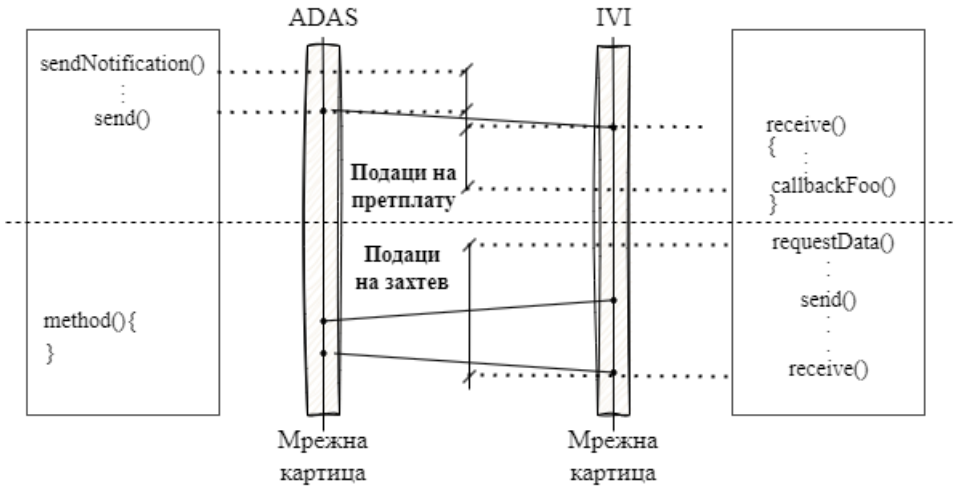
* Подаци за Alpha AMV развојну плочу у табели су дати на основу једног TDA2X система на чипу уколико уз сам податак коментаром није назначено другачије

6.1 Експериментална процена функионалности

Експериментална процена функционалности се најпре односи на мерења заузећа ресурса при извршавању предложеног решења, као и кашњења која измењени комуникациони механизми уносе. Мерења служе да утврде подобност за примену у реалним системима и сценаријима и детаљно разлуче случаје употребе различитих представљених приступа.

У оквиру експерименталне процене функционалности узорци су модели компонената *ADAS* домена и спрега међу њима, као и саме тестне компоненте из оба система које опонашају реалне сценарије и размену различитих типова података. У оквиру тестних компонената биће покривени сви механизми комуникације путем одабраног *SOME/IP* протокола. Тако су дата мерења за међудоменски пренос у дистрибуираном и централизованом случају, као и за даље прослеђивање података у случају коришћења *Android* оперативног система.

1. Учинковитост дистрибуираног приступа представљена је у табели 9, где је могуће видети времена извршавања за сваки појединачни *SOME/IP* механизам. Сваки од механизама је мерен са десет хиљада узорака и дате су најмања, средња и највећа измерена вредност. Поред тога је рачуницом добијена стандардна девијација (σ) која за метод, догађај, поље за обавештавање, добављање и постављање износи редом 1.801, 2.227, 3.76, 2.493 и 2.375. На основу стандардне девијације је на слици 35 дата расподела измерених узорака. Могуће је приметити да је расподела узорака за све комуникационе механизме слична и да се, у случају поља за обавештавање, највећи проценат (око 87% измерених узорака) налази у околини σ у односу на средњу вредност измерених узорака. Приближно 91% се налази у опсегу 2σ, односно ∼95% у опсегу 3σ. Остали комуникациони механизми

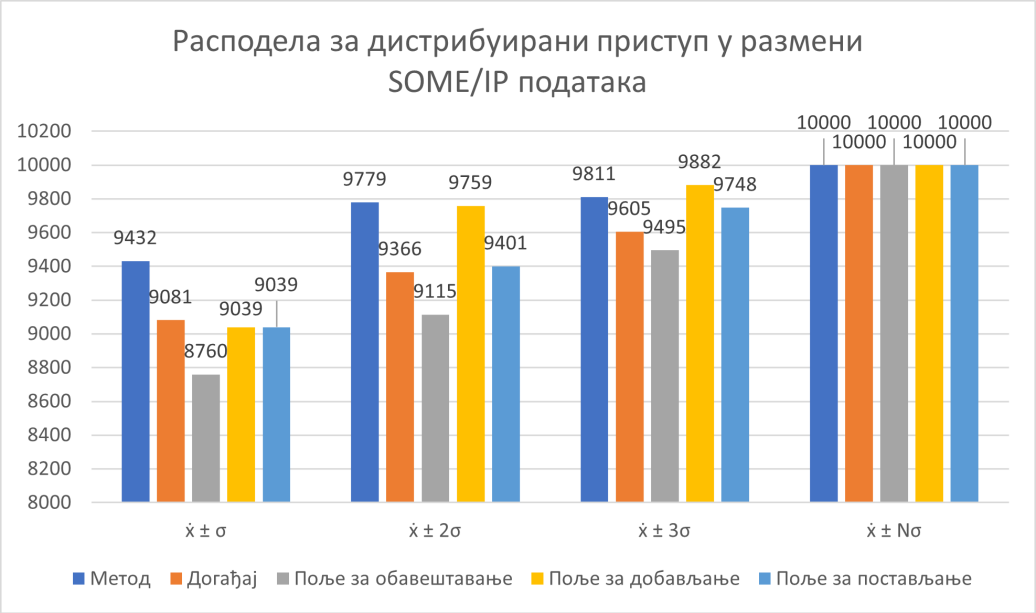


Слика 34: Мерени елементи и интервали при дистрибуираном приступу

имају нешто ефикаснију концентрацију узорака у најмањој σ околини. Мерена путања је илустрована на слици 34 где се види да су измерене све фазе комуникације, од извршавања функција програмске спреге на страни *ADAS* домена, времена које податак проведе на мрежи и времена за прихватање података у *SOME/IP* клијенту на *IVI* страни. Могуће је приметити да експериментална мерења прате семантичку и имплементациону поделу механизама на оне који врше добављање података на захтев и на претплату па су времена извршавања за метод и поље за добављање готово идентични, односно догађаји и поља за обавештавање. Поља за постављање вредности, иако семантички решењем припадају добављању података на претплату, уводе кашњење слично методи и пољу за добављање из разлога што је постављање вредности такође повратни механизам и очекује потврду да је постављање извршено. То значи да ће сам податак бити пренет брзином која одговара догађају и пољу за обавештавање, али укупно кашњење у систему подразумева и повратну информацију. У оквиру мерења за дистрибуирани приступ нису дате вредности потрошње ресурса и оптерећења платформе јер то зависи у потпуности од самих компонената, а с обзиром на то да су додаци које уводи ово решење само позиви функција из комуникационог стека, обрада и извршавање компоненте зависи од треће стране која имплементира одговарајућу компоненту. Такође, посебна напомена у овом приступу је да су мерења дата без укључених сигурносних механизама услед немогућности њихове имплементације из разлога представљених у поглављу 4.1.

Табела 9: Мерење *SOME/IP* комуникације у случају дистрибуираног приступа

	Најмања вредност $[ms]$	Највећа вредност $[ms]$	Средња вредност $[ms]$
Метод	6.273	34.143	9.669
Поље за добављање	6.315	27.071	10.922
Догађај	1.951	12.816	2.93
Поље за постављање	4.966	28.859	7.375
Поље за обавештавање	2.483	22.739	3.277



Слика 35: Расподела за дистрибуирани приступ у размени *SOME/IP* података

2. Учинковитост централизованог приступа је представљена табелама 10, 11, 12, 13 и 14, где је могуће видети времена извршавања за сваки појединачни *SOME/IP* механизам, такође на броју узорака од по десет хиљада. Међутим, овај пут укупно време подразумева и унутрашњу комуникацију између *ADAS* компонената и посредничке компоненте. Мерена путања је илустрована на слици 37 где се, такође, види да је мерено време извршавања функција програмске спреге на страни *ADAS* домена, време на мрежи и време за прихватање података у *SOME/IP* клијенту на *IVI* страни. Као и у случају дистрибуираног приступа, рачуницом је добијена стандардна девијација (σ) која износи 3.418, 1.917, 2.36, 3.194 и 2.755 за метод, догађај, поље за обавештавање, добављање и постављање респективно. Додатно је, и овај пут, расподела измерених узорака на основу стандардне девијације приказана на слици 36. Јасно се уочава да је расподела узорака за све комуникационе механизме слична и да је највећи проценат (око 91% измерених узорака) налази у околини σ у односу на средњу вредност измерених узорака. Приближно 95% се налази у опсегу 2σ, односно ∼98% у опсегу 3σ за метод који има најмању ефикасност узимајући у обзир количину узорака који се налазе у најмањој σ околини.

Табела 10: Мерење *SOME/IP* методе у случају централизованог приступа

	Најмања вредност $[ms]$	Највећа вредност $[ms]$	Средња вредност $[ms]$
Унутрашња размена	0.425	1.089	0.828
Спољашња размена	6.282	29.518	9.545
Сигурносни механизми на *ADAS* страни	0.544	2.171	0.711
Сигурносни механизми на *IVI* страни	0.4	2.003	0.522

Табела 11: Мерење *SOME/IP* поља за добављање у случају централизованог приступа

	Најмања вредност $[ms]$	Највећа вредност $[ms]$	Средња вредност $[ms]$
Унутрашња размена	0.434	1.063	0.801
Спољашња размена	6.086	25.661	10.396
Сигурносни механизми на *ADAS* страни	0.362	1.122	0.316
Сигурносни механизми на *IVI* страни	0.249	0.993	0.275

Табела 12: Мерење *SOME/IP* догађаја у случају централизованог приступа

	Најмања вредност $[ms]$	Највећа вредност $[ms]$	Средња вредност $[ms]$
Унутрашња размена	0.220	0.423	0.271
Спољашња размена	2.073	36.655	3.126
Сигурносни механизми на *ADAS* страни	0.16	1.445	0.305
Сигурносни механизми на *IVI* страни	0.183	0.698	0.221

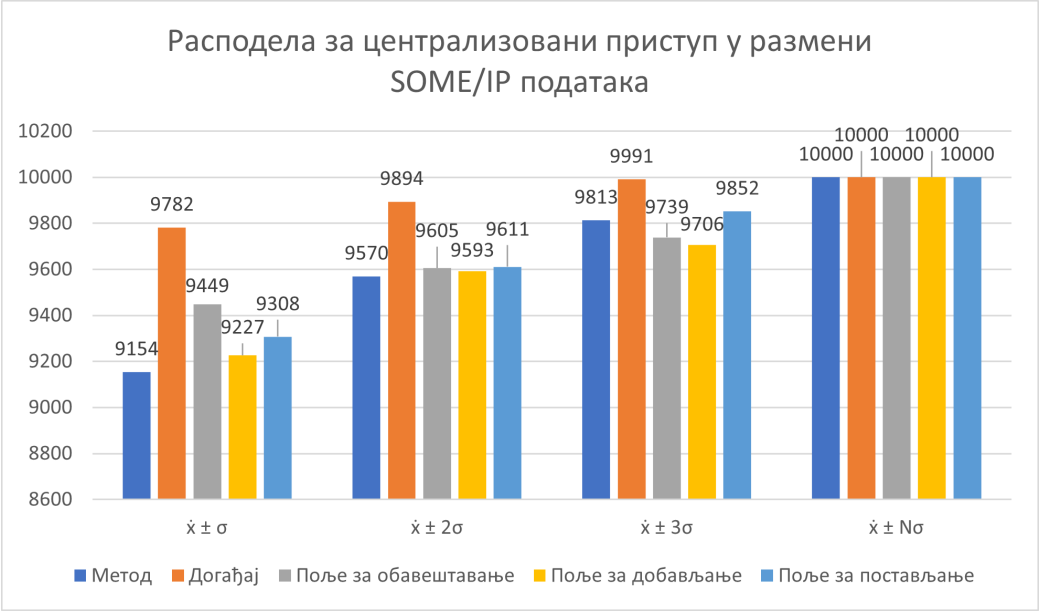
Табела 13: Мерење *SOME/IP* поља за постављање вредности у случају централизованог приступа

	Најмања вредност $[ms]$	Највећа вредност $[ms]$	Средња вредност $[ms]$
Унутрашња размена	0.504	0.952	0.565
Спољашња размена	5.1	29.988	7.616
Сигурносни механизми на *ADAS* страни	0.323	2.051	0.243
Сигурносни механизми на *IVI* страни	0.204	0.542	0.210

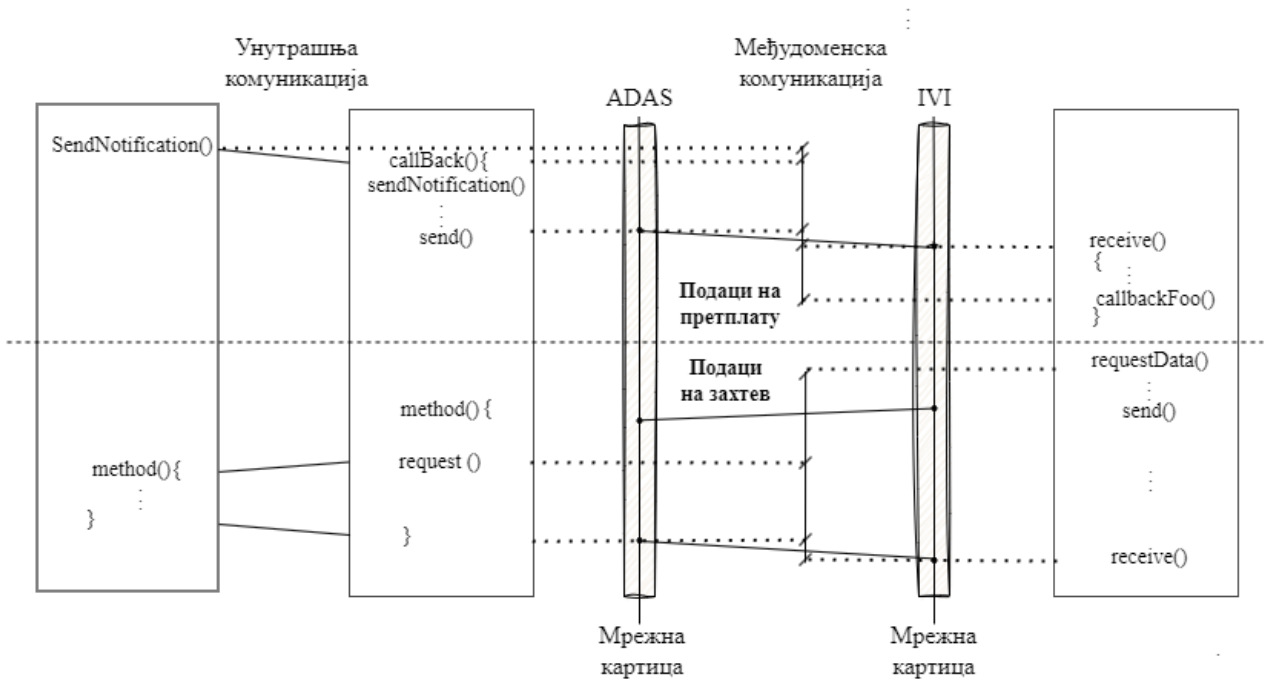
Табела 14: Мерење *SOME/IP* поља за обавештавање на промену вредности у случају централизованог приступа

	Најмања вредност $[ms]$	Највећа вредност $[ms]$	Средња вредност $[ms]$
Унутрашња размена	0.274	0.499	0.326
Спољашња размена	2.263	21.671	3.45
Сигурносни механизми на *ADAS* страни	0.178	1.315	0.347
Сигурносни механизми на *IVI* страни	0.169	0.742	0.196

Такође, поново је потребно напоменути понашање у случају поља за постављање вред-



Слика 36: Расподела за централизовани приступ у размени *SOME/IP* података



Слика 37: Мерени елементи и интервали при централизованом приступу

ности, где се могу раздвојити време потребно искључиво за пренос података од клијента који врши захтев за постављање до одредишта, тј. сервиса и неопходне повратне информације о успешности постављања. С обзиром на то да је посредничка компонента у потпуности створена овим решењем, заузеће меморије и процесора које она ствара ће бити дати касније на употребним примерима из функционалних тестова у табели 28.

3. У случају *Android* оперативног система - време за прослеђивање информације од *SOME/IP* клијента, преко *Android* сервиса, до корисничких апликација је дато у табелама

15-20. За потребе мерења кашњења у комуникацији која омогућава прикупљање података на захтев број тестних корисника је 1, 3 и 5, док је број активних спрега које ти корисници користе 1, 2 и 3. Са друге стране, за прикупљање података на претплату број тестних корисничких апликација је 1, 5, 10, 30 и 50, док број активних спрега на које подаци путем *SOME/IP* протокола стижу брзином од 30 порука у секунди износи 1, 2 и 3. Времена су дата за *C++*, *Java* и *Kotlin* сервисе. Из представљених резултата могуће је дедуковати неколико наредних тврдњи. Најбоље перформансе нуди *C++* сервис. За сваки од сервиса измерене су брзине преноса података путем *Intent* механизма које износе 12.145ms за *C++*, односно 9.621ms за *Java* и *Kotlin* сервисе. Ова времена треба посматрати као граничне вредности преласка са *AIDL binder* механизма на *Intent*. Из времена датим у поменутим табелама могуће је запазити да се за *C++* сервис ова времена јављају у ситуацијама када је број корисничких апликација које је потребно опслужити 50 (без обзира на расподелу претплаћених корисника и активних механизама, тј. свеједно је да ли је 50 корисника, а 1 активан механизам добављаљања података на претплату или је 25 корисника а 2 су активна механизма са којих подаци истовремено долазе). За случај *Java* и *Kotlin* сервиса гранични случај представља сценарио са више од 10 корисника. За правилно тумачење резултата важно је и напоменути да сваки од сервиса има способност истовременог опслуживања две спреге (без обзира да ли је реч о два захтева који долазе са различитих корисничких апликација или су два упоредна захтева једне апликације). Због ове конкурентне обраде је разлика између 2 и 3 активне спреге са корисницима (без обзира да ли је реч о добављају података на захтев или претплату) већа него разлика између 3 и 4. Разлог је што је скуп нити којима располаже *binder* у сервисима постављен на вредност 2 како би у теорији било омогућено истовремено регистровање и опслуживање нових клијената, иако у пракси није могуће одредити која нит ће прихватати који посао. Специфично за податке пристигле на претплату могуће је уочити драстично повећавање кашњења при повећавању активних спрега за велики број корисника (30 и 50), а разлог је тај што долази до нагомилавања пристиглих информација. Подаци се гомилају јер при учесталом пристизању сервис није у могућности да их проследи свима. Због овога поједини пакети стижу до одредишне апликације значајно касније (и до неколико секунди кашњења). Потрошња ресурса и оптерећење процесорске јединице ће бити приказани касније у сценаријима коришћеним за функционалне тестове у табели 28.

Потрошња ресурса и оптерећење је мерено *top* командом која у *Linux* базираним системима интерагује са језгром путем */proc* система датотека и извршавањем системских позива. Са друге стране за мерење времена извршавања функција и размењивања података коришћени су *Wireshark*, *tcpdump* и методе бележења системских часовника из самог кода. За потребе мерења размене података између *ADAS* и *IVI* система у једном смеру неопходно је узети у разматрање и разлике у часовницима. За испуњавање тог захтева, није било могуће ускладити их неким трећим временом које ће представљати апсолутну тачност и прецизност, већ је приоритет на њиховој међусобној усклађености. Стога је прибегнуто сличном принципу који имплементира *PTP* протокол за синхронизацију времена.

Пре сваког извршеног мерења, било је неопходно одредити однос између часовника на две различите платформе физичке архитектуре које учествују у комуникацији као што

Табела 15: Мерење размене података на захтев корисничких апликација са *Android C++* сервисом

		Подаци на захтев		
↓Број корисника		1	2	3
	Најмања вредност $[ms]$	2.21	2.419	5.472
	Највећа вредност $[ms]$	3.544	3.98	8.421
1	Средња вредност $[ms]$	2.968	3.197	6.994
	Најмања вредност $[ms]$	5.799	7.657	11.134
	Највећа вредност $[ms]$	8.471	10.676	17.642
3	Средња вредност $[ms]$	6.699	9.553	15.919
	Најмања вредност $[ms]$	7.614	11.916	20.802
	Највећа вредност $[ms]$	10.02	17.163	27.537
5	Средња вредност $[ms]$	9.358	16.595	25.19

Табела 16: Мерење размене података на захтев корисничких апликација са *Android Java* сервисом

		Подаци на захтев		
↓Број корисника		1	2	3
	Најмања вредност $[ms]$	3.639	4.716	9.723
	Највећа вредност $[ms]$	6.209	8.008	15.768
1	Средња вредност $[ms]$	5.244	5.921	13.67
	Најмања вредност $[ms]$	10.558	13.973	21.735
	Највећа вредност $[ms]$	15.508	19.47	31.346
3	Средња вредност $[ms]$	12.885	16.965	28.972
	Најмања вредност $[ms]$	13.855	21.076	35.579
	Највећа вредност $[ms]$	21.278	30.433	49.055
5	Средња вредност $[ms]$	16.823	30.38	43.963

Табела 17: Мерење размене података на захтев корисничких апликација са *Android Kotlin* сервисом

		Подаци на захтев		
↓Број корисника		1	2	3
	Најмања вредност $[ms]$	3.989	4.999	9.415
	Највећа вредност $[ms]$	6.995	7.562	15.121
1	Средња вредност $[ms]$	5.793	6.433	12.956
	Најмања вредност $[ms]$	11.233	14.752	22.258
	Највећа вредност $[ms]$	16.501	19.836	33.817
3	Средња вредност $[ms]$	13.156	17.542	28.493
	Најмања вредност $[ms]$	13.344	21.348	38.333
	Највећа вредност $[ms]$	20.877	30.022	51.61
5	Средња вредност $[ms]$	16.003	29.984	44.512

Табела 18: Мерење размене података на претплату корисничких апликација са *Android C++* сервисом

		Подаци на претплату		
↓Број корисника		1 x 30fps	2 x 30fps	3 x 30fps
	Најмања вредност $[ms]$	0.843	0.678	0.459
	Највећа вредност $[ms]$	8.184	6.855	10.304
1	Средња вредност $[ms]$	1.665	1.835	1.755
	Најмања вредност $[ms]$	1.387	1.052	1.193
	Највећа вредност $[ms]$	13.940	13.699	14.309
5	Средња вредност $[ms]$	3.143	3.233	3.547
	Најмања вредност $[ms]$	1.001	1.793	2.006
	Највећа вредност $[ms]$	17.171	14.019	24.615
10	Средња вредност $[ms]$	4.495	4.22	6.848
	Најмања вредност $[ms]$	5.966	6.453	14.58
	Највећа вредност $[ms]$	22.508	57.117	393.689
30	Средња вредност $[ms]$	7.162	12.674	24.125
	Најмања вредност $[ms]$	8.316	40.314	70.91
	Највећа вредност $[ms]$	58.397	1839.41	3044.399
50	Средња вредност $[ms]$	14.213	72.889	114.417

Табела 19: Мерење размене података на претплату корисничких апликација са *Android Java* сервисом

		Подаци на претплату		
↓Број корисника		1 x 30fps	2 x 30fps	3 x 30fps
	Најмања вредност $[ms]$	1.768	1.641	2.071
	Највећа вредност $[ms]$	3.802	3.946	3.485
1	Средња вредност $[ms]$	3.127	3.46	3.131
	Најмања вредност $[ms]$	3.361	3.562	3.388
	Највећа вредност $[ms]$	17.18	16.906	17.302
5	Средња вредност $[ms]$	6.545	6.654	7.115
	Најмања вредност $[ms]$	6.76	9.445	10.389
	Највећа вредност $[ms]$	21.421	24.257	26.158
10	Средња вредност $[ms]$	9.307	11.091	14.804
	Најмања вредност $[ms]$	12.223	20.208	42.274
	Највећа вредност $[ms]$	44.464	132.489	181.021
30	Средња вредност $[ms]$	15.144	23.483	46.599
	Најмања вредност $[ms]$	16.514	46.917	152.24
	Највећа вредност $[ms]$	57.654	1873.807	3681.906
50	Средња вредност $[ms]$	20.514	95.092	173.844

Табела 20: Мерење размене података на претплату корисничких апликација са *Android Kotlin* сервисом

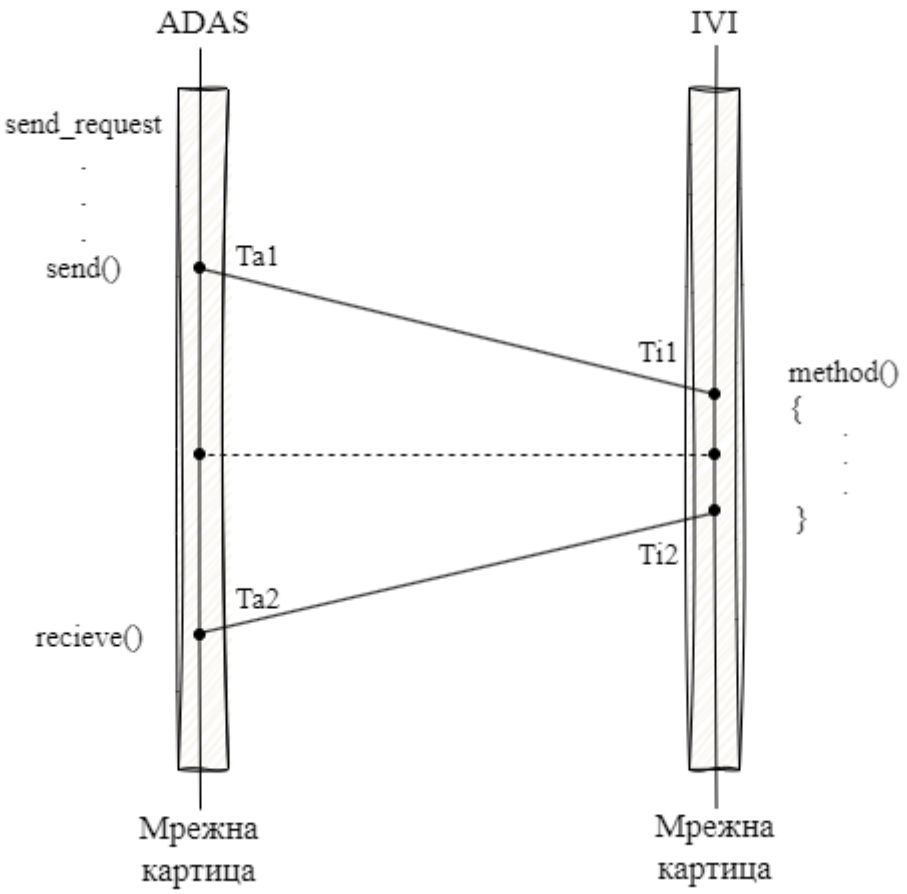
		Подаци на претплату		
↓Број корисника		1 x 30fps	2 x 30fps	3 x 30fps
	Најмања вредност $[ms]$	1.948	1.732	1.921
	Највећа вредност $[ms]$	4.005	3.848	
1	Средња вредност $[ms]$	2.997	3.281	3.762
	Најмања вредност $[ms]$	3.265	3.69	3.827
	Највећа вредност $[ms]$	17.682	16.854	15.322
5	Средња вредност $[ms]$	5.976	6.593	7.266
	Најмања вредност $[ms]$	7.536	9.45	10.269
	Највећа вредност $[ms]$	19.209	20.338	25.461
10	Средња вредност $[ms]$	10.184	12.061	14.725
	Најмања вредност $[ms]$	13.255	19.896	37.921
	Највећа вредност $[ms]$	45.967	194.157	162.55
30	Средња вредност $[ms]$	16.085	24.592	40.495
	Најмања вредност $[ms]$	17.398	32.142	77.903
	Највећа вредност $[ms]$	64.722	1346.589	2221.825
50	Средња вредност $[ms]$	19.432	102.549	164.265

је представљено на слици 38. Имплементирано је десет узастопних метода које започињу слање из *ADAS* домена при чему је убележено почетно време T_{a1}, прихватно време на *IVI* страни T_{i1}, време започињања одговора T_{i2} на страни *IVI* домена, као и време пристизања одговора на *ADAS* домену T_{a2}. Из ових временских ознака могуће је израчунати однос два часовника по формули

$$T_{a1} + \frac{(T_{a2} - T_{a1})}{2} = T_{i1} + \frac{(T_{i2} - T_{i1})}{2} \tag{4}$$

Овако добијен однос два часовника је коришћен за приказ добијених резултата изражених по референтном часовнику. Као референтни часовник је одабирана *ADAS* платформа ради поузданијег бележења мерења и једноставне могућности бележења времена са саме физичке архитектуре (односно мрежне картице) уместо времена контролисаног оперативним системом. Важно је напоменути да метода која је извршавана у сврху синхронизације има празну имплементацију, односно уводи занемарљиво кашњење и неуравнотеженост.

Из свих наведених мерења је могуће закључити да учинковитост решења задовољава услове реалног извршавања у случајевима преноса мањих података. За велике типове података у виду оквира са сензора камере високе резолуције и слично, као што је већ објашњено, *SOME/IP* протокол није погодан. За потребе ових случајева коришћења, испитан је и канал за пренос великих типова података као што приказано у табелама 21, 22 и 23 за *UDP, TCP* и *RTP* протоколе респективно. Могуће је приметити да је за велике податке (преко 50KB) очигледан пад у учинковитости услед потребе за безбедном и сигурном разменом која захтева серијализацију података, сегментацију како би се избегла *IP* фрагментација и могућ губитак података као и шифровање. Уколико су подаци већ



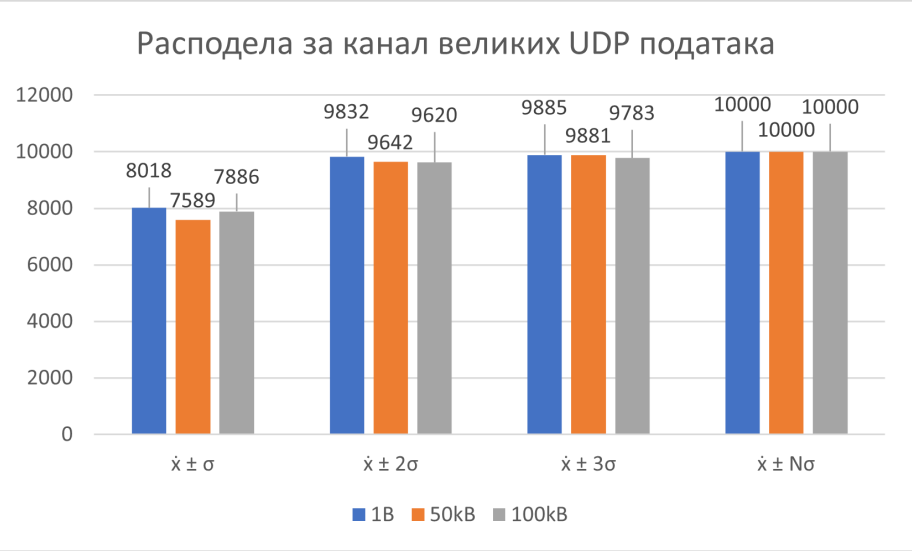
Слика 38: Синхронизација часовника

серијализовани то ће донети значајну уштеду времена. Додатно, ове механизме је могуће искључити приликом превођења имплементације чиме ће време потребно за пренос овако великих података бити драстично смањено - у зависности од величине податка може бити умањено за више од половине укупне вредности. За *RTSP* протокол није рађено додатно испитивање јер овај протокол у ствари користи у својој имплементацији *TCP* компоненте за размену иницијалних порука које учествују у руковању током података, односно *RTP* компоненте за пренос корисних података.

Стандардна девијација (σ) времена за размену великих података путем имплементираног канала износи 0.524, 5.543 и 7.018 за *UDP* размену података од 1B, 50kB и 100kB, односно 0.591, 9.049 и 6.97 за *TCP* и 0.5, 5,573 и 15,844 за *RTP* протокол респективно. Расподела која настаје овако добијеним стандардним девијацијама је за наведене протоколе дата на сликама 39, 40 и 41. У случају *UDP* протокола могуће је приметити да је највећи проценат (од преко ∼75% измерених узорака) налази у околини σ у односу на средњу вредност измерених узорака. Приближно 96% се налази у опсегу 2σ, односно ∼98% у опсегу 3σ. За *TCP* протокол су ти бројеви нешто дручачији и износе ∼64, ∼96 и ∼97 процената, а за *RTP* ∼66, ∼93, ∼95 у најгорем случају.

Табела 21: Мерење кашњења рада канала за пренос великих података - *UDP* протокол

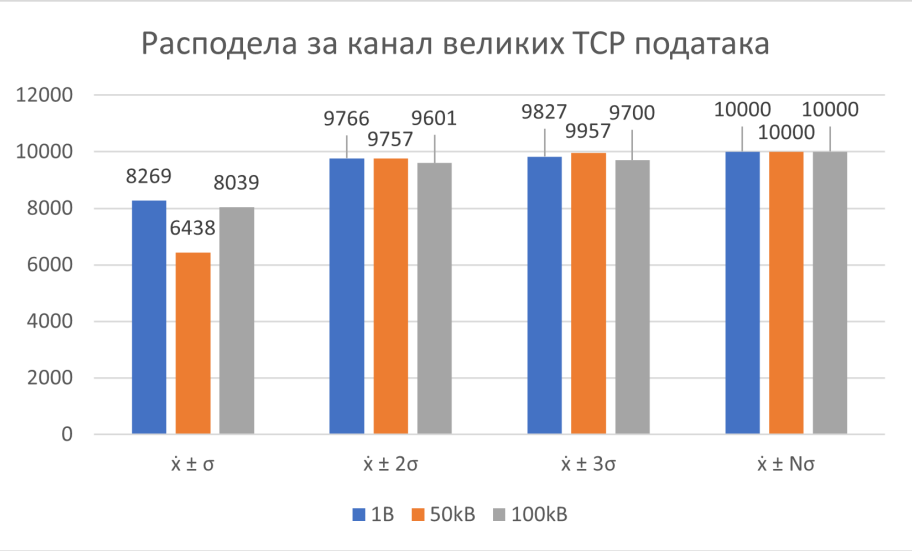
	1B Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$	50kB Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$	100kB Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$
Клијентска страна без сигурносних механизама	0.098 3.749 0.128	5.517 14.396 10.57	31.392 77.415 46.569
Клијентска страна са сигурносним механизмима	0.109 4.47 0.159	11.516 17.498 14.902	39.468 81.964 69.629
Серверска страна без сигурносних механизама	0.435 4.005 0.72	7.478 31.935 16.679	28.661 42.544 31.143
Серверска страна са сигурносним механизмима	0.485 5.773 0.753	15.82 35.466 22.703	38.737 78.531 49.175
Време на мрежи	2.932 4.83 3.653	32.085 52.136 40.401	86.303 112.015 97.959
Укупно	3.516 15.073 4.565	59.421 105.1 78.006	164.556 272.51 216.763
Подаци се без средњег слоја преносе преко мрежних утичница	2.774 7.169 3.638	44.009 54.5 47.675	/ / /



Слика 39: Расподела за канал великих *UDP* података

Табела 22: Мерење кашњења рада канала за пренос великих података - *TCP* протокол

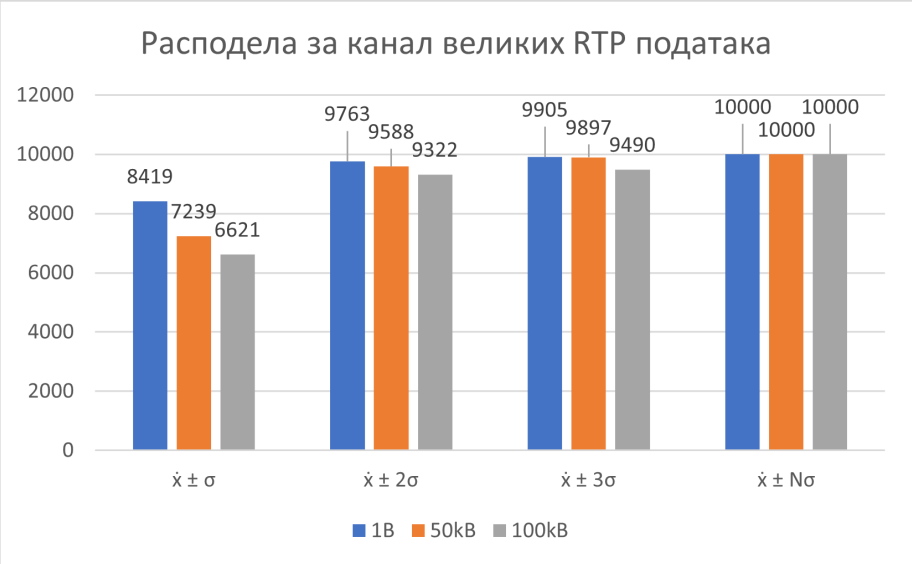
	1B Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$	50kB Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$	100kB Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$
Клијентска страна без сигурносних механизама	0.098 4.848 0.229	3.541 12.701 10.607	24.328 47.23 35.714
Клијентска страна са сигурносним механизмима	0.17 4.553 0.269	10.529 16.271 14.693	38.677 82.615 60.768
Серверска страна без сигурносних механизама	0.332 4.287 0.651	9.909 39.279 17.901	28.361 34.205 31.324
Серверска страна са сигурносним механизмима	0.501 6.034 0.779	14.291 33.633 23.822	38.849 82.564 60.746
Време на мрежи	3.709 9.126 4.593	44.047 70.842 49.3	97.109 337.561 110.333
Укупно	4.38 19.713 5.641	68.867 120.746 87.815	174.635 502.74 231.847
Подаци се без средњег слоја преносе преко мрежних утичница	3.52 7.851 4.613	52.227 61.608 56.798	/ / /



Слика 40: Расподела за канал великих *TCP* података

Табела 23: Мерење кашњења рада канала за пренос великих података - *RTP* протокол

	1B Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$	50kB Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$	100kB Најмања вр. $[ms]$ Највећа вр. $[ms]$ Средња вр. $[ms]$
Клијентска страна без сигурносних механизама	0.089 4.409 0.131	3.508 15.056 10.573	31.401 78.075 46.572
Клијентска страна са сигурносним механизмима	0.17 2.446 0.269	11.134 19.797 17.166	39.477 83.349 69.632
Серверска страна без сигурносних механизама	0.443 4.504 0.73	7.486 32.435 16.971	28.696 43.044 31.436
Серверска страна са сигурносним механизмима	0.54 5.244 0.749	11.496 46.83 26.067	38.772 79.033 54.175
Време на мрежи	2.809 4.928 4.092	37.418 52.896 41.968	123.865 229.789 137.818
Укупно	3.519 12.618 5.11	60.048 119.523 85.201	192.114 392.171 261.625
Подаци се без средњег слоја преносе преко мрежних утичница	2.896 6.249 4.155	52.227 61.608 56.798	/ / /



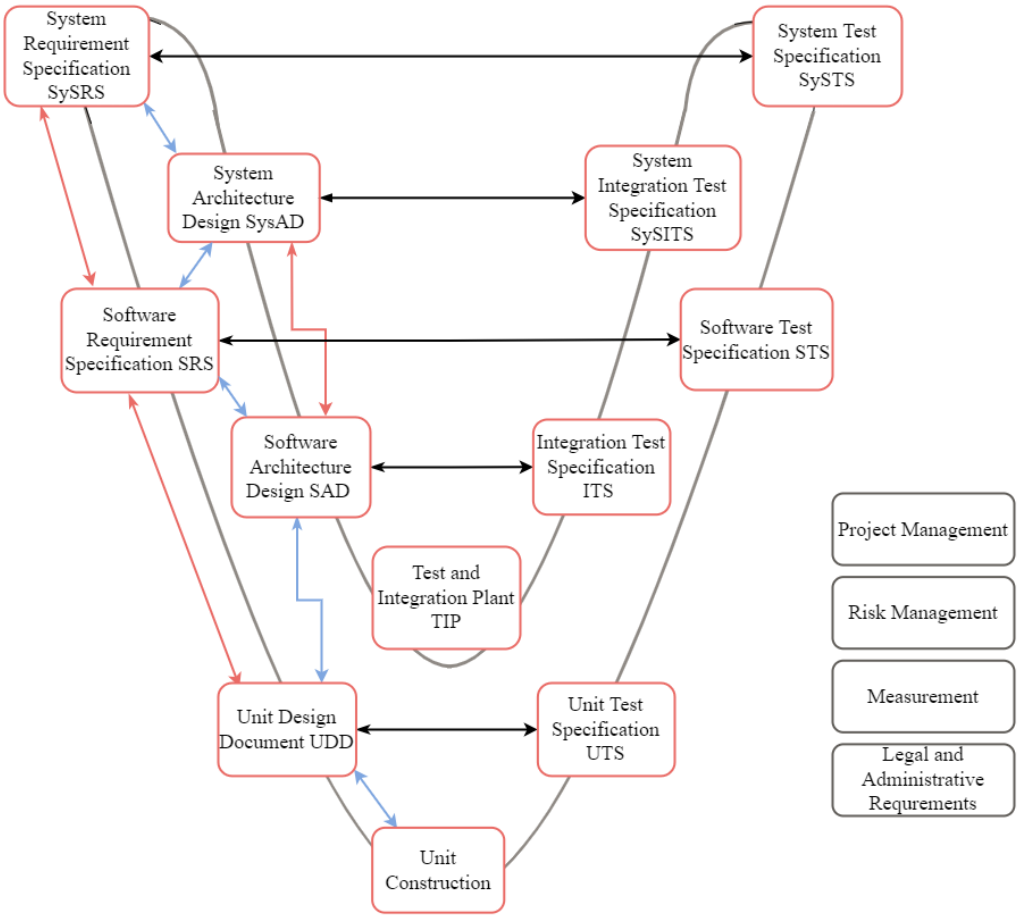
Слика 41: Расподела за канал великих *RTP* података

6.2 Испитивање рада алата

У грани развоја аутомобилских система, за потребе стварања програмске подршке није довољно само извршити многобројна тестирања са циљем верификације и валидације производа, већ од почетка развоја пројекта постоји фокус на процедурама и стандардима за начин његовог вођења и израде.

6.2.1 Процедура развоја пројекта

Стандардизован приступ вођења пројекта и развоја производа у аутомобилској индустрији представља извршавање процене по *Automotive Software Process Improvement Capability dEtermination - ASPICE* [154] режиму. *ASPICE* представља стандард за побољшање процеса и одређивање подобности програмске подршке у аутомобилу који за циљ има проналазак и дефинисање најбољих процеса како би се обезбедио највиши квалитет када је реч о развоју производа [197]. *ASPICE* је изграђен на моделу верификације и валидације, познатијем као *"V"* модел датом на слици 42. *ASPICE* стандард подразумева пет нивоа који означавају у којој мери је стандард испуњен [198]. С обзиром на то да ово решење није званично сертификовано, односно да није извршена званична процена, није утврђено који ниво је испуњен. Оно што је решењем било праћено су процеси развоја и планирања којима је потребно остварити следљивост и руковање ризицима.



Слика 42: Архитектура развоја и валидације пројекта по *"V"* моделу

Табела 24: Процена и руковање ризицима

	Значај последица			
	1 - Мали Регуларне потешкоће	2 - Средњи Кашњење у изведби	3 - Висок Темељна прерада функционалности	4 - Изузетан Производ постаје бескористан
Вероватноћа дешавања	**Додељен ризик**			
4 - Готово сигурно	7 - Средњи	11 - Висок	14 - Екстреман	16 - Екстреман
3 - Вероватно	4 - Низак	8 - Средњи	12 - Висок	15 - Екстреман
2 - Могуће	2 - Низак	5 - Средњи	9 - Средњи	13 - Висок
1 - Мало вероватно	1 - Низак	3 - Низак	6 - Средњи	10 - Висок

У процесу израде, пратећи *ASPICE* стандард, процена ризика (енгл. *Risk management*) је вршена на временском интервалу од 7 дана. Процена је вршена анализом штете тако што се у наведеном интервалу идентификују потенцијална дешавања која могу утицати негативно на ток развоја пројекта. Затим је за свако од дешавања додељена оцена од 1 до 4 која указује на учесталост појављивања, односно вероватноће да ће доћи до овог догађаја и оцена из истог опсега која указује на озбиљност последица, тј. утицај на пројекат. Из ових оцена се пратећи табелу 24 изводи укупна оцена коју носи уочени ризик, која говори о његовој приоритизацији међу осталим ризицима и начину праћења. Такође, за сваки од уочених ризика је неопходно било идентификовати радње које могу помоћи у његовој превенцији, као и конструисати план деловања уколико се ризик оствари. Табеле за руковање ризицима често садрже пет оцена уместо четири у случајевима када почетне оцене описују ризик занемарљивог значаја и ризик за који не постоји евидентирани проблем сличне природе. Међутим, ризици оцењени једном од ових оцена не буду класификовани као они за које је потребно праћење па се због једноставности приказа те оцене често уклањају.

Начин развоја и планирања који су предвиђени *ASPICE* стандардом захтевају потпуну следљивост. То значи да је неопходно документовати сваки корак који од анализе тржишта или захтева муштерија ствара захтеве израде високог нивоа. Ови захтеви су дељени по групама и из њих се најпре изводи системска архитектура, а потом и архитектура програмског решења. Како архитектуру чине модули чију је функционалност потребно остварити, да би њихова имплементација била ефикасно извршена они се описују јединицама имплементације којима се дефинише њихов задатак и оперативност. На тај начин је могуће пратити читаву путању којом је имплементација тестирана и верификована, а затим и валидирана тиме што свака јединица имплементације чини већу целину која суделује у архитектури решења којом су испуњени почетни захтеви што потврђује да је створени производ заиста оно што је тражено. За потребе вршења процене, све наведено се заводи у неколико основних докумената:

- *Software Development Plan (SDP)* - документ којим је представљена уопштена стра-

тегија, планови и приступи. Овде су назначени и фазе, циклуси и рокови предвиђени пројектом. Често се у оквиру овог документа дефинишу и зависности од употребљених производа треће стране, као и лиценцирања.

- *Requirements Specifications* - представља документ који пружа увид у пројектне захтеве вишег нивоа којима је дефинисано шта је тачно производ (захтеви специфични за систем се налазе у *System Requirement Specification*, а за имплементирану програмску подршку у *Software Requirement Specification* документу)

- *Software Architecture Design* - описује архитектуру програмске подршке, приказујући модуле који је чине и дефинишући њихову међусобну спрегу. Неретко се у оквиру истог, а некад и у оквиру додатног документа, дефинишу и сами модули, тј. компоненте и фунцкионалности јединица имплменетације (архитектура система се описује у одвојеном System Architecture Design документу, али са обзиром да дизајн на системском нивоу није био циљ овог пројекта, тај документ није покривен решењем)

- *Unit Design Document* - описује архитектуру сваког појединачног модула који је потребно имплементирати, са детаљном дефиницијом понашања, улазних података, повратних вредности и слично

- *Risk Management Plan* - документ у ком је описана процена свих ризика за израду пројекта. Овај документ садржи опис идентификовања, плана вођења, надгледања и руковања ризицима као и чланове колектива којима је ризик додељен на управљање

- *Configuration Management Plan* - дефинише алат за контролу верзије и начин на који ће бити коришћен као и остале алате који се користе у изради пројекта. Уколико у току развоја пројеката долази до промена у одлукама, руковање променама најчешће буде дефинисано кроз овај документ у спрези за основним *SDP* документом

- *Coding Standards and Guidelines* - дефинише правила и конвенције кодовања

- С обзиром на то да је развој рађен по *"V"* моделу, документи за развој решења треба да имају своје пандане у тестирању, па тако наспрам докумената за описивање захтева постоје *System Test Specification* и *Software Test Specification* документи, а наспрам докумената за опис архитектуре постоје *System Integration Test Specification* и *Software Integration Test Specification* документи

Тако су на почетку прикупљени захтеви тржишта у виду захтева на високом нивоу. Имајући у виду да је пројекат истраживачки, прикупљани захтеви нису оријентисани ка конкретној муштерији, већ представљају симбиозу заједничких потреба и наклоњености развоја читаве индустрије, односно значајних компанија и њихових истраживачких планова. Захтеви високог нивоа који усмеравају истраживање и одређују форму коју крајње решење, тј. производ треба да поседује и који су прикупљени анализом захтева тржишта су дати у наставку:

Захтев 1 - Портовање *Adaptive AUTOSAR* платформе на одговарајућу физичку архитектуру

Захтев 2 - Портовање *Android* оперативног система на одговарајућу физичку архитектуру

Захтев 3 - Анализа одговарајућег комуникационог протокола погодног за комуникацију ова два домена

Захтев 4 - Имплементација комуникације путем одабраног *SOME/IP* протокола

Захтев 5 - Анализа развоја вођеног моделом

Захтев 6 - Имплементација мета-модела за *AIDL* језик

Захтев 7 - Превођење између *ARXML* и *AIDL* модела

Захтев 8 - Превођење између *AIDL* и *FIDL* модела

Захтев 9 - Генерисање програмске подршке на *ADAS* страни

Захтев 10 - Генерисање програмске подршке за *SOME/IP* клијента на *IVI* страни

Захтев 11 - Генерисање програмске подршке за *Android* сервис на *IVI* страни

Захтев 12 - Сигурносни и безбедносни концепти

С обзиром на то да ово решење не зависи од физичке архитектуре, решење не покрива анализу системских захтева и системске архитектуре. Стога је, искључиво ради поштовања читаве процедуре развоја пројекта, дато свега неколико тривијалних системских захтева:

Системски захтев 1 - *ADAS* платформа, подршка за *POSIX* 51

Системски захтев 2 - *ADAS* платформа, подршка за сензоре попут камере

Системски захтев 3 - *Ethernet* подршка за *ADAS* и *IVI* платформе

Системски захтев 4 - *IVI* страна, подшка за екран

Што се тиче захтева за програмску подршку (у наставку ПП захтеви) који проистичу из претходно наведених захтева, заведени су:

ПП захтев 1 - Анализа и дефинисање комуникационог протокола погодног за међудоменску комуникацију

ПП захтев 2 - Превођење *Adaptive AUTOSAR* програмског стека на тестну платформу

ПП захтев 3 - Моделовање тестних компоненти на *ADAS* страни

ПП захтев 4 - Имплементација тестних компоненти на *ADAS* страни

ПП захтев 5 - Моделовање примера посредничке компоненте на *ADAS* страни у случају централизованог приступа

ПП захтев 6 - Имплементација примера посредничке компоненте на *ADAS* страни у случају централизованог приступа

ПП захтев 7 - Имплементација комуникације преко одабраног протокола (подразумева и анализу употребе модула средњег слоја нпр. *ara::com*, *CommonAPI*, итд. на *IVI* страни)

ПП захтев 8 - Имплементација тестних компоненти на *IVI* страни (апликативни ниво *Android* оперативног система)

ПП захтев 9 - Имплементација тестних компоненти на *IVI* страни (ниво апстракције физичке архитектуре *Android* оперативног система)

ПП захтев 10 - Имплементација посредничког у случају централизованог или више посредничких *Android C++* сервиса у дистрбуираном приступу

ПП захтев 11 - Имплементација посредничког у случају централизованог или више посредничких *Android Java/Kotlin* сервиса у дистрбуираном приступу

ПП захтев 12 - Имплементација безбедносних механизама у комуникацији

ПП захтев 13 - Имплементација сигурносних механизама у комуникацији

ПП захтев 14 - Анализа употребе развоја вођеног моделом и одређивање радног оквира погодног за развој

ПП захтев 15 - Имплементација *AIDL* мета-модела језика за дефиницију спреге у *Android* системима

ПП захтев 16 - Превођење између *ARXML* и *AIDL* модела

ПП захтев 17 - Превођење између *AIDL* и модела посредничког средњег слоја (нпр. *CommonAPI*)

ПП захтев 18 - Генерисање програмске подршке страна које комуницирају

ПП захтев 19 - Генерисање *Android C++* сервиса који прослеђује податке до крајњих корисничких апликација

ПП захтев 20 - Генерисање *Android Java/Kotlin* сервиса који прослеђује податке до крајњих корисничких апликација

ПП захтев 21 - Генерисање безбедносних механизама у комуникацији

ПП захтев 22 - Генерисање сигурносних механизама у комуникацији

ПП захтев 23 - Генерисање тестова

Из набројаних захтева су креиране архитектуре, најпре системска архитектура решења која се у оквиру овог рада практично сведе на тестно окружење и коришћење доступних ресурса.

Затим архитектура појединачно за сваки од захтева програмске подршке. Архитектуре су дате кроз решење представљено овим радом у поглављима 4 и 5. Из ових архитектура издвојене су јединице имплементације које граде архитектуру програмске подршке и извршавају функционалности које је потребно остварити. Јединице имплементације представљају све појединачне компоненте и функције које су описане решењем.

6.2.2 Тестирање имплементације програмске подршке за комуникацију

Упоредо са имплементацијом рађено је и тестирање. С обзиром на то да је програмска подршка најпре имплементирана, а потом и генерисана, тим редом је рађено и тестирање. Тестирање имплемнетиране програмске подршке је вршено на три нивоа: функционални, интеграциони и тестови јединица имплементације. Број и распоред имплементираних тестова је дат у табели 25, 26 и 27. Тестовима јединица имплементације су тестиране функције у решењу описаних компонената укључујући посредничке компоненте на *ADAS* страни, *SOME/IP* клијентске компоненте и компоненте *Android* сервиса на *IVI* страни и компоненте како клијентске тако и серверске стране канала за велике податке типски покривајући различите случаје:

- исправност иницијализацијског дела
- неправилне уносе

- граничне вредности
- позиве метода у неиницијализованим стањима
- позиве метода у правилно иницијализованим стањима
- правилне уносе
- исправност деиницијализацијског дела

Табела 25: Тестирање имплементиране комуникације путем *SOME/IP* протокола

Ниво тестирања	**Тестирана категорија**	**Број тестова**
Тестирање јединица	иницијализација клијента/сервиса на IVI страни	15
	неправилни уноси метода на IVI страни	28
	употреба граничних вредности на IVI страни	22
	позиви метода на IVI страни у неиницијализованим стањима	20
	позиви метода на IVI страни у правилно иницијализованим стањима	20
	исправни уноси метода за на IVI страни	20
	исправност деиницијализације на IVI страни	15
	исправност иницијализације на ADAS страни	25
	неправилни уноси метода на ADAS страни	35
	употреба граничних вредности на ADAS страни	42
	позиви метода на ADAS страни у неиницијализованим стањима	32
	позиви метода на ADAS страни у правилно иницијализованим стањима	14
	исправни уноси метода на ADAS страни	21
	исправност деиницијализације на ADAS страни	25
Интеграциони тестови	Метода	60
	Поље за добављање вредности	20
	Догађај	40
	Поље за постављање вредности	20
	Поље за обавештавање о промени вредности	20
Функционални тестови	Метода	4
	Поље за добављање вредности	5
	Догађај	6
	Поље за постављање вредности	2
	Поље за обавештавање о промени вредности	2
Укупно		513

Тестовима јединица имплементације је тестирана искључиво исправност изолованог рада компонената, у потпуности занемарујући међусобну интеракцију са другим компо-

Табела 26: Тестирање имплементације *Android* сервиса (за све подржане програмске језике)

Ниво тестирања	**Тестирана категорија**	**Број тестова**
Тестирање јединица	иницијализација сервиса	15
	иницијализација базена нити	15
	деиницијализација сервиса	15
	деиницијализација базена нити	15
	методе за добављање binder објекта	15
	неправилни уноси метода за размену података	27
	употреба граничних вредности	21
	исправни уноси метода за размену података	21
	методе за претплату и прекид претплате	45
	неправилно коришћење повратних метода од стране корисника	27
	методе за приоритизацију претплатника	15
	исправност деиницијализације на IVI страни	15
Интеграциони тестови	Добављање binder објекта	50
	Метода за размену података	50
	Претплата на информацију	50
	Дистрибуција података до корисника	50
Функционални тестови	Добављање binder објекта	45
	Метода за размену података	45
	Претплата на информацију	45
	Дистрибуција података до корисника	45
Укупно		626

нентама. Уколико у оквиру тестираних функција постоји интеракција са спољним компонентама или системом ова интеракција је опонашана креирањем лажних објеката (енгл. *Mock objects*). За креирање лажних објеката су, у зависности од програмског језика, коришћени *gMock* и *Mockito* за *C++* и *Java/Kotlin* језике, док су за саме тестове коришћена окружења *gTest* и *JUnit* респективно. На овај начин је тестирана искључиво исправност имплементације и очекивано понашање у различитим сценаријима.

Са друге стране, интеграционим тестовима је тестирана и спрега међу компонентама. Главна спрега је комуникација између два домена путем *SOME/IP* протокола чиме је тестиран сваки од механизама појединачно. Додатно, рађени су и интеграциони тестови за спрегу коју пружа канал за пренос великих података где је тестирана комуникација за сваки подржани транспортни протокол појединачно. У случају *Android* сервиса тестирана је и спрега са интегрисаним *SOME/IP* компонентама као и дистрибуција података до корисничких апликација. За потребе извршавања интеграционих тестова су првобитно задаване очекиване вредности ручно за сваки покренути тест, а потом је имплементирана и тестна компонента на *ADAS* страни која шаље исте вредности као и регуларна спрега до *IVI* домена како би потврда о исправности податка била двострука и како вредности који-

ма се врши тестирање не би морале да буду експлицитно задате, већ је довољно извршити проверу једнакости података пристиглих регуларном спрегом и путем тестне компоненте (овај аспект је значајан због омогућавања генерисања тестова).

Табела 27: Тестирање имплементираног канала за за размену великих података

Ниво тестирања	**Тестирана категорија**	**Број тестова**
Тестирање јединица	иницијализација сваког појединачног модула који чини спрегу канала	14
	неправилни уноси метода за слање и пријем тока података	14
	употреба граничних вредности при слању и пријему тока података	16
	методе у неиницијализованим стањима	20
	методе компоненте за сегментацију и унификацију	12
	методе компоненте за серијализацију и десеријализацију	12
	методе компоненте за шифровање и дешифровање	12
	исправни уноси метода за слање и пријем тока података	16
	прослеђивање позива компоненте за повезивање до транспортног поротокола	20
	руковање мрежним утичницама	15
	руковање слањем на нивоу транспортног протокола	20
	руковање пријемом на нивоу транспортног протокола	15
	руковање повратном функцијом којом стижу подаци кроз клијентску спрегу до корисничке апликације	15
Интеграциони тестови	слање великог тока података	35
	пријем великог тога података	35
	стварање вишеструког броја инстанци клијената и сервиса	20
Функционални тестови	слање великог тока података	6
	пријем великог тога података	6
Укупно		303

На самом крају су извршени и функционални тестови. Функционалним тестовима је потврђено да целокупна комуникација програмске подршке ради на примерима из реалних сценарија. Основни случајеви употребе којима је демонстриран рад решења и који су коришћени за функционалне тестове као и за мерење потрошње ресурса су следећи:

- Визуелно представљање околине возила у *IVI* домену на основу перципираних сао-

браћајних трака, других учесника у саобраћају и саобраћајних знакова детектованих у *ADAS* домену (подаци са *ADAS* домена се у овом случају коришћења шаљу путем обавештавајућих *SOME/IP* механизама у тренуцима њиховог опажања). Подаци су били различитих типова: целобројни тип, низ целобројног типа, структуре и угњеждене структуре као и низови оваквих структура, а интервал извршавања размене података је пратио учесталост пристизања нових података са сензора камере од 35 оквира у секунди које служе као улазни подаци за екстраховање корисних информација. За потребе вршења тестирања је креирана корисничка апликација у *Android* систему и употребљен је *Godot Engine* оквир за визуелизацију и креирање *3D* окружење како би интеграција на највишем апликативном нивоу била потврђена

- Обављање видео позива у *IVI* домену коришћењем удаљене камере, односно камере за надгледање будности и пажње возача (у овом случају коришћења *SOME/IP* је употребљен тако да се методом, на захтев за употребу камере започне слање великог тока података путем канала за велике податке). Учесталост бележења података са камере је 35 оквира у секунди, а резолуција видео записа је 1280x720. За потребе вршења тестирања је коришћен постојећи модул апстракције физичке архитектуре из *Android* система како би интеграција на нижем нивоу такође била потврђена.

- Коришћење камера за 360 степени приказ околине возила из *ADAS* домена у *Android* сервисима проширене визије (енгл. *Extended View System*) креираним од стране компаније *Google* који типично захтевају четири камере прикључене непосредно на *IVI* систем (у овом случају коришћења *SOME/IP* је употребљен тако да се методом, на захтев за употребу камера започне слање тока података путем канала за велике податке). Учесталост бележења нових података за сваку од четири камере појединачно је 35 оквира у секунди, а резолуција 1280x720.

- Обавештавање клијената у *IVI* домену о будности и стању концентрације возача на основу података обрађених у *ADAS* домену. Тип податка за пренос је целобројни, а слање стохастично без предвиђене периодичности путем поља за обавештавање.

- Добављање типичних података о возилу и његовом стању попут вредности притиска у гумама (подаци се у овом случају са *ADAS* домена добављају путем *SOME/IP* механизама поља на захтев). Типови података за пренос су целобројни, а слање је извршавано на принципу поља за добављање вредности, без предвиђене периодичности.

- Обавештење клијената у *IVI* домену у случају отказивања фунцкионалности уочене у *ADAS* домену (пример је, такође, лош притисак у гумама). Коришћене су тестне компоненте из претходне тезе, па и типови података за пренос остају целобројне вредности, а слање је механизмом поља за постављање вредности и нема предвиђену периодичност.

Табела 28: Мерење заузећа ресурса на примерима функционалних тестова

		заузеће мемрије [%]	заузеће процесора[%]
ServiceProxy ADAS		15.0	3.0
Android сервис	C++	0.4	4.1
	Java	1.5	22.6
	Kotlin	1.6	23.1

*за мерење је коришћена Alpha AMV на ADAS и Qualcomm SA8155P платформа на IVI страни

6.2.3 Тестирање имплементираног генератора

Након имплементиране програмске подршке којом је извршена комуникација, потребно је тестирати и имплементирани алат за генерисање. Тестирање алата се састоји из тестирања неколико различитих ствари. Најпре тестирање мета-модела, затим тестирање превођења међу моделима и напослетку тестирање генерисања самог кода програмске подршке. Тестирање мета-модела и генерисања кода се састоји из писања тестова који проверавају исправност добијеног модела на основу учитане *.aidl* датотеке (као и сачуване *.aidl* датотеке на основу задатог модела), односно исправност генерисаног кода на основу задатих модела. У случају тестирања превођења међу моделима није довољно проверити да ли је од задате датотеке једног језика за дефинисање спреге настала исправна датотека другог језика за дефинисање спреге [199]. Разлог је то што постоји могућност да дође до двоструке неправилности која има лажно правилан исход. То значи да је могуће, на пример, да учитавање буде неисправно тако што је елемент А протумачен као елемент Б, а да одмах потом и превођење буде неиспрано тако што је елемент Б преведен у елемент А и на тај начин прикаже лажно исправно превођење које у овом случају није могуће детектовати, али у неком другом случају може бити узрок неправилности. Да би потврда превођења била извршена искључиво на основу задатих и генерисаних датотека потребан број узорака би морао покрити све могуће случаје, што је у пракси немогуће. Зато је неопходно превођење додатно тестирати кроз етапе, где је први степен тестирање самих функција за превођење задајући изоловане елементе. Други степен подразумева тестирање задајући објекте модела њиховим ручним стварањем (имплементацијом, а не учитавањем) и проверавајући превођењем добијене објекте модела, а трећи задајући улазне датотеке и проверавајући превођењем добијене датотеке.

6.2.3.1 Тестирање *AIDL* мета-модела Као што је већ речено, најпре је тестиран мета-модел *AIDL* језика за дефинисање спреге. Мета-модел је већински тестиран тестовима за јединице имплементације којима се задају дириговани елементи *AIDL* језика

- анотације
- константе
- *Parcelable* типови

- набројиви типови
- спреге
- пакети
- форматирање
- валидација

и тестира се очекивана интерпретација као што је приказано у табели 29. За интеграционе тестове су једино имплементирани тестови за процес валидације који се врши као финална фаза учитавања модела и овом приликом су за улазе коришћене стварне *.aidl* датотеке са циљем да валидација целокупног модела, а не само његових засебних елемената буде потврђена. Функционални тестови нису имплементирани. Случај тестирања имплементираног *AIDL* мета-модела је специфичан због тога што је дубоко повезан са коришћењем *EMF* радног оквира који заправо врши почетно парсирање карактера из *.aidl* датотеке и то резултује непогодним раздвајањем функционалних, интеграционих и тестова за јединицу имплементације чиме они бивају најчешће обједињени [199]. Након што је потврђено исправно учитавање, интерпретација, валидација и чување *AIDL* модела, отпочињу фазе тестирања превођења.

Табела 29: Тестирање имплементираног *AIDL* мета-модела

Ниво тестирања	**Тестирана категорија**	**Број тестова**
Тестирање јединица	анотације	15
	константе	15
	Parcelables	30
	набројиви типови	5
	спреге	50
	пакети	10
	форматирање	80
	валидација	55
Интеграциони тестови	валидација	55
Укупно		315

6.2.3.2 Тестирање превођења између *ARXML* и *AIDL* модела Превођење између *ARXML* и *AIDL* модела је, као и имплементација комуникације, тестирано на три нивоа у виду функционалних, интеграционих и тестова за јединице имплементације као што је приказано у табели 30. Тестови за јединице имплеметације покривају руковање функцијама за превођење, односно потврђују исправност њихове имплементације, не вршећи валидацију производа превођења. Овиме су тестиране функције превођења:

- за руковање моделом (покрива и учитавање спољашњих, тј међудоменских спрежних пролаза и мапирања уколико је потребно)

- пакета компонената програмске подршке, комуникационих спрега и задатих типова података који се преносе
- *ARXML* компоненте која обухвата спрежне пролазе у главну *AIDL* спрегу покривајући и случаје компоненте која не садржи спрежне пролазе (подразумева и проверу исправности назива који ће имати новонастала датотека)
- комуникационих спрега које садрже све комбинације *SOME/IP* механизама укључујући и празне спреге (подразумева и проверу исправности назива који ће имати новонастала датотека)
- за руковање механизмима који омогућавају укључивање и повезивање међусобно зависних делова и које се своди на исправно тумачење пакета за различите случаје (укључивање главне спреге, додатних спрега за догађаје, сложених типова и сл.)
- метода из комуникационе спреге покривајући бесповратне методе, методе са једним и више улазних и излазних параметара
- догађаја из комуникационе спреге које се највише односе на проверу функција задужених за настанак додатне *AIDL* спреге која садржи повратну методу поред регуларне методе која служи за претплату
- поља из комуникационе спреге које је тестирано на исти начин као и методе и догађаји с тим да поља немају аргументе као што је случај са методама, па се у том случају тестирања имплемнетационих јединица своде на провере да ли је конвенција именовања насталих метода исправно изведена
- аргумената које подразумева проверу исправности створеног назива, типа и смера (улазни, излазни или улазно-излазни аргумент)
- типова података које подразумева проверу исправности мапирања свих основних типова (подразумева и случаје превођења типова који нису усклађени као и могућност непосредне доделе вредности)
- сложених типова података које подразмевају проверу исправности свих сегмената новонасталог *AIDL* модела, подразумева и проверу угњеждених сложених података као и сигурносну активацију
- набројивих елемената
- константи које подразумва проверу исправности настанка константи за сваки могући прости тип са фокусом на покушај задавања вредности ван опсега

За разлику од тестова имплементационих јединица које проверавају исправност функција за превођење позивајући их у диригованим условима дајући различите улазне параметре, интеграциони тестови користе стварне *ARXML* моделе и на основу задатих *ARXML* модела проверају исправност добијеног *AIDL* модела кроз исте одлике наведене и у тестовима јединица имплементације. Функционални тестови су имплементирани један корак

даље у погледу ширине обухватања операција превођења и своде се на свеобухватне провере које почињу од задатих *.arxml* датотека и проверавају исправност свих *.aidl* датотека које је створио генератор. Интеграционим и функионалним тестовима су покривени случаји коришћења са једном или више компонената, са вишеструким спрегама и вишеструким инстанцама истоветних спрега, свим комуникационим механизмима, комбинацијама аргумената, типова података и слично.

Табела 30: Тестирање превођења између *ARXML* и *AIDL* модела

Ниво тестирања	**Тестирана категорија**	**Број тестова**
Тестирање јединица	руковање креирањем модела	20
	превођење пакета	10
	превођење ARXML компоненте у главну AIDL спрегу	7
	превођење SOME/IP комуникационих спрега	15
	руковање учитавањима и зависности међу пакетима	10
	превођење SOME/IP метода	15
	руковање аргументима	6
	превођење SOME/IP догађаја	15
	превођење SOME/IP поља	18
	прости типови података	12
	сложени типови података	17
	набројиви елементи	9
	константе	20
Интеграциони тестови	превођење читавих модела	74
Функционални тестови	учитавање, превођење и чување читавих датотека модела	18
Укупно		256

6.2.3.3 Тестирање превођења између *AIDL* и *FIDL* модела Истоветно као што је тестирано претходно превођење и за ову фазу превођења су коришћени функционални, интеграциони и тестови јединица имплементације као што је приказано у табели 31. Тестирани су:

- руковање моделом (покрива и учитавање спољашњих, тј међудоменских спрежних пролаза и мапирања уколико је потребно)
- превођење пакета како главног тако и регуларних *AIDL* спрега, као и сложених типова
- превођење спрега које подразумева препознавање различитих инстанци које припадају истој спрези и које покривају различите комбинације комуникационих механизама

- превођење метода
- превођење догађаја (такође са нагласком на исправно упаривање метода регуларних спрега које служе за претплату и додатних спрега које садрже повратну методу)
- превођење поља (укључује и проверу онемогућавања прослеђивања аргумената јер је то карактеристично само за методе, не и за поље)
- превођење аргумената
- превођење типова које проверава враћање типских неслагања, различите вредности на које су променљиве иницијализоване, сигурносну активацију, просте и сложене типове (укључујући и угњеждене)

Интеграциони тестови потврђују целокупан процес тиме што кроз комбинације сценарија коришћених у тестовима јединица имплементације проверавају да ли је читав модел који је добијен превођењем исправан. Односно, у случају функционалних тестова да ли су генерисане датотеке исправне.

Табела 31: Тестирање превођења између *AIDL* и *FIDL* модела

Ниво тестирања	**Тестирана категорија**	**Број тестова**
Тестирање јединица	руковање креирањем модела	12
	превођење пакета	18
	превођење SOME/IP комуникационих спрега	17
	превођење SOME/IP метода	9
	руковање аргументима	11
	превођење SOME/IP догађаја	9
	превођење SOME/IP поља	16
	прости типови података	12
	сложени типови података	17
	набројиви типови	9
Интеграциони тестови	превођење читавих модела	74
Функционални тестови	учитавање, превођење и чување читавих модела	37
Укупно		241

6.2.3.4 Формална верификација превођења међу моделима Поред тестирања имплементираног генератора на описан начин, од почетка развоја је рађена и формална верификација превођења међу моделима. Формална верификација је неопходна како би модели превођења били апстраховани и како би ток генерисања био дефинисан [200]. Примена формалне верификације у овом решењу се састоји из три корака:

1. Прикупљање захтева, које је урађено у почетној фази како би и процедура развоја *ASPICE* стандардом била испоштована. У овом случају, резултат прикупљених захтева претходи анализи језика за дефинисање спрега и њиховом семантичком мапирању.

2. Стварање модела превођења које представља стања и кораке у самом превођењу. Модели за превођење су, такође, представљени блок дијаграмима у поглављу 5.

3. Математичка дефиниција. Један од најзначајанијих доприноса математике у рачунарској науци је могућност прецизног дефинисања елемената и стања. Формална верификација омогућава апстракцију неопходну за дефинисање превођења математичким путем [201]. Стога су најпре дефинисане граматике језика за дефинисање спрега, а потом и превођења између *ARXML* модела у *AIDL*, *AIDL* модела у *FIDL*, као и *ARXML* модела у *FDEPL*. Овај корак формалне верификације је дат у наставку.

Граматика сваког од језика за дефинисање спреге је дефинисана симболима и продукцијама, баш као што је то случај са описом граматике програмских преводилаца [202, 203]. Тако су дефинисани терминални симболи, који у овом случају представљају кључне карактере и речи које дефинишу крајње симболе, којима тумачење реченица из граматике добија смисао. Уколико би граматика била поређена са дрветом, терминални симболи су листови на гранама. Поред терминалних, дефинисани су и нетерминални симболи који представљају семантички ниво изнад терминалних и који настају комбинацијом других терминалних и нетерминалних симбола. У поређењу граматике са дрветом, нетерминални симболи представљају гране. Само дефинисање симбола језика за дефинисање спреге приказаних табелама 32, 33, 34 и 35 није довољно да би граматика била јасно описана. Зато су дефинисана и правила продукција којима је описан однос и редослед симбола такав да дефинише тумачење реченица и појмова конструисаних одговарајућим језиком. Почетни симбол је за све језике сам модел (он често не бива сврстан у нетерминалне симболе већ је издвојен као почетни симбол, али је због прегледности и јасноће приказа одлучено да овде то буде случај).

У сврху омогућавања боље прегледности и јасноће, у самим продукцијама су употребљене три најчешће коришћене додатне ознаке које упрошћавају опис граматике (без њиховог коришћења, било би потребно користити значајно више продукција за опис истоветних појмова):

- (*) → Клејнијева звезда (енгл. *Kleene star*) која индикује да симбол након ког је постављена звезда не мора бити присутан у продукцији, а уколико јесте, може бити присутан једном или више пута
- ([]) → угласте заграде које индикују опционо појављивање симбола који обухватају, тј. наглашавају да симбол не мора бити присутан, а уколико јесте, може бити пристутан само једном у одговарајућој продукцији
- (|) → усправна линија која индикује могућност појављивања искључиво једног од два симбола између којих је постављена

Граматика *AIDL* језика

У табели 32 биће приказани терминални симболи *AIDL* језика на основу техничке документације [99] и нетерминални симболи који представљају логичку структуру терминалних симбола потребну за описивање граматике. Након табеле у наставку су дате и продукције, односно правила која дефинишу нетерминалне симболе, односно граматику.

Табела 32 - Симболи *AIDL* језика

Тип симбола	Назив симбола	Представа симбола
Терминални	lparen	"("
	rparen	")"
	lbrace	"{"
	rbrace	"}"
	lbrack	"["
	rbrack	"]"
	lcomp	"<"
	rcomp	">"
	ann_sym	"@"
	null	"nullable"
	back	"Backing"
	utf8	"utf8InCpp"
	vintf	"VintfStability"
	desc	"Descriptor"
	app_usage	"UnsupportedAppUsage"
	java_stable	"JavaOnlyStableParcelable"
	comma	","
	eq	"="
	quote	"""
	arrow	"→"
	semicol	";"
	imp_id	"import"
	pack_id	"package"
	parc_id	"parcelable"
	iface_id	"interface"
	in_id	"in"
	out_id	"out"
	inout_id	"inout"
	ow_id	"oneway"
	var_id	називи идентификатора (променљиве, спреге, методе, ...)
	byte	"byte"
	int	"int"
	char	"char"
	long	"long"
	bool	"boolean"
	double	"double"
	float	"float"
	string	"String"
	map_id	"Map"
	list_id	"List"
	enum_id	"enum"

Наставак на следећој страни

Табела 32 - Наставак

	const_id	"const"
Нетерминални	AIDL_MODEL PACK IMP IFACE_ANN IFACE PARC PARC_FIELD PARC_ANN METHOD ARG TYPE TYPE_ID MAP LIST ENUM ENUM_FIELD NULL_ANN UTF8_ANN VINTF_ANN APP_USAGE_ANN JAVA_STABLE_ANN DESC_ANN BACK_ANN CONST CONST_FIELD	

Правила продукције за *AIDL* језик су дата у наставку.

Најпре су ту правила којима отпочиње стварање *AIDL* модела. Њима је модел подељен на основне целине, тј. описано је који елементи на вишем нивоу учествују у стварању модела. Затим ће сваки од ових елемената бити детаљно разложен.

AIDL_MODEL → PACK
AIDL_MODEL → PACK IMP
AIDL_MODEL → PACK [IFACE_ANN] IFACE
AIDL_MODEL → PACK [PARC_ANN] PARC
AIDL_MODEL → PACK CONST
AIDL_MODEL → PACK ENUM
AIDL_MODEL → PACK IMP [IFACE_ANN] IFACE
AIDL_MODEL → PACK IMP [PARC_ANN] PARC
AIDL_MODEL → PACK IMP CONST
AIDL_MODEL → PACK IMP ENUM

Првобитно се разлажу најједноставнији елементи модела, а то су елементи за дефинисање пакета и укључивања зависности од осталих модела. Ове елементе је могуће већ у овом моменту описати терминалним симболима чиме се може потврдити њихово постојање.

PACK → pack_id var_id semicol
IMP → imp_id var_id semicol

Потом следи дефинисање продукција за стварање спрега, тачније једне спреге по моделу. Спрега је сложенији елемент и састоји се из више нетерминалних симбола, за које су такође написане продукције до нивоа коришћења терминалних симбола.

IFACE_ANN → VINTF_ANN | APP_USAGE_ANN | DESC_ANN
IFACE → iface_id var_id lbrace METHOD* rbrace
METHOD → [ow_id] [NULL_ANN] TYPE var_id lparen ARG* BPAREN semicol

Поред спрега, у моделима се могу дефинисати и други сложени елементи попут *Parcelable* елемената који су такође описани вишеструким нетерминалним симболима.

PARC_ANN → VINTF_ANN | APP_USAGE_ANN | JAVA_STABLE_ANN
PARC → parc_id var_id lbrace PARC_FIELD* rbrace
PARC_FIELD → [NULL_ANN] TYPE var_id semicol
ARG → [in_id] | [out_id] | [inout_id] [NULL_ANN] TYPE var_id

Сложени елементи модела, уколико нису празни напослетку дођу до самих типова података. Подржани типови су претходно у решењу рада обрађени, а овде је приказано како продукције прате управо то што је представљено.

TYPE → int
TYPE → byte
TYPE → char
TYPE → long
TYPE → bool
TYPE → double
TYPE → float
TYPE → [UTF8_ANN] string
TYPE → MAP
TYPE → LIST
TYPE → PARC
TYPE → TYPE_DEF
TYPE_DEF → ENUM
MAP → map_id lcomp TYPE comma TYPE rcomp var_id
LIST → list_id lcomp TYPE rcomp var_id
ENUM → [BACK_ANN] enum_id var_id lbrace ENUM_FIELD* rbrace
ENUM_FIELD → var_id comma
TYPE → TYPE lbrack rbrack
CONST → const_id CONST_FIELD
CONST_FIELD → int | string var_id eq var_id

AIDL модел може да садржи и разне анотације за које су такође дефинисане продукције.

BACK_ANN → ann_sym back lparen TYPE_ID eq quote TYPE quote
UTF8_ANN → ann_sym utf8
VINTF_ANN → ann_sym vintf
APP_USAGE_ANN → ann_sym app_usage
JAVA_STABLE_ANN → ann_sym java_stable
DESC_ANN → ann_sym desc

Граматика *FIDL* језика

У табели 33 биће приказани терминални симболи *FIDL* језика на основу техничке документације [104] и нетерминални симболи који представљају логичку структуру терминалних симбола потребну за описивање граматике. Након табеле у наставку су дате и продукције, односно правила која дефинишу нетерминалне симболе, односно граматику.

Табела 33 - Симболи *FIDL* језика

Тип симбола	Назив симбола	Представа симбола
Терминални	lparen	"("
	rparen	")"
	lbrace	"{"
	rbrace	"}"
	lbrack	"["
	rbrack	"]"
	lcomp	"<"
	rcomp	">"
	comma	","
	eq	"="
	arrow	"→"
	semicol	";"
	imp_id	"import"
	pack_id	"package"
	iface_id	"interface"
	meth_id	"method"
	ff_id	"fireAndForget"
	broad_id	"broadcast"
	att_id	"attribute"
	nr_id	"noRead"
	ro_id	"readonly"
	ns_id	"noSubscriptions"
	in_id	"in"
	out_id	"out"
	inout_id	"inout"
	ver_id	"version"
	major_id	"major"
	minor_id	"minor"
	var_id	називи идентификатора (променљиве, спреге, методе, ...)
	tc_id	"typeCollection"
	int	"Integer"
	int8	"Int8"
	int16	"Int16"
	int32	"Int32"
	int64	"Int64"
	uint32	"UInt8"
	uint32	"UInt16"
	uint32	"UInt32"
	uint64	"UInt64"
	char	"char"
	long	"Long"

Наставак на следећој страни

Табела 33 - Наставак

	bool	"Boolean"
	double	"Double"
	float	"Float"
	string	"String"
	map_id	"Map"
	list_id	"List"
	arr_id	"array"
	of	"of"
	enum_id	"enumeration"
	const_id	"const"
	struct	"Struct"
Нетерминални		FIDL_MODEL
		PACK
		IMP
		IFACE
		TYPE_COLL
		METH
		ATT
		BROAD
		VERSION
		MAJOR
		MINOR
		ARG_M
		ATT_OP
		ARG_B
		ARG_DIR
		TYPE
		TYPE_DEF
		MAP
		LIST
		ARRAY
		ENUM
		ENUM_ITEM
		CONST
		CONST_FIELD

Правила продукције за *FIDL* језик су дата у наставку:

Исто као и у случају *AIDL* модела, најпре су дате продукције којима отпочиње стварање *FIDL* модела и којима су описани елементи на вишем нивоу који ће бити разложени у наставку.

FIDL_MODEL → PACK
FIDL_MODEL → PACK IMP
FIDL_MODEL → PACK IFACE
FIDL_MODEL → PACK TYPE_COLL
FIDL_MODEL → PACK IMP IFACE
FIDL_MODEL → PACK IMP TYPE_COLL

Елементи за дефинисање пакета и укључивања зависности од осталих модела су најједноставнији јер их је било могуће одмах описати терминалним симболима.

PACK → pack_id var_id
IMP → imp_id var_id

Правила, односно продукције за стварање спрега су кључни део и овог модела. Спрега се састоје од више нетерминалних симбола, датих за *SOME/IP* механизме методе, поља и догађаја, као и аргумената који се у њима појављују.

IFACE → iface_id var_id lbrace [VERSION] TYPE_DEF* METH* ATT* BROAD* rbrace
VERSION → ver_id lparen MAJOR MINOR BPAREN
MAJOR → major_id var_id
MINOR → minor_id var_id
METH → meth_id var_id [ff_id] lbrace ARG_M* rbrace
ATT → att_id TYPE var_id ATT_OP*
ATT_OP → [nr_id] [ro_id] [ns_id]
BROAD → broad_id var_id lparen [ARG_B] rparen
ARG_M → ARG_DIR lbrace TYPE var_id rbrace
ARG_DIR → in_id | out_id | inout_id
ARG_B → [out_id] lbrace TYPE var_id rbrace

Поред спрега, посебне продукције су написане и за део *FIDL* модела који представља скуп типова, односно саме типове, било да је реч о простим, сложеним, набројивои типовима и слично.

TYPE_COLL → tc_id lbrace TYPE_DEF* rbrace
TYPE → int
TYPE → int8
TYPE → int16
TYPE → int32
TYPE → int64
TYPE → uint8
TYPE → uint16
TYPE → uint32
TYPE → uint64
TYPE → char
TYPE → long
TYPE → bool
TYPE → double
TYPE → float
TYPE → string
TYPE → TYPE_DEF

TYPE_DEF → int var_id
TYPE_DEF → int8 var_id
TYPE_DEF → int16 var_id
TYPE_DEF → int32 var_id
TYPE_DEF → int64 var_id
TYPE_DEF → uint8 var_id
TYPE_DEF → uint16 var_id
TYPE_DEF → uint32 var_id
TYPE_DEF → uint64 var_id
TYPE_DEF → char var_id
TYPE_DEF → long var_id
TYPE_DEF → bool var_id
TYPE_DEF → double var_id
TYPE_DEF → float var_id
TYPE_DEF → string var_id
TYPE_DEF → MAP
TYPE_DEF → LIST
TYPE_DEF → ARRAY
TYPE_DEF → struct
TYPE_DEF → ENUM
STRUCT → struct_ID var_id lbrace TYPE_DEF* rbrace
MAP → map_id lcomp [TYPE] rcomp var_id
LIST → list_id var_id
ARRAY → arr_id var_id of TYPE
ENUM → enum_id var_id lbrace ENUM_ITEM* rbrace
ENUM_ITEM → var_id comma
TYPE → TYPE lbrack rbrack
CONST → const_id CONST_FIELD
CONST_FIELD → TYPE eq var_id

Граматика *FDEPL* језика

У табели 34 биће приказани терминални симболи *FDEPL* језика на основу техничке документације [104] и нетерминални симболи који представљају логичку структуру терминалних симбола потребну за описивање граматике. Након табеле у наставку су дате и продукције, односно правила која дефинишу нетерминалне симболе, односно граматику.

Табела 34 - Симболи *FDEPL* језика

Тип симбола	Назив симбола	Представа симбола
Терминални	lbrace	"{"
	rbrace	"}"
	eq	"="
	var_id	називи идентификатора (променљиве, спреге, методе, ...)
	imp_id	"import"
	define_id	"define"
	for	"for"
	iface_id	"interface"
	provider_id	"provider"
	as	"as"
	service	"Service"
	service_id	"SomeIpServiceID"
	attribute	"attribute"
	method	"method"
	broadcast	"broadcast"
	reliable_id	"SomeIpReliable"
	get_id	"SomeIpGetterID"
	set_id	"SomeIpSetterID"
	not_id	"SomeIpNotifierID"
	meth_id	"SomeIpMethodID"
	event_id	"SomeIpEventID"
	event_gps_id	"SomeIpEventGroups"
	instance	"instance"
	inst_id	"InstanceId"
	someip_inst_id	"SomeIpInstanceID"
Нетерминални		FDEPL_MODEL
		IMP
		DEFINE
		IFACE_DEF
		METH_DEF
		ATT_DEF
		BROAD_DEF
		GET_ID
		SET_ID
		NOT_ID
		METH_ID
		EVENT_ID
		EVENT_GROUP_ID
		RELIABLE_ID
		INSTANCE_DEF

Наставак на следећој страни

Табела 34 - Наставак

	INSTANCE_ID SOMEIP_INSTANCE_ID

Правила продукције за *FDEPL* језик су дата у наставку.

С обзиром на то да је *FDEPL* допуна *FIDL* модела коришћена у сврху конфигурације параметара неопходних за омогућавање комуникације преко *SOME/IP* протокола, начин дефинисања правила је повезан. Тако се најпре дефинишу продукције за модел, укључивања зависности са моделима и дефинисање параметара.

FDEPL_MODEL → IMP DEFINE
IMP → imp_id var_id
DEFINE → define_id var_id for iface_id var_id lbrace IFACE_DEF rbrace
DEFINE → define_id var_id for provider_id as service lbrace INSTANCE_DEF* rbrace

Елемент дефинисања (претходно наведен као *DEFINE*) је логичка целина чија се продукција даље разлаже на нетерминалне симболе за опис спреге, појединачних параметара сваког од коришћених *SOME/IP* механизама редом за поље, метод и догађај и на крају инстанце спрежног пролаза.

IFACE_DEF → service_id eq var_id EVENT_DEF* METH_DEF* ATT_DEF*
ATT_DEF → attribute var_id lbrace [GET_ID] [SET_ID] [NOT_ID] RELIABLE_ID rbrace
GET_ID → get_id eq var_id
SET_ID → set_id eq var_id
NOT_ID → not_id eq var_id
RELIABLE_ID → reliable_id eq var_id
METH_DEF → method var_id lbrace METH_ID RELIABLE_ID rbrace
METH_ID → meth_id eq var_id
BROAD_DEF → broadcast var_id lbrace EVENT_ID RELIABLE_ID EVENT_GROUP_ID rbrace
EVENT_ID → event_id eq var_id
EVENT_GROUP_ID → event_gps_id eq var_id
INSTANCE_DEF → instance var_id lbrace INSTANCE_ID someip_inst_id rbrace
INSTANCE_ID → inst_id eq var_id
SOMEIP_INSTANCE_ID → someip_inst_id eq var_id

Граматика *ARXML* језика

У табели 35 биће приказани терминални симболи *ARXML* језика на основу техничке документације [204] и нетерминални симболи који представљају логичку структуру терминалних симбола потребну за описивање граматике. Након табеле у наставку су дате и продукције, односно правила која дефинишу нетерминалне симболе, односно граматику.

Табела 35 - Симболи *ARXML* језика

Тип симбола	Назив симбола	Представа симбола
Терминални	lcomp	"<"
	lcomp_cl	"</"
	rcomp	">"
	eq	"="
	pack_id	"AR-PACKAGE"
	autosar_id	"AUTOSAR"
	xml	"xmlns"
	xsi	"xmlns:xsi"
	loc	xsi:schemaLocation
	el_id	"ELEMENTS"
	shn_id	"SHORT-NAME"
	aa_swc_id	"ADAPTIVE-APPLICATION-SW-COMPONENT-TYPE"
	ports_id	"PORTS"
	pport_id	"P-PORT-PROTOTYPE"
	rport_id	"R-PORT-PROTOTYPE"
	p_iface_id	"PROVIDED-INTERFACE-TREF"
	iface_id	"SERVICE-INTERFACE"
	meths_id	"METHODS"
	cs_op_id	"CLIENT-SERVICE-OPERATION"
	ff_id	"FIRE-AND-FORGET"
	true_id	"true"
	false_id	"false"
	args_id	"ARGUMENTS"
	arg_data_id	"ARGUMENT-DATA-PROTOTYPE"
	dir_id	"DIRECTION"
	in_id	"IN"
	out_id	"OUT"
	inout_id	"INOUT"
	events_id	"EVENTS"
	var_data_id	"VARIABLE-DATA-PROTOTYPE"
	fields_id	"FIELDS"
	field_id	"FIELD"
	get_id	"HAS-GETTER"
	not_id	"HAS-NOTIFIER"
	set_id	"HAS-SETTER"
	type_ref_id	"TYPE-TREF"
	dest	"dest"
	var_id	називи идентификатора (променљиве, спреге, методе, ...)
	int8	"int8"
	int16	"int16"
	int32	"int32"

Наставак на следећој страни

Табела 35 - Наставак

int64	"int64"
uint32	"uint8"
uint32	"uint16"
uint32	"uint32"
uint64	"uint64"
char	"char"
long	"long"
bool	"boolean"
double	"double"
float	"float"
string	"string"
map	"ASSOCIATIVE_MAP"
list	"List"
impl_data_id	"STD-CPP-IMPLEMENTATION-DATA-TYPE"
cat_id	"CATEGORY"
temp_arg	"TEMPLATE-ARGUMENTS"
cpp_temp_arg_id	"CPP-TEMPLATE-ARGUMENT"
sub_el_id	"SUB-ELEMENTS"
data_type_el_id	"CPP-IMPLEMENTATION-DATA-TYPE-ELEMENT"
type_em_id	"TYPE-EMITTER"
emit_ara	"TYPE_EMITTER_ARA"
data_props_id	"SW-DATA-DEF-PROPS"
variants_id	"SW-DATA-DEF-PROPS-VARIANTS"
cond_id	"SW-DATA-DEF-PROPS-CONDITIONAL"
compu_ref_id	"COMPU-METHOD-REF"
compu_m_id	"COMPU-METHOD"
compu_scales_id	"COMPU-SCALES"
compu_scale_id	"COMPU-SCALE"
low_lim_id	"LOWER-LIMIT"
up_lim_id	"UPPER-LIMIT"
c_const_id	"COMPU-CONST"
vt_id	"VT"
const_spec	"CONSTANT-SPECIFICATION"
value_spec_id	"VALUE-SPEC"
text_spec	"TEXT-VALUE-SPECIFICATION"
num_spec	"NUMERICAL-VALUE-SPECIFICATION"
value_id	"VALUE"
inst_to_port_id	"SERVICE-INSTANCE-TO-PORT-PROTOTYPE-MAPPING"
port_proto_iref	"PORT-PROTOTYPE-IREF"
root_sw_ref_id	"CONTEXT-ROOT-SW-COMPONENT-PROTOTYPE-REF"
trgt_port_ref_id	"TARGET-PORT-PROTOTYPE-REF"
inst_ref_id	"SERVICE-INSTANCE-REF"
inst_to_mach_id	"SOMEIP-SERVICE-INSTANCE-TO-MACHINE-MAPPING"
com_ref_id	"COMMUNICATION-CONNECTOR-REF"
udp_port_id	"UDP-PORT"
tcp_port_id	"TCP-PORT"
someip_inst_id	"PROVIDED-SOMEIP-SERVICE-INSTANCE"
iface_depl_ref_id	"SERVICE-INTERFACE-DEPLOYMENT-REF"

Наставак на следећој страни

Табела 35 - Наставак

	p_event_g_id	"PROVIDED-EVENT-GROUPS"
	someip_event_g_id	"SOMEIP-PROVIDED-EVENT-GROUP"
	event_g_ref_id	"EVENT-GROUP-REF"
	mcast_thold_id	"MULTICAST-THRESHOLD"
	sd_conf_id	"SD-SERVER-CONFIG-REF"
	inst_id	"SERVICE-INSTANCE-ID"
	iface_depl_id	"SOMEIP-SERVICE-INTERFACE-DEPLOYMENT"
	event_depls_id	"EVENT-DEPLOYMENTS"
	event_depl_id	"SOMEIP-EVENT-DEPLOYMENT"
	event_gps_id	"EVENT-GROUPS"
	event_ref_id	"EVENT-REF"
	event_id	"EVENT-ID"
	tp_id	"TRANSPORT-PROTOCOL"
	udp_id	"UDP"
	tcp_id	"TCP"
	meth_depls_id	"METHOD-DEPLOYMENTS"
	meth_depl_id	"SOMEIP-METHOD-DEPLOYMENT"
	meth_ref_id	"METHOD-REF"
	meth_id	"METHOD-ID"
	field_depls_id	"FIELD-DEPLOYMENTS"
	field_depl_id	"SOMEIP-FIELD-DEPLOYMENT"
	field_ref_id	"FIELD-REF"
	get_depl_id	"GET"
	set_depl_id	"SET"
	not_depl_id	"NOTIFIER"
	someip_iface_id	"SERVICE-INTERFACE-ID"
	someip_iface_v_id	"SERVICE-INTERFACE-VERSION"
	major_id	"MAJOR-VERSION"
	minor_id	"MINOR-VERSION"
Нетерминални	ARXML_MODEL	
	AUTOSAR	
	AUTOSAR_CL	
	PACK	
	PACK_CL	
	ELEM	
	ELEM_CL	
	NAME	
	SWC	
	PORTS	
	PPORT	
	RPORT	
	PROVIDE_REF	
	IFACE	
	METHS	
	CS_OP	
	FF	
	ARGS	
	ARG_DATA	

Наставак на следећој страни

Табела 35 - Наставак

	ARG_DIR
	FIELDS
	FIELD
	ATT_OP
	GET
	SET
	NOT
	EVENTS
	VAR_DATA
	TYPE_REF
	TYPE
	VAR_DATA
	DATA_TYPE
	CAT
	IDENTIFIER
	TEMPLATE
	CPP_TEMP_ARG
	SUB_ELEM
	DATA_TYPE_ELEM
	TYPE_EM
	SW_DATA_PROPS
	VARIANTS
	COND
	COMPU_REF
	COMPU_DEF
	COMPU_SCALES
	COMPU_SCALE
	LIMITS
	C_CONST
	VT
	CONST
	VALUE_SPEC
	TEXT_CONST
	NUM_CONST
	VALUE
	INSTANCE_TO_PORT
	PORT_PROTOTYPE_REF
	CONTEXT_ROOT_REF
	TARGET_PORT_REF
	INSTANCE_REF
	INSTANCE_TO_MACHINE
	COM_REF
	UDP
	TCP
	SOMEIP_INSTANCE
	IFACE_DEPL_REF
	EVENT_GROUP
	SOMEIP_EVENT_GROUP
	EVENT_GROUP_REF

Наставак на следећој страни

Табела 35 - Наставак

	MCAST_THOLD
	SD_CONF
	INSTANCE_ID
	SOMEIP_IFACE_DEPL
	EVENT_DEPLS
	EVENT_DEPL
	EVENT_GROUP_ID
	EVENT_REF
	EVENT_ID
	TP
	METH_DEPLS
	METH_DEPL
	METH_REF
	METH_ID
	FIELD_DEPLS
	FIELD_DEPL
	FIELD_REF
	GET_DEPL
	SET_DEPL
	NOT_DEPL
	SOMEIP_IFACE_ID
	SOMEIP_IFACE_VERSION
	MAJOR
	MINOR

Правила продукције за *ARXML* језик су дата у наставку:

Као и за све претходно дате граматике језика, у случају *ARXML* језика најпре су дефинисане продукције самог модела. Овај модел је најсложенији па је и број нетерминалних симбола, тј логичких целина на које се модел разлаже највећи што даље подразумева највише додатних продукција, тј. правила која је потребно дефинисати.

ARXML_MODEL → AUTOSAR PACK* PACK_CL* AUTOSAR_CL
ARXML_MODEL → AUTOSAR PACK* ELEM SWC* IFACE* DATA_TYPE* COMPU_DEF* CONST* ELEM_CL PACK_CL* INSTANCE_TO_PORT* INSTANCE_TO_MACHINE* SOMEIP_INSTANCE* SOMEIP_IFACE_DEPL* AUTOSAR_CL

Продукције, тј. правила за такорећи заглавље модела је дато на самом почетку и чине га терминални симболи који означавају верзију коришћеног станарда, шеме модела и слично. Након тога су дата правила за опис пакета у моделу који групишу остале елемененте.

AUTOSAR → lcomp_cl autosar_id xml eq var_id xsi eq var_id loc eq var_id rcomp
AUTOSAR_CL → lcomp_cl autosar_id rcomp
PACK → lcomp pack_id rcomp [NAME]
PACK_CL → lcomp_cl pack_id rcomp
NAME → lcomp shn_id rcomp var_id lcomp_cl shn_id rcomp
ELEM → lcomp el_id rcomp
ELEM_CL → lcomp_cl el_id rcomp

Основни елелемт је сама компонента програмске подршке која је дефинисана једноставном продукцијом која даље подразумева дефинисање нетерминалних симбола спрежних пролаза који се у компоненти налазе. За сваки спрежни пролаз, правилима се одређује смер тока података кроз њега као и спреге које овај пролаз имплементира.

SWC → lcomp aa_swc_id rcomp NAME PORTS lcomp_cl aa_swc_id rcomp
PORTS → lcomp ports_id rcomp PPORT* RPORT* lcomp_cl ports_id rcomp
PPORT → lcomp pport_id rcomp NAME PROVIDE_REF lcomp_cl pport_id rcomp
RPORT → lcomp rport_id rcomp NAME PROVIDE_REF lcomp_cl rport_id rcomp
PROVIDE_REF → lcomp p_iface_id rcomp var_id lcomp_ID p_iface_id rcomp

Нетерминални симболи који представљју спреге су дефинисани продукцијом која се наставља на продукције сваког појединачног *SOME/IP* механизма за методу, поље и догађај респективно.

IFACE → lcomp iface_id rcomp NAME METHS* FIELDS* EVENTS* lcomp_cl iface_id rcomp
METHS → lcomp meths_id rcomp CS_OP*
CS_OP → lcomp cs_op_id rcomp NAME ARGS FF lcomp_ID cs_op_id rcomp
ARGS → lcomp args_id rcomp ARG_DATA* lcomp_cl args_id rcomp
ARG_DATA → lcomp arg_data_id NAME TYPE_REF ARG_DIR rcomp lcomp_cl arg_data_id rcomp
ARG_DIR → lcomp dir_id rcomp in_id | out_id | inout_id lcomp_cl dir_id rcomp
FF → lcomp ff_id rcomp true_id | false_id lcomp_cl ff_id rcomp
FIELDS → lcomp fields_id rcomp FIELD* lcomp_cl fields_id rcomp
FIELD → lcomp field_id rcomp NAME TYPE_REF ATT_OP lcomp_cl field_id rcomp
ATT_OP → [GET] [SET] [NOT]
GET → lcomp get_id rcomp true_id | false_id lcomp_cl get_id rcomp
SET → lcomp set_id rcomp true_id | false_id lcomp_cl set_id rcomp
NOT → lcomp not_id rcomp true_id | false_id lcomp_cl not_id rcomp
EVENTS → lcomp events_id rcomp VAR_DATA* lcomp_cl events_id rcomp

Сваки од *SOME/IP* механизама у својој дефиницији користи податке чији су типови такође дефинисани засебним правилима, поготово у случају сложених и набројивих типова и константи.

VAR_DATA → lcomp var_data_id rcomp NAME TYPE_REF lcomp_cl var_data_id rcomp
TYPE_REF → lcomp type_ref_id dest eq TYPE rcomp lcomp_cl type_ref_id rcomp
TYPE → int
TYPE → int8
TYPE → int16
TYPE → int32
TYPE → int64
TYPE → uint8
TYPE → uint16
TYPE → uint32
TYPE → uint64
TYPE → char
TYPE → long
TYPE → bool
TYPE → double
TYPE → float
TYPE → DATA_TYPE
DATA_TYPE → lcomp impl_data_id rcomp NAME CAT IDENTIFIER TYPE_EM lcomp_cl impl_data_id rcomp
CAT → lcomp cat_id rcomp VEC | ARR | string | struct | map | REFERENCE | TEXT lcomp cat_id rcomp
IDENTIFIER → TEMPLATE | SUB_ELEM | TYPE_REF [SW_DATA_PROPS]
TEMPLATE → lcomp temp_arg* rcomp CPP_TEMP_ARG* lcomp_cl temp_arg rcomp
CPP_TEMP_ARG → lcomp cpp_temp_arg_id rcomp TYPE_REF lcomp_cl cpp_temp_arg_id rcomp
SUB_ELEM → lcomp sub_el_id rcomp DATA_TYPE_ELEM* lcomp sub_el_id rcomp
DATA_TYPE_ELEM → lcomp data_type_el_id rcomp NAME TYPE_REF lcomp_cl data_type_el_id rcomp
TYPE_EM → lcomp type_em_id rcomp emit_ara lcomp_cl type_em_id rcomp
SW_DATA_PROPS → lcomp data_props_id rcomp VARIANTS lcomp data_props_id rcomp
VARIANTS → lcomp variants_id rcomp COND lcomp_cl variants_id rcomp
COND → lcomp cond_id rcomp COMPU_REF lcomp_cl cond_id rcomp
COMPU_REF → lcomp compu_ref_id dest eq COMPU_DEF rcomp
COMPU_DEF → lcomp compu_m_id rcomp NAME CAT COMPU_SCALES lcomp_cl compu_m_id rcomp
COMPU_SCALES → lcomp compu_scales_id rcomp COMPU_SCALE* lcomp_cl compu_scales_id rcomp
COMPU_SCALE → lcomp compu_scale_id rcomp [NAME] [LIMITS] [C_CONST] lcomp_cl compu_scale_id rcomp
LIMITS → lcomp low_lim_id rcomp var_id lcomp_cl low_lim_id rcomp lcomp up_lim_id rcomp var_id lcomp_cl up_lim_id rcomp
C_CONST → lcomp c_const_id rcomp VT lcomp_cl c_const_id rcomp
VT → lcomp vt_id rcomp var_id lcomp_cl vt_id rcomp
CONST → lcomp const_spec rcomp NAME VALUE lcomp_cl const_spec rcomp
VALUE_SPEC → lcomp value_spec_id rcomp TEXT_CONST | NUM_CONST lcomp_cl value_spec_id rcomp
TEXT_CONST → lcomp text_spec rcomp VALUE lcomp_cl text_spec rcomp
NUM_CONST → lcomp num_spec rcomp VALUE lcomp_cl num_spec rcomp
VALUE → lcomp value_id rcomp var_id lcomp_cl value_id rcomp

За сваки спрежни пролаз који имплементира инстанцу одговарајуће спреге која је дата као један од основних елемената у моделу су дефинсана правила, тј. продукције. Ова правила описују настанак параметара неопходних за остваривање комуникације тако што повезују инстанцу спреге са одговарајућим спрежним пролазом из компоненте програмске подршке, дефинишу протокол, мрежни пролаз и адресу за пренос, као и параметре специфичне за сваки појединачни *SOME/IP* механизам као што је то случај у *FDEPL* моделу.

INSTANCE_TO_PORT → lcomp inst_to_port_id rcomp NAME PORT_PROTOTYPE_REF INSTANCE_REF lcomp_cl inst_to_port_id rcomp

PORT_PROTOTYPE_REF → lcomp port_proto_iref rcomp CONTEXT_ROOT_REF TARGET_PORT_REF lcomp_cl port_proto_iref rcomp

CONTEXT_ROOT_REF → lcomp root_sw_ref_id dest eq var_id rcomp var_id lcomp_cl root_sw_ref_id rcomp

TARGET_PORT_REF → lcomp trgt_port_ref_id dest eq var_id rcomp var_id lcomp_cl trgt_port_ref_id rcomp

INSTANCE_REF → lcomp inst_ref_id dest eq var_id rcomp lcomp_cl inst_ref_id rcomp

INSTANCE_TO_MACHINE → lcomp inst_to_mach_id rcomp NAME COM_REF INSTANCE_REF UDP | TCP lcomp_cl inst_to_mach_id rcomp

COM_REF → lcomp com_ref_id dest eq var_id rcomp var_id lcomp_cl com_ref_id rcomp

UDP → lcomp udp_port_id rcomp var_id lcomp_cl udp_port_id rcomp

TCP → lcomp tcp_port_id rcomp var_id lcomp_cl tcp_port_id rcomp

SOMEIP_INSTANCE → LCOMPSOMEIP_INSTANCE_ID rcomp NAME IFACE_DEPL_REF [EVENT_GROUP] SD_CONF INSTANCE_ID lcomp_cl someip_inst_id rcomp

IFACE_DEPL_REF → lcomp iface_depl_ref_id rcomp lcomp_cl iface_depl_ref_id rcomp

EVENT_GROUP → lcomp p_event_g_id rcomp SOMEIP_EVENT_GROUP lcomp_cl p_event_g_id rcomp

SOMEIP_EVENT_GROUP → lcomp someip_event_g_id rcomp NAME EVENT_GROUP_REF MCAST_THOLD lcomp_cl someip_event_g_id rcomp

EVENT_GROUP_REF → lcomp event_g_ref_id dest eq var_id rcomp var_id lcomp_cl event_g_ref_id rcomp

MCAST_THOLD → lcomp mcast_thold_id rcomp var_id lcomp_cl mcast_thold_id rcomp

SD_CONF → lcomp sd_conf_id dest eq var_id rcomp var_id lcomp_cl sd_conf_id rcomp

INSTANCE_ID → lcomp inst_id rcomp var_id lcomp_cl inst_id rcomp

SOMEIP_IFACE_DEPL	→ lcomp iface_depl_id rcomp NAME [EVENT_DEPL] [METH_DEPL] [FIELD_DEPL] SOMEIP_IFACE_ID SOMEIP_IFACE_VERSION lcomp_cl iface_depl_id rcomp
EVENT_DEPLS	→ lcomp event_depls_id rcomp EVENT_DEPL* lcomp_cl event_depls_id rcomp
EVENT_DEPL	→ lcomp event_depl_id rcomp NAME [EVENT_GROUP_ID] EVENT_REF EVENT_ID TP lcomp_cl event_depl_id rcomp
EVENT_GROUP_ID	→ lcomp event_gps_id rcomp var_id lcomp_cl event_gps_id rcomp
EVENT_REF	→ lcomp event_ref_id dest eq var_id rcomp var_id lcomp_cl event_ref_id rcomp
EVENT_ID	→ lcomp event_id rcomp var_id lcomp_cl event_id rcomp
TP	→ lcomp tp_id rcomp udp_id | tcp_id lcomp_cl tp_id rcomp
METH_DEPLS	→ lcomp meths_depl_id rcomp METH_DEPL* lcomp_cl meths_depl_id rcomp
METH_DEPL	→ lcomp meth_depl_id rcomp NAME METH_REF METH_ID TP lcomp_cl meth_depl_id rcomp
METH_REF	→ lcomp meth_ref_id dest eq var_id rcomp var_id lcomp_cl meth_ref_id rcomp
METH_ID	→ lcomp meth_id rcomp var_id lcomp_cl meth_id rcomp
FIELD_DEPLS	→ lcomp fields_depl_id rcomp FIELD_DEPL* lcomp_cl fields_depl_id rcomp
FIELD_DEPL	→ lcomp field_depl_id rcomp NAME FIELD_REF [GET_DEPL] [SET_DEPL] [NOT_DEPL] lcomp_cl field_depl_id rcomp
FIELD_REF	→ lcomp meth_ref_id dest eq var_id rcomp var_id lcomp_cl meth_ref_id rcomp
GET_DEPL	→ lcomp get_depl_id rcomp NAME METH_ID TP lcomp_cl get_depl_id rcomp
SET_DEPL	→ lcomp set_depl_id rcomp NAME METH_ID TP lcomp_cl set_depl_id rcomp
NOT_DEPL	→ lcomp not_depl_id rcomp NAME EVENT_ID TP lcomp_cl not_depl_id rcomp
SOMEIP_IFACE_ID	→ lcomp someip_iface_id rcomp var_id lcomp_cl someip_iface_id rcomp
SOMEIP_IFACE_VERSION	→ lcomp someip_iface_v_id rcomp MAJOR MINOR lcomp_cl someip_iface_v_id rcomp
MAJOR	→ lcomp major_id rcomp var_id lcomp_cl major_id rcomp
MINOR	→ lcomp minor_id rcomp var_id lcomp_cl minor_id rcomp

Након што је дат приказ граматике сваког од језика за дефинисање спреге, извршена је и верификација дефинисањем продукција за само превођење. Продукције су дефинисане тако што је најпре описано мапирање модела који говори о томе од чега тачно настаје циљани модел језика у који се преводи. Затим је за сваки нетерминални симбол из граматике модела који настаје дефинисана продукција употребом симбола из изворног модела. Иако се у граматици продукцијама не дефинишу терминални симболи, у неким случајевима је то овде урађено зато што ове продукције представљају правила мапирања, која постоје и међу терминалним симболима (углавном када је реч о типовима). С обзиром на то да нетерминални симболи могу бити из различитих изворних продукција, како би превођење било што јасније дефинисано, потребно је идентификовати нетерминални

симбол односно његово порекло. За то је коришћена ознака ":"којом је назначено тачно порекло нетерминалног симбола из изворног модела. Због разлика у именима, уведене су и манипулације (додавање и одузимање) специфичних низова карактера који су означени наводницима.

Најпре је приказано превођење из *ARXML* модела у *AIDL*.

Превођење *ARXML* модела у *AIDL*

Прво ће бити представљено мапирање за *AIDL* моделе који садрже сложене типове података. Овим продукцијама је приказано и мапирање примитивних типова.

AIDL_MODEL<TYPE> → DATA_TYPE:ARXML_MODEL

PACK	→ pack_id var_id:NAME:PACK:DATA_TYPE:ARXML
IMP	→ imp_id PACK:AIDL_MODEL<TYPE> + var_id:PARC semicol
PARC	→ parc_id var_id:NAME:DATA_TYPE lbrace PARC_FIELD* rbrace
PARC_FIELD	→ TYPE_REF:DATA_TYPE_ELEM:SUB_ELEM:DATA_TYPE var_id:NAME:DATA_TYPE_ELEM:SUB_ELEM:DATA_TYPE semicol
PARC_FIELD	→ TYPE_REF:CPP_TEMP_ARG:TEMPLATE:DATA_TYPE var_id:NAME:DATA_TYPE semicol
PARC_FIELD	→ TYPE_REF:DATA_TYPE var_id:NAME:DATA_TYPE semicol
string	→ string:CAT:DATA_TYPE
MAP	→ map_id lcomp TYPE_REF:CPP_TEMP_ARG:TEMPLATE:DATA_TYPE comma TYPE_REF:CPP_TEMP_ARG:TEMPLATE:DATA_TYPE rcomp var_id
LIST	→ list_id lcomp TYPE_REF:CPP_TEMP_ARG:TEMPLATE:DATA_TYPE rcomp var_id
TYPE	→ TYPE_REF:CPP_TEMP_ARG:TEMPLATE:VEC:CAT:DATA_TYPE lbrack rbrack
TYPE	→ TYPE_REF:CPP_TEMP_ARG:TEMPLATE:ARR:CAT:DATA_TYPE lbrack rbrack
int	→ int:ARXML_MODEL
int	→ int8:ARXML_MODEL
int	→ int16:ARXML_MODEL
int	→ int32:ARXML_MODEL
long	→ int64:ARXML_MODEL
int	→ uint:ARXML_MODEL
int	→ uint8:ARXML_MODEL
int	→ uint16:ARXML_MODEL
long	→ uint32:ARXML_MODEL
char	→ char:ARXML_MODEL
bool	→ bool:ARXML_MODEL
double	→ double:ARXML_MODEL
float	→ float:ARXML_MODEL

Након мапирања простих и сложених типова, приказана су и правила за настанак набројивих типова и константи.

AIDL_MODEL<ENUM> → COMPU_REF:ARXML_MODEL

PACK → pack_id var_id:NAME:PACK:COMPU_REF:ARXML
ENUM → BACK_ANN enum_id var_id:NAME:COMPU_DEF:COMPU_REF lbrace ENUM_FIELD* rbrace
ENUM_FIELD → var_id comma
BACK_ANN → ann_sym back lparen TYPE_ID eq quote CAT quote

AIDL_MODEL<CONST> → CONST:ARXML_MODEL

PACK → pack_id var_id:NAME:PACK:COMPU_REF:ARXML
ENUM → BACK_ANN enum_id var_id:NAME:COMPU_DEF:COMPU_REF lbrace ENUM_FIELD* rbrace
ENUM_FIELD → var_id comma
BACK_ANN → ann_sym back lparen TYPE_ID eq quote CAT quote

Након што је представљено мапирање свих типова података, дате су продукције за стварање главног *AIDL* модела, односно модела у ком се налази главна спрега којом кориснички програми добављају све остале спреге помоћу којих се врши *SOME/IP* комуникација.

AIDL_MODEL<SWC> → SWC:ARXML_MODEL

PACK → pack_id var_id:NAME:PACK:SWC:ARXML_MODEL semicol
IMP → imp_id var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:IFACE:ARXML_MODEL + var_id:NAME:PPORT:SWC:ARXML_MODEL | var_id:NAME:RPORT:SWC:ARXML_MODEL + semicol
IFACE → iface_id "ISWC_"+ var_id:NAME:SWC lbrace METHOD* rbrace
METHOD → var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:PPORT:"SWC:ARXML_MODEL | var_id:NAME:RPORT:SWC:ARXML_MODEL + var_id:NAME:INSTANCE_ID:PPORT:SWC:ARXML_MODEL | var_id:NAME:INSTANCE_ID:RPORT:SWC:ARXML_MODEL "get"+ var_id:PROVIDE_REF:PPORT:SWC | var_id:PROVIDE_REF:RPORT:SWC lparen BPAREN semicol

Након главног модела, за сваки спрежни пролаз који служи да пружа податке и који је имплементиран у одговарајућој компоненти на *ADAS* страни се ствара појединачни *AIDL* модел са свим својим методама које настају из *SOME/IP* механизама.

AIDL_MODEL<PPORT> → PPORT:SWC:ARXML_MODEL

PACK → pack_id var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:IFACE:ARXML_MODEL + var_id:NAME:PPORT:SWC:ARXML_MODEL
IMP → imp_id PACK:AIDL_MODEL<TYPE> + var_id:PARC semicol

IMP → imp_id PACK:AIDL_MODEL<ENUM> + var_id:PARC semicol
IMP → imp_id PACK:AIDL_MODEL<CONST> + var_id:PARC semicol
IMP → imp_id var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:PPORT:SWC:ARXML_MODEL + var_id:NAME:IFACE:ARXML_MODEL +
→ var_id:IFACE:ARXML_MODEL<EVENT> | var_id:IFACE:ARXML_MODEL<NOT>
IFACE → iface_id "I"+ var_id:PROVIDE_REF:PPORT lbrace METHOD* rbrace
METHOD → [FF:CS_OP:METHS] VOID var_id:NAME:CS_OP:METHS lparen ARG* BPAREN semicol
ARG → [in_id:ARG_DIR:ARG_DATA:ARGS:CS_OP:METHS] | [out_id:ARG_DIR:ARG_DATA:ARGS:CS_OP:METHS]
→ | [inout_id:ARG_DIR:ARG_DATA:ARGS:CS_OP:METHS] TYPE_REF:ARG_DATA:ARGS:CS_OP:METHS var_id:NAME:ARG_DATA:ARGS:CS_OP:METHS
METHOD → VOID var_id:NAME:GET:ATT_OP:FIELD:FIELDS + "_GETTER"lparen ARG* BPAREN semicol
ARG → out_id TYPE_REF:FIELD:FIELDS var_id:NAME:FIELD:FIELDS
METHOD → byte "register"+ var_id:NAME:NOT:ATT_OP:FIELD:FIELDS + "Listener"+ "_NOTIFIER"lparen ARG* BPAREN semicol
ARG → in_id var_id:IFACE:AIDL_MODEL<NOT> + "listener"
METHOD → byte "register"+ var_id:NAME:VAR_DATA:EVENTS + "Listener"+ "_EVENT"lparen ARG* BPAREN semicol
ARG → in_id var_id:IFACE:AIDL_MODEL<EVENT> + "listener"

Како би обавештавајући механизми били омогућени, неопходно је за почетак направити *AIDL* модел за свако поље за обавештавање из спрега на *ADAS* страни. Ово наравно подразумева и методе којима се корисник претплаћује на пристигле информације.

AIDL_MODEL<NOT> → NOT:ATT_OP:FIELD:FIELDS:IFACE:PROVIDE_REF: PPORT:SWC:ARXML_MODEL

PACK → pack_id var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:IFACE:ARXML_MODEL + var_id:NAME:PPORT:SWC:ARXML_MODEL
IMP → imp_id PACK:AIDL_MODEL<TYPE> + var_id:PARC semicol
IMP → imp_id PACK:AIDL_MODEL<ENUM> + var_id:PARC semicol
IMP → imp_id PACK:AIDL_MODEL<CONST> + var_id:PARC semicol
IFACE → iface_id "I"+ var_id:NAME:NOT:ATT_OP:FIELD:FIELDS + "Listener"lbrace METHOD* rbrace
METHOD → [ow_id] VOID var_id:NAME:NOT:ATT_OP:FIELD:FIELD lparen ARG* BPAREN semicol
ARG → in_id TYPE_REF:FIELD:FIELDS var_id NAME:FIELD:FIELDS

Поред модела који настаје од поља за обавештавање и у случају догађаја је неопходно створити *AIDL* модел и методе путем којих че крајњи корисник бити обавештен о промени података на које је претплаћен.

AIDL_MODEL<EVENT> → EVENTS:IFACE:PROVIDE_REF:PPORT: SWC:ARXML_MODEL

PACK	→ pack_id var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:PPORT:SWC:ARXML_MODEL
IMP	→ imp_id PACK:AIDL_MODEL<TYPE> + var_id:PARC semicol
IMP	→ imp_id PACK:AIDL_MODEL<ENUM> + var_id:PARC semicol
IMP	→ imp_id PACK:AIDL_MODEL<CONST> + var_id:PARC semicol
IFACE	→ iface_id "I"+ var_id:NAME:VAR_DATA:EVENTS + "Listener"lbrace METHOD* rbrace
METHOD	→ [ow_id] VOID var_id:NAME:VAR_DATA:EVENTS lparen ARG* BPAREN semicol
ARG	→ in_id TYPE_REF:VAR_DATA:EVENTS var_id NAME:VAR_DATA:EVENTS

Као што је случај са спрежним пролазима за пружање података, *AIDL* модел се ствара и из сваког спрежног пролаза за добављање података са одговарајуће компоненте на *ADAS* страни.

AIDL_MODEL<RPORT> → RPORT:SWC:ARXML_MODEL

PACK	→ pack_id var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:RPORT:SWC:ARXML_MODEL
IMP	→ imp_id PACK:AIDL_MODEL<TYPE> + var_id:PARC semicol
IMP	→ imp_id PACK:AIDL_MODEL<ENUM> + var_id:PARC semicol
IMP	→ imp_id PACK:AIDL_MODEL<CONST> + var_id:PARC semicol
IMP	→ imp_id var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:RPORT:SWC:ARXML_MODEL + var_id:IFACE:ARXML_MODEL<SET>
IFACE	→ iface_id "I"+ var_id:PROVIDE_REF:RPORT lbrace METHOD* rbrace
METHOD	→ byte "register"+ var_id:NAME:SET:ATT_OP:FIELD:FIELDS + "Listener"+ "_SETTER"lparen ARG* BPAREN semicol
ARG	→ in_id var_id:IFACE:AIDL_MODEL<SET> + "listener"

Постоји још један случај стварања *AIDL* модела, а то је од поља за постављање на *ADAS* страни. У овом случају превођење је исто као и за моделе који настају од догађаја и поља за обавештавање.

AIDL_MODEL<SET> → SET:ATT_OP:FIELD:FIELDS:IFACE:PROVIDE_REF: RPORT:SWC:ARXML_MODEL

PACK → pack_id var_id:NAME:PACK:SWC:ARXML_MODEL + var_id:NAME:SWC:ARXML_MODEL + var_id:NAME:PACK:IFACE:ARXML_MODEL + var_id:NAME:RPORT:SWC:ARXML_MODEL + var_id:NAME:INSTANCE_ID:RPORT:SWC:ARXML_MODEL

IMP → imp_id PACK:AIDL_MODEL<TYPE> + var_id:PARC semicol

IMP → imp_id PACK:AIDL_MODEL<ENUM> + var_id:PARC semicol

IMP → imp_id PACK:AIDL_MODEL<CONST> + var_id:PARC semicol

IFACE → iface_id "I"+ var_id:NAME:SETT:ATT_OP:FIELD:FIELDS + "Listener"lbrace METHOD* rbrace

METHOD → [ow_id] VOID var_id:NAME:SETT:ATT_OP:FIELD:FIELDS lparen ARG* BPAREN semicol

ARG → in_id TYPE_REF:FIELD:FIELDS var_id NAME:FIELD:FIELDS

Превођење *AIDL* модела у *FIDL*

У случају превођења *AIDL* модела у *FIDL*, нема главног *FIDL* модела који настаје од саме програмске компоненте, већ *FIDL* модел настаје од појединачне спреге. Да би ово било обезбеђено не гледа се сваки *AIDL* модел спрежног пролаза (јер више спрежних пролаза могу бити имплемтирани од исте спреге), већ се то ради на основу укључених зависности у главном *AIDL* моделу где је могуће закључити који спрежни пролази су настали од исте спреге на основу пакета који се укључује. Најпре се модел креира од спрежних пролаза који шаљу информације.

FIDL_MODEL<IFACE> → PPORT:IMP:AIDL_MODEL<SWC>

PACK → pack_id var_id:PACK:AIDL_MODEL - "INSTANCE_ID"

IMP → imp_id PACK:AIDL_MODEL<TYPE_COLL>

IFACE → iface_id var_id:IFACE:AIDL_MODEL<PPORT> lbrace [VERSION] METH* ATT* BROAD* rbrace

METH → meth_id var_id:METH:IFACE:AIDL_MODEL<PPORT> [ff_id] lbrace ARG_M* rbrace

ff_id → ow_id:METH:IFACE:AIDL_MODEL<IFACE>

ARG_M → ARG_DIR lbrace TYPE:ARG:METH:IFACE:AIDL_MODEL<PPORT> var_id:ARG:METH:IFACE:AIDL_MODEL<PPORT> rbrace

ARG_DIR → in_id:ARG:METH:IFACE:AIDL_MODEL<PPORT> | out_id:ARG:METH:IFACE:AIDL_MODEL<PPORT> | inout_id:ARG:METH:IFACE:AIDL_MODEL<PPORT>

ATT → att_id TYPE:ARG:METH:IFACE:AIDL_MODEL<GET> var_id:ARG:METH:IFACE:AIDL_MODEL<GET> ATT_OP*

ATT → att_id TYPE:ARG:METH:IFACE:AIDL_MODEL<NOT> var_id:ARG:METH:IFACE:AIDL_MODEL<NOT> ATT_OP*

ATT_OP → [nr_id] [ro_id] [ns_id]

ro_id ns_id → METH:IFACE:AIDL_MODEL<GET>

ro_id nr_id → METH:IFACE:AIDL_MODEL<NOT>

ro_id → METH:IFACE:AIDL_MODEL<GET> & METH:IFACE:AIDL_MODEL<NOT>

BROAD → broad_id var_id:ARG:METH:IFACE:AIDL_MODEL<EVENT> lparen [ARG_B] rparen

ARG_B → [out_id] lbrace TYPE:ARG:METH:IFACE:AIDL_MODEL<EVENT> rbrace

Након стварања *FIDL* модела на основу спрега чији спрежни пролази шаљу податке, на исти начин се стварају и *FIDL* модели за спреге који спрежни пролази добављају податке.

FIDL_MODEL<IFACE> → RPORT:IMP:AIDL_MODEL<SWC>

PACK → pack_id var_id:PACK:AIDL_MODEL - "INSTANCE_ID"
IMP → imp_id PACK:AIDL_MODEL<TYPE_COLL>
IFACE → iface_id var_id:IFACE:AIDL_MODEL<RPORT> lbrace [VERSION] ATT* rbrace
ATT → att_id TYPE:ARG:METH:IFACE:AIDL_MODEL<SET> var_id:ARG:METH:IFACE:AIDL_MODEL<SET> ATT_OP*
ATT_OP → nr_id ns_id

Поред *FIDL* модела који настају од самих спрега, ствара се и модел који дефинише типове. Ово подразумева превожење сложених и набројивих типова и константи.

FIDL_MODEL<TYPE_COLL> → AIDL_MODEL<TYPE>

PACK → pack_id var_id:PACK:AIDL_MODEL<TYPE>
IMP → imp_id PACK:FIDL_MODEL<TYPE_COLL>
TYPE_COLL → tc_id lbrace TYPE_DEF* rbrace
TYPE_DEF → TYPE:PARC_FIELD:PARC:AIDL_MODEL<TYPE> var_id:PARC_FIELD:PARC:AIDL_MODEL<TYPE>

FIDL_MODEL<TYPE_COLL> → AIDL_MODEL<ENUM>

PACK → pack_id var_id:PACK:AIDL_MODEL<ENUM>
IMP → imp_id PACK:FIDL_MODEL<TYPE_COLL>
TYPE_COLL → tc_id lbrace TYPE_DEF* rbrace
TYPE_DEF → ENUM
ENUM → enum_id var_id:ENUM:AIDL_MODEL<ENUM> lbrace ENUM_ITEM* rbrace
ENUM_ITEM → var_id:ENUM_FIELD:ENUM:AIDL_MODEL<ENUM> comma

FIDL_MODEL<TYPE_COLL> → AIDL_MODEL<CONST>

PACK → pack_id var_id:PACK:AIDL_MODEL<CONST>
IMP → imp_id PACK:FIDL_MODEL<TYPE_COLL>
TYPE_COLL → tc_id lbrace TYPE_DEF* rbrace
TYPE_DEF → CONST
CONST → const_id CONST_FIELD
CONST_FIELD → int:CONST_FIELD:CONS:IFACE:AIDL_MODEL<CONST> var_id:CONST_FIELD:CONST:IFACE:AIDL_MODEL<CONST> eq var_id:CONST_FIELD:CONST:IFACE:AIDL_MODEL<CONST>
CONST_FIELD → string:CONST_FIELD:CONS:IFACE:AIDL_MODEL<CONST> var_id:CONST_FIELD:CONST:IFACE:AIDL_MODEL<CONST> eq var_id:CONST_FIELD:CONST:IFACE:AIDL_MODEL<CONST>

Превођење *ARXML* модела у *FDEPL*

Формална верификације је урађена и за превођење параметара извршавања *SOME/IP* комуникације мапирањем између *ARXML* и *FDEPL* модела. У овом превођењу, најпре се укључују се зависности са *FIDL* моделом за чије се спреге дефинишу параметри, а потом се из *ARXML* модела преводе идентификатори инстанце дефинисаних спрега и параметри појединачних *SOME/IP* механизама коришћених у дефинисаној спрези.

FDEPL_MODEL<IFACE> → **IFACE_DEPL_REF:SOMEIP_INSTANCE: ARXML_MODEL<SWC>**

IMP	→ imp_id IFACE:PROVIDE_REF:PPORT:SWC: ARXML_MODEL<SWC> + ".fidl"
DEFINE	→ define_id var_id for iface_id var_id lbrace IFACE_DEF rbrace
DEFINE	→ define_id var_id for provider_id as service lbrace INSTANCE_DEF* rbrace
IFACE_DEF	→ service_id eq var_id:SOMEIP_IFACE_ID EVENT_DEF* METH_DEF* ATT_DEF*
ATT_DEF	→ attribute var_id lbrace [GET_ID] [SET_ID] [NOT_ID] RELIABLE_ID rbrace
GET_ID	→ get_id eq var_id:METHOD_ID:GET_DEPL:FIELD_DEPL: FIELD_DEPLS:SOMEIP_IFACE_DEPL:ARXML_MODEL
SET_ID	→ set_id eq var_id:METHOD_ID:SET_DEPL:FIELD_DEPL: FIELD_DEPLS:SOMEIP_IFACE_DEPL:ARXML_MODEL
NOT_ID	→ not_id eq var_id:EVENT_ID:NOT_DEPL:FIELD_DEPL: FIELD_DEPLS:SOMEIP_IFACE_DEPL:ARXML_MODEL
RELIABLE_ID	→ reliable_id eq udp_id:TP:SOMEIP_IFACE_DEPL:ARXML_MODEL | tcp_id:TP:SOMEIP_IFACE_DEPL:ARXML_MODEL
METH_DEF	→ method var_id lbrace METH_ID RELIABLE_ID rbrace
METH_ID	→ meth_id eq var_id:METHOD_ID:METH_DEPL:METHS_DEPLS: SOMEIP_IFACE_DEPL:ARXML_MODEL
BROAD_DEF	→ broadcast var_id lbrace EVENT_ID RELIABLE_ID EVENT_GROUP_ID rbrace
EVENT_ID	→ event_id eq var_id:EVENT_ID:EVENT_DEPL:EVENT_DEPLS: SOMEIP_IFACE_DEPL:ARXML_MODEL
EVENT_GROUP_ID	→ event_gps_id eq var_id:EVENT_GROUP_ID:EVENT_DEPL: EVENT_DEPLS:SOMEIP_IFACE_DEPL:ARXML_MODELL
INSTANCE_DEF	→ instance var_id:NAME:SOMEIP_INSTANCE:ARXML_MODEL lbrace INSTANCE_ID someip_inst_id rbrace
INSTANCE_ID	→ inst_id eq var_id:INSTANCE_ID: SOMEIP_INSTANCE:ARXML_MODEL
SOMEIP_INSTANCE_ID	→ someip_inst_id eq var_id:INSTANCE_ID: SOMEIP_INSTANCE:ARXML_MODEL

Остатак генерисања програмске подршке, односно кода у неком од програмских језика је такође, како је већ приказано, грађен почевши од дефинисаних захтева, затим модела генерисања приказаним у поглављу 5. Са друге стране, продукције за ове фазе генерисања нису приказане из разлога што је стварање кода у одређеном програмском језику вршено путем модел у текст превођења без првобитне апстракције самог програмског језика којим је имплементирана програмска подршка потребна за размену података.

6.2.3.5 Тестирање генерисања *CommonAPI SOME/IP* клијената Тестирање јединица имплементације генерисања *CommonAPI SOME/IP* клијената је извршено провером функција за генерисање:

- клијентских или серверских датотека у зависности од тога да ли је коришћено поље за постављање вредности или не
- мапе и помоћних променљивих које служе за правилно руковање вишеструким инстанцама исте спреге
- иницијализационог и деиницијализационог дела
- методе и њених аргумената (укључује и проверу типова)
- поља за добављање вредности (укључује и проверу онемогућавања аргумената јер је то карактеристично само за методе, не и за поље)
- претплате на догађаје и поља за обавештавање и постављање вредности (укључује и проверу типова као и исправност позива претплаћене повратне методе)
- метода које служе за прилагођавање типских неслагања

Табела 36: Тестирање генерисања *CommonAPI SOME/IP* клијената

Ниво тестирања	**Тестирана категорија**	**Број тестова**
Тестирање јединица	стварање клијентских или серверских датотека	8
	стварање промењљивих за руковање вишеструким инстанцама спрега	6
	иницијализација	7
	деиницијализација	7
	руковање SOME/IP методама и аргументима	6
	руковање SOME/IP пољима за добављање вредности	4
	руковање SOME/IP догађајима и пољима за обавештавање и постављање вредности	8
	прилагођавање типских неслагања	9
Интеграциони тестови	провера размене података путем генерисаних клијената	19
Функционални тестови	потврђивање рада реалних сценарија користећи генерисане клијенте	19
Укупно		93

Интеграционо и функционално тестирање није вршено за сам генератор, већ су на исти начин као и за имплементацију комуникације проверени генерисани *CommonAPI SOME/IP* клијенти у сценаријима где је од њих очекивана исправна размена података. Тестови су приказани табелом 36.

6.2.3.6 Тестирање генерисања *Android* сервиса Као и сва тестирања до сад и генерисање *Android* сервиса је тестирано на три нивоа функционалним, интеграционим и тестовима јединица имплементације, као што се може видети у табели 37.

Тестови јединица имплемнтације за генерисање сервиса за сва три програмска језика (*C++*, *Java* и *Kotlin*) подразумевају тестирање функција за генерисање:

- главног дела који врши иницијализацију и додавање сервиса у систем
- главне спреге и метода које корисницима пружају инстанце *binder* обејаката над којима се врши комуникација
- наслеђивање генерисаног модула сервисног дела *AIDL* спрега за сваку појединачну инстанцу *SOME/IP* спреге
- метода које позивају корисничке апликације (узимајући у обзир аргументе и њихове типове) и које надаље позивају методе *CommonAPI SOME/IP* клијената
- метода које позивају корисничке апликације, а које надаље позивају поља за добављање вредности путем *CommonAPI SOME/IP* клијената
- метода које позивају корисничке апликације за потребе претплате на догађај или поља за обавештавање и постављање вредности
- поља и компоненте које служе да рукују додавањем и избацивањем претплаћених корисника из групе корисника које је потребно обавестити о пристиглим информацијама
- повратних метода које позивају *CommonAPI SOME/IP* клијенти када тражена информација пристигне
- механизми који на основу броја клијената и активности квалитета услуге раздвајају начин дистрибуирања пристиглих информација (*Intent*, *oneway* или руковалац догађајима)
- компоненте која представља базен нити у случају даље дистрибуирања података путем руковаоца догађајима
- компоненте за руковање догађајима
- метода за ослобађање ресурса

Генерисање *.bp* и *.te* датотека за превођење и омогућавање *SELinux* модула нису тестирани јер могу да варирају од система до система (односно захтевају ручну проверу исправности и прилагођавање по потреби).

Са друге стране, интеграциони и функционални тестови се, као и у случају генерисања *CommonAPI SOME/IP* клијената, своде на проверу исправности преузимања података са интегрисаних *SOME/IP* клијената и дистрибуирања до корисничких апликација на примерима коришћеним при тестирању имплементиране комуникације.

Табела 37: Тестирање генерисања *Android* сервиса (за све подржане програмске језике)

Ниво тестирања	**Тестирана категорија**	**Број тестова**
Тестирање јединица	иницијализација и додавање сервиса у систем	12
	стварање главне спреге	9
	имплементација сервисног дела AIDL спрега	6
	методе којима корисничке апликације потражују извршавање SOME/IP метода	15
	методе којима корисничке апликације потражују извршавање SOME/IP поља за добављање вредности	9
	методе којима корисничке апликације врше претплату SOME/IP обавештавајуће механизме	18
	руковање додавањем и избацивањем претплатних корисничких апликација	9
	повратне методе прозване на долазак информације	11
	механизми за дистрибуицију пристиглих информација	7
	компонента за руковање догађајима	4
	компонента која садржи базен нити	4
	ослобађање ресурса	5
Интеграциони тестови	провера размене података путем генерисаних сервиса	19
Функционални тестови	потврђивање рада реалних сценарија користећи генерисане сервисе	19
Укупно		145

7 Закључак

У овој дисертацији је предложено једно решење за побољшање целокупног комуникационог подсистема у возилима. Решење се темељи на тренутној потреби за поспешивањем комуникационог подсистема како би се избегле вишеструке имплементације истих или сличних компонената, како програмске подршке тако и физичке архитектуре и поред тога смањио циклус развоја и интеграције у производњи. Ове потребе су производ тренутних захтева и архитектуралних праваца развоја система модерног возила приказаних у уводном делу дисертације. Најпре су дате теоријске основе неопходне за разумевање тренутног правца развоја, сложености имплементација и технологија које се користе у аутомобилским системима и потреба за решењем које ефикасно омогућава размену података између два домена круцијална за програмску подршку аутомобилских система. Реч је о доменима за напредну подршку возачу и инфо-забавним доменима. Иако решење најпре решава питање међудоменске комуникације у тренутној архитектури возила, подржава и будуће правце развоја аутомобилског система које обухвата зоналну и централизвану архитектуру.

Дисертацијом је изложена идеја и реализација која нуди аутоматско генерисање програмске подршке потребне за остваривање сервисно-оријентисане размене података из система за напредну подршку возачу до инфо-забавних система. Представљени су главни изазови, као и методе и фазе истраживања које су примењене у њиховим решавањима.

Најпре је представљена путања дефинисања комуникационе технологије која највише одговара оваквом решењу. Истраживање је показао да је најповољније усвајање сервисно-оријентисане парадигме кроз имплементацију *SOME/IP* протокола изнад *Ethernet* слоја везе. Ова фаза истраживања је подразумевала теоријски приступ изучавања магистрала, протокола и комуникационих механизама заступљених у аутомобилској индустрији, као и експерименталне имплементације, мерења и поређења перформанси међу различитим протоколима са циљем да најпогоднији буде одабран.

Затим је адресирана хетерогеност система, која додатно отежава изазов усвајања протокола који чини окосницу оваквог решења услед неслагања у парадигмама комуникационих технологија у различитим деловима, односно доменима возила. Стога је за *Android*, који се показује као најзаступљеније решење уз још већи потенцијал раста у употреби, имплементирано усвајање *SOME/IP* механизама у комуникацију остварену *binder* механизмима међупроцесне комуникације типичне за *Android* системе. Изложене су и различите могућности достављања информација до крајњих корисничких апликација или системских модула. За остале оперативне системе заступљене у инфо-забавним доменима нуди се једноставна спрега коју је могуће интегрисати непосредно у одговарајући процес или посредством сервиса који дистрибуира информације путем типичних међупроцесних механизама за жељени систем. Додатно, имајући у виду све значајнији аспект обезбеђивања сигурносних механизама, комуникација дата овим решењем је додатно имплементирана с обзиром на то да *SOME/IP* протокол по основи не решава сигурносна питања. Потом је изложен проблем усвајања решења кроз јединствен протокол погодан за све случаје употребе. С обзиром на то да *SOME/IP* није погодан за пренос великих података, представљен је део решења који то омогућава путем посебно имплементираног канала, односно модула

средњег слоја. Овај канал подржава *UDP, TCP, RTP* и *RTSP* протоколе. Омогућава једноставну програмску спрегу за коришћење и оставља простор за једноставно проширење и придруживање других протокола.

Напослетку је дат и процес аутоматизације, односно генерисања потребне програмске подршке како би допринос дисертацијом представљеног решења био максимизован. У оквиру овога аспекта решења је истражен развој вођен моделима, типични језици за дефинисање комуникационих спрега и примена генеративног програмирања у аутомобилској индустрији. Решењем је омогућено да се из језика за дефинисање спрега коришћеног у домену за напредну подршку (*ARXML*) најпре изврши превођење у друге језике за дефинисање спрега намењене претежно за инфо-забавни домен (*AIDL, FIDL*). Потом се генерише и читав код који имплементира и тестира комуникацију.

Читав развој је вођен по принципима *ASPICE* стандарда, с тим да још једном треба напоменути да формална процена и сертрификација по овим стандардима није рађена. Решење је тестирано на три нивоа тестовима јединица имплементације, интеграционим и функционалним тестовима, док је за превођење између модела урађена и формална верификација. Приказана је и процена решења у виду постигнутих перформанси, најпре кашњења које је уведено решењем за различите случаје употребе. Решење је потврђено радом на неколико развојних платформи и демонстритрано је на међународним конференцијама *CES* у Лас Вегасу и *ZINC* у Новом Саду.

Истраживање представљено овом дисертацијом отворило је доста додатних праваца у којима је могуће даље радити на развоју комуникационог подсистема.

Најпре, пратећи трендове развоја погодно је обезбедити и могућност употребе *DDS* протокола. Иако у решењу није приказан као приступ ком се даје предност, због погодности раздвајања саобраћаја у теме и учесника у домене као и уграђених механизама квалитета услуге неки произвођачи аутомобила се у имплементацијама компонената програмске подршке ослањају на његову употребу. За усвајање коришћења овог протокола у генерисаном решењу потребно је имплементирати и мета-модел језика за дефинисање спрега (у овом случају *OMG IDL*).

Додатно, крајњи циљ је направити надјезик који би био основа за дефинисање тока података у читавом систему. Овај надјезик би било могуће превести у било који од језика за дефинисање спреге и потом генерисати неопходну програмску подршку. Путем овог надјезика било би могуће међусобно преводити моделе који описују спреге у различитим језицима без потребе за имплементацијом превођења за сваки пар појединачно.

Са стране размене тока великих података такође има простора за даљи развој и напредак. Најпре, проширивање имплементације усвајањем *Audio Video Bridging - AVB* стандарда као нижег слоја комуникације омогућавањем спреге са системским управљачким програмима (енгл. *Drivers*) неопходним за правилну употребу *AVB* протокола. Затим, за оптималан рад овог канала за пренос великих података могуће је учинити серијализацију, сегментацију и шифровање конкурентним како би се постигле боље перформансе. Напослетку, имплементација распоређивача при коришћену мрежних утичница како би било могуће приоритизовати размену је такође потенцијални напредак.

8 Литература

[1] Y. Dajsuren and M. van den Brand, *Automotive Systems and Software Engineering*. Springer, 2019.

[2] D. Kenjić and M. Antić, "Connectivity challenges in automotive solutions," *IEEE Consumer Electronics Magazine*, 2022.

[3] A. Frigerio, B. Vermeulen, and K. G. Goossens, "Automotive architecture topologies: Analysis for safety-critical autonomous vehicle applications," *IEEE Access*, vol. 9, pp. 62 837–62 846, 2021.

[4] J. Mossinger, "Software in automotive systems," *IEEE software*, vol. 27, no. 2, p. 92, 2010.

[5] S. Chakraborty, M. Lukasiewycz, C. Buckl, S. Fahmy, N. Chang, S. Park, Y. Kim, P. Leteinturier, and H. Adlkofer, "Embedded systems and software challenges in electric vehicles," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 424–429.

[6] S. Fürst, "Challenges in the design of automotive software," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 256–258.

[7] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf, "Special session: Future automotive systems design: Research challenges and opportunities," in *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2018, pp. 1–7.

[8] G. Niedrist, "Deterministic architecture and middleware for domain control units and simplified integration process applied to adas," in *Fahrerassistenzsysteme 2016*. Springer, 2018, pp. 235–250.

[9] M. Maul, G. Becker, and U. Bernhard, "Service-oriented ee zone architecture key elements for new market segments," *ATZelektronik worldwide*, vol. 13, no. 1, pp. 36–41, 2018.

[10] J. Klaus-Wagenbrenner, "Zonal ee architecture: Towards a fully automotive ethernet-based vehicle infrastructure," in *Proc. Automotive E/E Architecture Technology Innovation Conference*, 2019.

[11] C. Cui, C. Park, and S. Park, "Physical length and weight reduction of humanoid in-robot network with zonal architecture," *Sensors*, vol. 23, no. 5, p. 2627, 2023.

[12] M. Z. Bjelica and Z. Lukac, "Central vehicle computer design: software taking over," *IEEE Consumer Electronics Magazine*, vol. 8, no. 6, pp. 84–90, 2019.

[13] A. Amditis, M. Bimpas, G. Thomaidis, M. Tsogas, M. Netto, S. Mammar, A. Beutner, N. Möhler, T. Wirthgen, S. Zipser *et al.*, "A situation-adaptive lane-keeping support system: Overview of the safelane approach," *IEEE Transactions on Intelligent Transportation Systems*, vol. 11, no. 3, pp. 617–629, 2010.

[14] V. Ilková and A. Ilka, "Legal aspects of autonomous vehicles—an overview," in *2017 21st international conference on process control (PC)*. IEEE, 2017, pp. 428–433.

[15] K. Evans, N. de Moura, S. Chauvier, R. Chatila, and E. Dogan, "Ethical decision making in autonomous vehicles: The av ethics project," *Science and engineering ethics*, vol. 26, pp. 3285–3312, 2020.

[16] A. Hevelke and J. Nida-Rümelin, "Responsibility for crashes of autonomous vehicles: An ethical analysis," *Science and engineering ethics*, vol. 21, pp. 619–630, 2015.

[17] H. Zhu, K.-V. Yuen, L. Mihaylova, and H. Leung, "Overview of environment perception for intelligent vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 10, pp. 2584–2601, 2017.

[18] A. D. Storsæter, K. Pitera, and E. McCormack, "Camera-based lane detection—can yellow road markings facilitate automated driving in snow?" *Vehicles*, vol. 3, no. 4, pp. 661–690, 2021.

[19] J. He, H. Rong, J. Gong, and W. Huang, "A lane detection method for lane departure warning system," in *2010 International Conference on Optoelectronics and Image Processing*, vol. 1. IEEE, 2010, pp. 28–31.

[20] J. Janai, F. Güney, A. Behl, A. Geiger *et al.*, "Computer vision for autonomous vehicles: Problems, datasets and state of the art," *Foundations and Trends® in Computer Graphics and Vision*, vol. 12, no. 1–3, pp. 1–308, 2020.

[21] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213–3223.

[22] S. Stević, M. Dragojević, M. Krunić, and N. Četić, "Vision-based extrapolation of road lane lines in controlled conditions," in *2020 zooming innovation in consumer technologies conference (ZINC)*. IEEE, 2020, pp. 174–177.

[23] G. Kaur and D. Kumar, "Lane detection techniques: A review," *International Journal of Computer Applications*, vol. 112, no. 10, 2015.

[24] A. Scheuer and T. Fraichard, "Collision-free and continuous-curvature path planning for car-like robots," in *Proceedings of international conference on robotics and automation*, vol. 1. IEEE, 1997, pp. 867–873.

[25] Z. Feng, S. Guo, X. Tan, K. Xu, M. Wang, and L. Ma, "Rethinking efficient lane detection via curve modeling," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 17 062–17 070.

[26] I. Bae, J. Moon, H. Park, J. H. Kim, and S. Kim, "Path generation and tracking based on a bezier curve for a steering rate controller of autonomous vehicles," in *16th International IEEE conference on intelligent transportation systems (ITSC 2013)*. IEEE, 2013, pp. 436–441.

[27] X. Huang, F. Gao, G. Xu, N. Ding, L. Li, and Y. Cai, "Extended-search, bézier curve-based lane detection and reconstruction system for an intelligent vehicle," *International Journal of Advanced Robotic Systems*, vol. 12, no. 9, p. 132, 2015.

[28] D. Khosla, "Accurate estimation of forward path geometry using two-clothoid road model," in *Intelligent Vehicle Symposium, 2002. IEEE*, vol. 1. IEEE, 2002, pp. 154–159.

[29] A. Bar Hillel, R. Lerner, D. Levi, and G. Raz, "Recent progress in road and lane detection: a survey," *Machine vision and applications*, vol. 25, no. 3, pp. 727–745, 2014.

[30] R. Laguna, R. Barrientos, L. F. Blazquez, and L. J. Miguel, "Traffic sign recognition application based on image processing techniques," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 104–109, 2014.

[31] H. Luo, Y. Yang, B. Tong, F. Wu, and B. Fan, "Traffic sign recognition using a multi-task convolutional neural network," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 4, pp. 1100–1111, 2017.

[32] W. Shi, M. B. Alawieh, X. Li, and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration*, vol. 59, pp. 148–156, 2017.

[33] H. Ameur, A. Msolli, A. Helali, A. Youssef, and H. Maaref, "Vehicle recognition system based on customised hog for automotive driver assistance system," *International Journal of Intelligent Engineering Informatics*, vol. 5, no. 3, pp. 283–295, 2017.

[34] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[35] F. Baba, D. Kenjić, M. Bjelica, and I. Kaštelan, "Optimizing deep learning based semantic video segmentation on embedded gpus," in *2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin)*. IEEE, 2019, pp. 1–3.

[36] S. Srivastava, R. Singal, and M. Lumba, "Efficient lane detection algorithm using different filtering techniques," *International Journal of Computer Applications*, vol. 88, no. 3, 2014.

[37] D. Kenjic, F. Baba, D. Samardzija, and Z. Kaprocki, "Utilization of the open source datasets for semantic segmentation in automotive vision," in *2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin)*. IEEE, 2019, pp. 420–423.

[38] K. Kuźniar and M. Zając, "Some methods of pre-processing input data for neural networks," *Computer Assisted Methods in Engineering and Science*, vol. 22, no. 2, pp. 141–151, 2017.

[39] S. Hossain and D.-j. Lee, "Deep learning-based real-time multiple-object detection and tracking from aerial imagery via a flying robot with gpu-based embedded devices," *Sensors*, vol. 19, no. 15, p. 3371, 2019.

[40] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[41] M. Krabbes, H.-J. Bohme, V. Stephan, and H.-M. Gross, "Extension of the alvinn-architecture for robust visual guidance of a miniature robot," in *Proceedings Second EUROMICRO Workshop on Advanced Mobile Robots.* IEEE, 1997, pp. 8–14.

[42] S. I. Koval, "Data preparation for neural network data analysis," in *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. IEEE, 2018, pp. 898–901.

[43] D. Dworak, F. Ciepiela, J. Derbisz, I. Izzat, M. Komorkiewicz, and M. Wójcik, "Performance of lidar object detection deep learning architectures based on artificially generated point cloud data from carla simulator," in *2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR).* IEEE, 2019, pp. 600–605.

[44] S. Stević, M. Krunić, M. Dragojević, and N. Kaprocki, "Development and validation of adas perception application in ros environment integrated with carla simulator," in *2019 27th Telecommunications Forum (TELFOR).* IEEE, 2019, pp. 1–4.

[45] Y. Jaafra, J. L. Laurent, A. Deruyver, and M. S. Naceur, "Seeking for robustness in reinforcement learning: application on carla simulator," 2019.

[46] S. Chen, "Kalman filter for robot vision: a survey," *IEEE Transactions on industrial electronics*, vol. 59, no. 11, pp. 4409–4420, 2011.

[47] H. Woo, Y. Ji, H. Kono, Y. Tamura, Y. Kuroda, T. Sugano, Y. Yamamoto, A. Yamashita, and H. Asama, "Lane-change detection based on vehicle-trajectory prediction," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 1109–1116, 2017.

[48] S. Chen, J. Hu, Y. Shi, L. Zhao, and W. Li, "A vision of c-v2x: Technologies, field testing, and challenges with chinese development," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 3872–3881, 2020.

[49] H. Zhou, W. Xu, J. Chen, and W. Wang, "Evolutionary v2x technologies toward the internet of vehicles: Challenges and opportunities," *Proceedings of the IEEE*, vol. 108, no. 2, pp. 308–323, 2020.

[50] P. Hosur, R. B. Shettar, and M. Potdar, "Environmental awareness around vehicle using ultrasonic sensors," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2016, pp. 1154–1159.

[51] P. Furgale, U. Schwesinger, M. Rufli, W. Derendarz, H. Grimmett, P. Mühlfellner, S. Wonneberger, J. Timpner, S. Rottmann, B. Li *et al.*, "Toward automated driving in cities using close-to-market sensors: An overview of the v-charge project," in *2013 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2013, pp. 809–816.

[52] M. Aeberhard, S. Rauch, M. Bahram, G. Tanzmeister, J. Thomas, Y. Pilat, F. Homm, W. Huber, and N. Kaempchen, "Experience, results and lessons learned from automated driving on germany's highways," *IEEE Intelligent transportation systems magazine*, vol. 7, no. 1, pp. 42–57, 2015.

[53] E. Ward and J. Folkesson, "Vehicle localization with low cost radar sensors," in *2016 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2016, pp. 864–870.

[54] J. Ziegler, P. Bender, M. Schreiber, H. Lategahn, T. Strauss, C. Stiller, T. Dang, U. Franke, N. Appenrodt, C. G. Keller *et al.*, "Making bertha drive—an autonomous journey on a historic route," *IEEE Intelligent transportation systems magazine*, vol. 6, no. 2, pp. 8–20, 2014.

[55] J. Steinbaeck, C. Steger, G. Holweg, and N. Druml, "Next generation radar sensors in automotive sensor fusion systems," in *2017 Sensor Data Fusion: Trends, Solutions, Applications (SDF)*. IEEE, 2017, pp. 1–6.

[56] R. Thakur, "Scanning lidar in advanced driver assistance systems and beyond: Building a road map for next-generation lidar technology," *IEEE Consumer Electronics Magazine*, vol. 5, no. 3, pp. 48–54, 2016.

[57] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of autonomous car—part ii: A case study on the implementation of an autonomous driving system based on distributed architecture," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 8, pp. 5119–5132, 2015.

[58] J. Wei, J. M. Snider, J. Kim, J. M. Dolan, R. Rajkumar, and B. Litkouhi, "Towards a viable autonomous driving research platform," in *2013 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2013, pp. 763–770.

[59] "Smart camera for automotive," https://www.renesas.com/us/en/application/automotive/adas-autonomous/smart-camera-automotive, , Renesas Electronics Corporation, [Online], Присутпљено: 2023-01-20.

[60] F. Reway, W. Huber, and E. P. Ribeiro, "Test methodology for vision-based adas algorithms with an automotive camera-in-the-loop," in *2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. IEEE, 2018, pp. 1–7.

[61] A. Greiner, F. Petrot, M. Carrier, M. Benabdenbi, R. Chotin-Avot, and R. Labayrade, "Mapping an obstacles detection, stereo vision-based, software application on a multi-processor system-on-chip," in *2006 IEEE Intelligent Vehicles Symposium*. IEEE, 2006, pp. 370–376.

[62] J. Khan, S. Niar, A. Menhaj, Y. Elhillali, and J. L. Dekeyser, "An mpsoc architecture for the multiple target tracking application in driver assistant system," in *2008 International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 2008, pp. 126–131.

[63] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.

[64] Ž. Lukač, I. Kaštelan, M. Vranješ, and B. M. Todorović, "Amv alpha learning platform for automotive embedded software engineering," *IEEE Transactions on Learning Technologies*, vol. 14, no. 3, pp. 292–298, 2021.

[65] N. Perkovic, M. Pranjkic, I. Kolak, and G. Velikic, "System for automotive machine vision," in *2017 IEEE 7th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*. IEEE, 2017, pp. 243–245.

[66] R. Bleuse, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling independent tasks on multi-cores with gpu accelerators," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 6, pp. 1625–1638, 2015.

[67] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, 2013.

[68] M. H. Jooybar, "Deterministic execution on gpu architectures," Ph.D. dissertation, University of British Columbia, 2013.

[69] G. A. Elliott, "Real-time scheduling for gpus with applications in advanced automotive systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2015.

[70] V. Miller, "Determinism in gpu programs-real time applications on the nvidia jetson tk1," 2016.

[71] D. Kenjić, M. Milošević, M. Antić, and N. Teslić, "One solution for deterministic scheduling on gpu for automotive algorithms," in *2021 Zooming Innovation in Consumer Technologies Conference (ZINC)*. IEEE, 2021, pp. 264–268.

[72] B. Poudel, N. K. Giri, and A. Munir, "Design and comparative evaluation of gpgpu-and fpga-based mpsoc ecu architectures for secure, dependable, and real-time automotive cps," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017, pp. 29–36.

[73] V. Bandur, G. Selim, V. Pantelic, and M. Lawford, "Making the case for centralized automotive e/e architectures," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 2, pp. 1230–1245, 2021.

[74] I. T. AG, "32-bit tricore™ aurix™– tc3xx," https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/, 2017, , [Online], Присутпљено: 2023-05-26.

[75] R. E. Corporation, "Rh850/p1x-c application note," https://www.renesas.com/eu/en/doc/products/mpumcu/apn/rh850/001/R01AN4327ED0300_RH850_P1x-C.pdf, 2018, , [Online], Присутпљено: 2023-05-26.

[76] M. Staron and D. Durisic, "Autosar standard," in *Automotive Software Architectures*. Springer, 2017, pp. 81–116.

[77] "Autosar manifest," https://www.autosar.org/fileadmin/user_upload/standards/adaptive/19-11/AUTOSAR_TPS_ManifestSpecification.pdf, AUTOSAR organisation, Explanatory doc., 2019.

[78] M. Oertel and B. Zimmer, "More performance with autosar adaptive," *ATZelectronics worldwide*, vol. 14, no. 5, pp. 36–39, 2019.

[79] G. B. Meike and L. Schiefer, *Inside the Android OS: Building, Customizing, Managing and Operating Android System Services (Android Deep Dive) 1st Edition*. Addison-Wesley Professional, 2021.

[80] B. QNX, "Qnx car platform for infotainment," https://blackberry.qnx.com/content/dam/qnx/products/qnxcar/qnx-car-infotaintment-product-brief.pdf, 2017, , [Online], Присутпљено: 2023-05-26.

[81] Z. Zhang, Y. Liu, J. Chen, Z. Qi, Y. Zhang, and H. Liu, "Performance analysis of open-source hypervisors for automotive systems," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2021, pp. 530–537.

[82] M. Ponos, N. Lazic, M. Bjelica, T. Andjelic, and M. Manic, "One solution for integrating graphics in vehicle digital cockpit," in *2021 29th Telecommunications Forum (TELFOR)*. IEEE, 2021, pp. 1–3.

[83] B. Kockoth, "Integration and isolation safety aspects in the cockpit of the future," *ATZelektronik worldwide*, vol. 11, no. 1, pp. 28–31, 2016.

[84] P. Sivakumar, R. S. Devi, A. N. Lakshmi, B. VinothKumar, and B. Vinod, "Automotive grade linux software architecture for automotive infotainment system," in *2020 International Conference on Inventive Computation Technologies (ICICT)*. IEEE, 2020, pp. 391–395.

[85] S. Usorac, D. Bogdanovic, D. Peric, and Z. Lukac, "Adding android capabilities in automotive linux infotainment: available virtualization technologies," in *2021 29th Telecommunications Forum (TELFOR)*. IEEE, 2021, pp. 1–4.

[86] S. Usorac and B. Pavkovic, "Linux container solution for running android applications on an automotive platform," in *2021 Zooming Innovation in Consumer Technologies Conference (ZINC)*. IEEE, 2021, pp. 209–213.

[87] A. N. Update, "Future of automotive hmi: How technology is driving innovation, top 10 companies," https://automotive-industry-news-trends.blogspot.com/2023/06/future-of-automotive-hmi-how-technology.html, 2023, , [Online], Присутпљено: 2023-06-15.

[88] Infopulse, "Future of digital car cockpit: Top infotainment trends for oems to monitor," https://www.infopulse.com/blog/the-future-of-the-digital-car-cockpit-top-infotainment-trends-for-oems-to-monitor, 2019, , [Online], Присутпљено: 2023-06-15.

[89] Deloitte, "Deloitte automotive consumer study: Advanced vehicle technologies," https://www2.deloitte.com/content/dam/Deloitte/tr/Documents/consumer-business/Deloitte-Automotive-Consumer-Study-2019.pdf, Deloitte, Tech. Rep., 2019.

[90] A. P. de Miranda and J. S. Germaine, "New direction and trends for vehicle entertainment systems," SAE Technical Paper, Tech. Rep., 2012.

[91] G. Jaiswal, "Android in-vehicle infotainment system," *Int. J. Innovat. Res. Electron. Commun*, vol. 1, no. 4, pp. 12–16, 2014.

[92] Q. Rao, C. Grünler, M. Hammori, and S. Chakraborty, "Design methods for augmented reality in-vehicle infotainment systems," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[93] Google, "Android for cars app library design guidelines," https://developer.android.com/static/training/cars/Android%20for%20Cars%20App%20Library%20design%20guidelines.pdf, 2020, , [Online], Присутпљено: 2023-04-11.

[94] ——, "Design settings," https://developers.google.com/cars/design/automotive-os/apps/media/create-your-app/design-settings, 2020, , [Online], Присутпљено: 2023-04-11.

[95] G. Macario, M. Torchiano, and M. Violante, "An in-vehicle infotainment software architecture based on google android," in *2009 IEEE International Symposium on Industrial Embedded Systems*. IEEE, 2009, pp. 257–260.

[96] A. Moiz and M. H. Alalfi, "A survey of security vulnerabilities in android automotive apps," in *Proceedings of the 3rd International Workshop on Engineering and Cybersecurity of Critical Systems*, 2022, pp. 17–24.

[97] N. Pajic and M. Bjelica, "Integrating android to next generation vehicles," in *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*. IEEE, 2018, pp. 152–155.

[98] T. Schreiber, "Android binder," http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf, 2011, , [Online], Присутпљено: 2023-04-11.

[99] Google, "Android interface definition language(aidl)," https://developer.android.com/guide/components/aidl, 2022, , [Online], Присутпљено: 2023-04-11.

[100] C. Organization, "Covesa," https://github.com/COVESA, 2022, , [Online], Присутпљено: 2023-03-09.

[101] B.-H. Cho and H.-H. Ahn, "Analysis and design of connected car infotainment system," *The Journal of The Institute of Internet, Broadcasting and Communication*, vol. 17, no. 5, pp. 17–23, 2017.

[102] C. Organization, "vsomeip," https://github.com/COVESA/vsomeip, 2022, , [Online], Присутпљено: 2023-03-09.

[103] ——, "Commonapi c++ tutorial," https://at.projects.genivi.org/wiki/pages/viewpage.action?pageId=5472311, 2022, , [Online], Присутпљено: 2023-03-09.

[104] ——, "Welcome to franca!" https://github.com/franca/franca, 2022, , [Online], Присутпљено: 2023-03-09.

[105] D. Kenjić and M. Antić, "Overview of the swcs communication mechanisms in autonomous vehicles," in *2021 IEEE 11th International Conference on Consumer Electronics (ICCE-Berlin)*. IEEE, 2021, pp. 1–3.

[106] J. Martinez, I. Sañudo, and M. Bertogna, "End-to-end latency characterization of task communication models for automotive systems," *Real-Time Systems*, vol. 56, no. 3, pp. 315–347, 2020.

[107] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 2, pp. 1–27, 2011.

[108] M. Urbina and R. Obermaisser, "Multi-core architecture for autosar based on virtual electronic control units," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2015, pp. 1–5.

[109] S. V. Tota, M. R. Casu, M. R. Roch, L. Rostagno, and M. Zamboni, "Medea: a hybrid shared-memory/message-passing multiprocessor noc-based architecture," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 45–50.

[110] S.-J. Chen, A.-Y. A. Wu, and J. Xu, "Networks-on-chip: architectures, design methodologies, and case studies," *Journal of Electrical and Computer Engineering*, vol. 2012, 2012.

[111] W.-C. Tsai, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen, "Networks on chips: structure and design methodologies," *Journal of Electrical and Computer Engineering*, vol. 2012, 2012.

[112] A. Lankes, T. Wild, and A. Herkersdorf, "Hierarchical nocs for optimized access to shared memory and io resources," in *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. IEEE, 2009, pp. 255–262.

[113] N. Concer, L. Bononi, M. Soulié, R. Locatelli, and L. P. Carloni, "The connection-then-credit flow control protocol for heterogeneous multicore systems-on-chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 6, pp. 869–882, 2010.

[114] N. Navet and F. Simonot-Lion, "In-vehicle communication networks-a historical perspective and review," *Industrial Communication Technology Handbook*, vol. 96, pp. 1204–1223, 2013.

[115] "Road vehicles – Controller area network (CAN)," International Organization for Standardization, Standard, 2016.

[116] W. Zeng, M. A. Khalid, and S. Chowdhury, "In-vehicle networks outlook: Achievements and challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1552–1571, 2016.

[117] "Road vehicles – FlexRay communications system," International Organization for Standardization, Standard, 2013.

[118] S. Tuohy, M. Glavin, E. Jones, M. Trivedi, and L. Kilmartin, "Next generation wired intra-vehicle networks, a review," in *2013 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2013, pp. 777–782.

[119] "Road vehicles – Local Interconnect Network (LIN)," International Organization for Standardization, Standard, 2016.

[120] G. Steiger, "Towards a new mobility–the driver becomes the passenger," in *Fahrerassistenzsysteme 2016*. Springer, 2018, pp. 1–17.

[121] D. Kenjić, M. Antić, and M. Bjelica, "Evaluation of ethernet subsystem for domain controller in autonomous vehicles," in *2021 Zooming Innovation in Consumer Technologies Conference (ZINC)*. IEEE, 2021, pp. 59–63.

[122] P. Hank, T. Suermann, and S. Müller, "Automotive ethernet, a holistic approach for a next generation in-vehicle networking standard," in *Advanced Microsystems for Automotive Applications 2012*. Springer, 2012, pp. 79–89.

[123] "IEEE Standard for Ethernet-Amendment 1: Physical Layer Specifications and Management Parameters for 100 Mb/s Operation over a Single Balanced Twisted Pair Cable (100BASE-T1)," Institute of Electrical and Electronics Engineers, Standard, 2015.

[124] "IEEE Standard for Ethernet-Amendment 8:Physical Layer Specifications and Management Parameters for 2.5 Gb/s, 5 Gb/s, and 10 Gb/s Automotive Electrical Ethernet," Institute of Electrical and Electronics Engineers, Standard, 2020.

[125] O. Alparslan, S. Arakawa, and M. Murata, "Next generation intra-vehicle backbone network architectures," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2021, pp. 1–7.

[126] L. Zhao, F. He, E. Li, and J. Lu, "Comparison of time sensitive networking (tsn) and ttethernet," in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. IEEE, 2018, pp. 1–7.

[127] P. B., *Racunarske mreze, magistrale i protokoli u automobilu*. Fakultet tehnickih nauka, Univerzitet u Novom Sadu, 2022.

[128] "IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications," Institute of Electrical and Electronics Engineers, Standard, 2020.

[129] "IEEE Standard for Local and metropolitan area networks– Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams," Institute of Electrical and Electronics Engineers, Standard, 2009.

[130] "IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks - Amendment: 9: Stream Reservation Protocol (SRP)," Institute of Electrical and Electronics Engineers, Standard, 2010.

[131] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic," Institute of Electrical and Electronics Engineers, Standard, 2015.

[132] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks," Institute of Electrical and Electronics Engineers, Standard, 2014.

[133] M. Sandic, B. Pavkovic, and N. Teslic, "Ttethernet mixed-critical communication: Overview and impact of faulty switches," *IEEE Consumer Electronics Magazine*, vol. 9, no. 4, pp. 97–103, 2020.

[134] "IEEE Standard for Ethernet - Amendment 5: Physical Layer Specifications and Management Parameters for 10 Mb/s Operation and Associated Power Delivery over a Single Balanced Pair of Conductors," Institute of Electrical and Electronics Engineers, Standard, 2019.

[135] R. N. K, "Data traffic in new generation vehicles," *CSI Communications*, vol. 18, no. 3, pp. 8–12, 2012.

[136] M. Rahmani, R. Steffen, K. Tappayuthpijarn, E. Steinbach, and G. Giordano, "Performance analysis of different network topologies for in-vehicle audio and video communication," in *2008 4th International Telecommunication Networking Workshop on QoS in Multiservice IP Networks*. IEEE, 2008, pp. 179–184.

[137] D. Teshler and B. Andretzky, "Fpgas for the zonal service-oriented network architecture in cars," *ATZelectronics worldwide*, vol. 16, no. 10, pp. 8–13, 2021.

[138] J. R. Seyler, T. Streichert, M. Glaß, N. Navet, and J. Teich, "Formal analysis of the startup delay of some/ip service discovery," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 49–54.

[139] "Some/ip protocol specification," https://www.autosar.org/fileadmin/standards/foundation/1-3/AUTOSAR_PRS_SOMEIPProtocol.pdf, AUTOSAR organisation, Explanatory doc., 2019.

[140] "Specification of some/ip transformer," https://www.autosar.org/fileadmin/standards/classic/4-2/AUTOSAR_SWS_SOMEIPTransformer.pdf, AUTOSAR organisation, Explanatory doc., 2019.

[141] "Some/ip service discovery protocol specification," https://www.autosar.org/fileadmin/standards/foundation/1-3/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf, AUTOSAR organisation, Explanatory doc., 2019.

[142] K. Czarnecki and U. W. Eisenecker, "Components and generative programming," in *Software Engineering—ESEC/FSE'99*. Springer, 1999, pp. 2–19.

[143] K. Czarnecki, "Overview of generative software development," in *Unconventional Programming Paradigms: International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers*. Springer, 2005, pp. 326–341.

[144] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti *et al.*, "Go meta! a case for generative programming and dsls in performance critical systems," *1st Summit on Advances in Programming Languages (SNAPL 2015)*, vol. 32, pp. 238–261, 2015.

[145] G. Guta and D. Varro, "Model-to-text transformation modification by examples," Ph.D. dissertation, Technical-scientific Faculty, Linz, Germany, 2012.

[146] L. George, A. Wider, and M. Scheidgen, "Type-safe model transformation languages as internal dsls in scala," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2012, pp. 160–175.

[147] J. Sánchez Cuadrado, "Towards a family of model transformation languages," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2012, pp. 176–191.

[148] M. Biehl, "Supporting model evolution in model-driven development of automotive embedded system," Ph.D. dissertation, US-AB, 2010.

[149] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3. USA, 2003, pp. 1–17.

[150] T. Wang, S. Truptil, and F. Benaben, "An automatic model-to-model mapping and transformation methodology to serve model-based systems engineering," *Information Systems and e-Business Management*, vol. 15, no. 2, pp. 323–376, 2017.

[151] J. M. Küster, S. Sendall, M. Wahler *et al.*, "Comparing two model transformation approaches," in *Proc. Workshop on OCL and Model Driven Engineering*, 2004.

[152] "Road vehicles – Functional safety," International Organization for Standardization, Standard, 2018.

[153] "Road vehicles – Safety of the intended functionality," International Organization for Standardization, Standard, 2022.

[154] "Automotive SPICE Process Assessment / Reference Model," VDA QMC Working Group 13 / Automotive SIG, Process Assessment and Reference Model, 2017.

[155] L. Perneel, H. Fayyad-Kazan, and M. Timmerman, "Can android be used for real-time purposes?" in *2012 International Conference on Computer Systems and Industrial Informatics*. IEEE, 2012, pp. 1–6.

[156] M. Unger, G. Fries, T. Steinecke, C. Waghmare, and R. Ramaswamy, "Functional safety test strategy for automotive microcontrollers during electro-magnetic compatibility characterization," in *2019 12th International Workshop on the Electromagnetic Compatibility of Integrated Circuits (EMC Compo)*. IEEE, 2019, pp. 49–51.

[157] C. Ebert and J. Favaro, "Automotive software," *IEEE Software*, vol. 34, no. 03, pp. 33–39, 2017.

[158] K.-T. Cho and K. G. Shin, "Error handling of in-vehicle networks makes them vulnerable," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1044–1055.

[159] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote attacks on automated vehicles sensors: Experiments on camera and lidar," *Black Hat Europe*, vol. 11, no. 2015, p. 995, 2015.

[160] S. Lv, S. Nie, L. Liu, and W. Lu, "Car hacking research: remote attack tesla motors," *Keen Security Lab of Tencent, sl*, 2016.

[161] C. Joe Warminsky, "Tesla model 3's onboard browser attacked successfully at pwn2own," https://cyberscoop.com/tesla-hacked-pwn-2-own-2019/, 2019, , [Online], Присутпљено: 2023-05-20.

[162] R. Tod Beardsley, "R7-2017-02: Hyundai blue link potential info disclosure (fixed)," https://www.rapid7.com/blog/post/2017/04/25/r7-2017-02-hyundai-blue-link-potential-info-disclosure-fixed/, 2017, , [Online], Присутпљено: 2023-05-20.

[163] W. Andy Greenberg, "Hackers remotely kill a jeep on the highway—with me in it," https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/, 2015, , [Online], Присутпљено: 2023-05-20.

[164] J. Schneider and T. Nett, "Safety issues of integrating ivi and adas functionality via running linux and autosar in parallel on a dual-core-system," *Automotive-Safety & Security 2014*, 2015.

[165] H. Yu and C.-W. Lin, "Security concerns for automotive communication and software architecture," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2016, pp. 600–603.

[166] W. Wu, R. Li, G. Xie, J. An, Y. Bai, J. Zhou, and K. Li, "A survey of intrusion detection for in-vehicle networks," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 3, pp. 919–933, 2019.

[167] M. Iorio, M. Reineri, F. Risso, R. Sisto, and F. Valenza, "Securing some/ip for in-vehicle service protection," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 11, pp. 13 450–13 466, 2020.

[168] M. Iorio, A. Buttiglieri, M. Reineri, F. Risso, R. Sisto, and F. Valenza, "Protecting in-vehicle services: Security-enabled some/ip middleware," *IEEE Vehicular Technology Magazine*, vol. 15, no. 3, pp. 77–85, 2020.

[169] E. Türk and M. Challenger, "An android-based iot system for vehicle monitoring and diagnostic," in *2018 26th Signal Processing and Communications Applications Conference (SIU)*. IEEE, 2018, pp. 1–4.

[170] Z. Zuo, S. Yang, B. Ma, B. Zou, Y. Cao, Q. Li, S. Zhou, and J. Li, "Design of a canfd to some/ip gateway considering security for in-vehicle networks," *Sensors*, vol. 21, no. 23, p. 7917, 2021.

[171] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *2016 46th annual IEEE/IFIP international conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2016, pp. 215–217.

[172] R. Manoharan and S. Chandrakala, "Android opencv based effective driver fatigue and distraction monitoring system," in *2015 International Conference on Computing and Communications Technologies (ICCCT)*. IEEE, 2015, pp. 262–266.

[173] R. Jabbar, M. Shinoy, M. Kharbeche, K. Al-Khalifa, M. Krichen, and K. Barkaoui, "Driver drowsiness detection model using convolutional neural networks techniques for android application," in *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*. IEEE, 2020, pp. 237–242.

[174] J. Guo, B. Song, Y. He, F. R. Yu, and M. Sookhak, "A survey on compressed sensing in vehicular infotainment systems," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2662–2680, 2017.

[175] S. Aust, "Paving the way for connected cars with adaptive autosar and agl," in *2018 IEEE 43rd Conference on Local Computer Networks Workshops (LCN Workshops)*. IEEE, 2018, pp. 53–58.

[176] M. Kotur, N. Lukić, M. Krunić, and G. Velikić, "One solution of camera service in autosar adaptive environment," in *2020 IEEE 10th International Conference on Consumer Electronics (ICCE-Berlin)*. IEEE, 2020, pp. 1–5.

[177] K. Omerovic, J. Janjatovic, M. Milosevic, and T. Maruna, "Supporting sensor fusion in next generation android in-vehicle infotainment units," in *2016 IEEE 6th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*. IEEE, 2016, pp. 187–189.

[178] Google, "Vehicle hardware abstraction layer > vehicle properties," https://source.android.com/docs/devices/automotive/vhal/properties, 2023, , [Online], Присутпљено: 2023-06-26.

[179] ——, "Neural networks api," https://developer.android.com/ndk/guides/neuralnetworks, 2023, , [Online], Присутпљено: 2023-06-26.

[180] M. Milosevic, M. Z. Bjelica, T. Maruna, and N. Teslic, "Software platform for heterogeneous in-vehicle environments," *IEEE Transactions on Consumer Electronics*, vol. 64, no. 2, pp. 213–221, 2018.

[181] L. L. Bello, R. Mariani, S. Mubeen, and S. Saponara, "Recent advances and trends in on-board embedded and networked automotive systems," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1038–1051, 2018.

[182] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Caad: Computer architecture for autonomous driving," *arXiv preprint arXiv:1702.01894*, 2017.

[183] C. Buckl, M. Geisinger, D. Gulati, F. J. Ruiz-Bertol, and A. Knoll, "Chromosome: a run-time environment for plug & play-capable embedded real-time systems," *ACM SIGBED Review*, vol. 11, no. 3, pp. 36–39, 2014.

[184] A. Kampmann, B. Alrifaee, M. Kohout, A. Wüstenberg, T. Woopen, M. Nolte, L. Eckstein, and S. Kowalewski, "A dynamic service-oriented software architecture for highly automated vehicles," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2019, pp. 2101–2108.

[185] "Unified sotfware," https://cariad.technology/de/en/solutions/unified-software.html, , CARIAD, a VolksWagen group company, [Online], Присутпљено: 2023-02-24.

[186] F. Knapp, "Challenges for audi in the e-mobility business in china," Ph.D. dissertation, Technische Hochschule Ingolstadt, 2021.

[187] K. C. Felser, "The impact of digital technologies on it sourcing strategies in the german automotive industry," in *Handbook of Research on Digital Transformation, Industry Use Cases, and the Impact of Disruptive Technologies*. IGI Global, 2022, pp. 383–408.

[188] "Renault group and google accelerate partnership to develop the vehicle of tomorrow and strengthen renault group's digital transformation," https://media.renaultgroup.com/renault-group-and-google-accelerate-partnership-to-develop-the-vehicle-of-tomorrow-and-strengthen
, Renault Group, [Online], Присутпљено: 2023-02-24.

[189] "Qualcomm puts cockpit and adas functions on one chip," https://www.electronicsweekly.com/news/business/qualcomm-puts-cockpit-and-adas-functions-on-one-chip-2023-01/, , M. Davids, Elecrtonics Weekly, [Online], Присутпљено: 2023-02-24.

[190] "Franca / ara::com interoperability," https://wiki.covesa.global/download/attachments/364361/GENIVI%20Franca-ARA-COM-tech-brief-20181217-V4.pdf?version=1&modificationDate=1551377097000&api=v2, Technology Brief doc., 2018, , GENIVI, Itemis, ITK engnieering.

[191] "Franca ara tools," https://github.com/COVESA/franca_ara_tools, , COVESA, [Online], Присутпљено: 2023-01-20.

[192] M. Bellanger and E. Marmounier, "Service oriented architecture: impacts and challenges of an architecture paradigm change," in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, 2020.

[193] S. De, M. Niklas, B. Rooney, J. Mottok, and P. Brada, "Towards translation of semantics of automotive interface description models from franca to autosar frameworks: An approach using semantic synergies," in *2019 International Conference on Applied Electronics (AE)*. IEEE, 2019, pp. 1–6.

[194] D. Kenjić, D. Živkov, and M. Antić, "Automated data transfer from adas to android-based ivi domain over some/ip," *IEEE Transactions on Intelligent Vehicles*, 2023.

[195] "Ieee 802.3-2022: Ieee standard for ethernet," https://standards.ieee.org/ieee/802.3/10422/, , IEEE, [Online], Присутпљено: 2023-01-20.

[196] "Ieee 802.1q-2018: Ieee standard for local and metropolitan area networks–bridges and bridged networks," https://standards.ieee.org/ieee/802.3/10422/, , IEEE, [Online], Приступљено: 2023-01-20.

[197] J. Vuckovic, "Hibridni model upravljanja razvojem softvera u automobilskoj industriji," Fakultet tehnickih nauka, Univerzitet u Novom Sadu, 2023.

[198] R. Messnarz, D. Ekert, T. Zehetner, and L. Aschbacher, "Experiences with aspice 3.1 and the vda automotive spice guidelines–using advanced assessment systems," in *Systems, Software and Services Process Improvement: 26th European Conference, EuroSPI 2019, Edinburgh, UK, September 18–20, 2019, Proceedings 26*. Springer, 2019, pp. 549–562.

[199] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[200] J. Harrison, "Formal proof—theory and practice," *Notices of the AMS*, vol. 55, no. 11, pp. 1395–1406, 2008.

[201] J. McCarthy, "Towards a mathematical science of computation," in *Program Verification: Fundamental Issues in Computer Science*. Springer, 1993, pp. 35–56.

[202] O. Semeráth, Á. Barta, Á. Horváth, Z. Szatmári, and D. Varró, "Formal validation of domain-specific languages with derived features and well-formedness constraints," *Software & Systems Modeling*, vol. 16, pp. 357–392, 2017.

[203] G. Eakman, H. Reubenstein, T. Hawkins, M. Jain, and P. Manolios, "Practical formal verification of domain-specific language applications," in *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings 7*. Springer, 2015, pp. 443–449.

[204] "Adaptive platform release overview," https://www.autosar.org/fileadmin/standards/R21-11/AP/AUTOSAR_TR_AdaptivePlatformReleaseOverview.pdf, AUTOSAR organisation, Explanatory doc., 2021.

Овај Образац чини саставни део докторске дисертације, односно докторског уметничког пројекта који се брани на Универзитету у Новом Саду. Попуњен Образац укоричити иза текста докторске дисертације, односно докторског уметничког пројекта.

План третмана података

Назив пројекта/истраживања
Аутоматско генерисање програмске подршке за сервисно-оријентисану комуникацију између инфо-забавних и система за напредну подршку возачу у аутомобилима
Назив институције/институција у оквиру којих се спроводи истраживање
а) Факултет техничких наука, Универзитет у Новом Саду **б) Институт РТ-РК, Нови Сад**
Назив програма у оквиру ког се реализује истраживање
Истраживање се врши у оквиру израде докторске дисертације на студијском програму Рачунарство и аутоматика.
1. Опис података

1.1 Врста студије

Укратко описати тип студије у оквиру које се подаци прикупљају

У овој студији нису прикупљани подаци. Истраживање је базирано на доступним подацима чији су извори наведени у дисертацији.

1.2 Врсте података

а) квантитативни

б) квалитативни

1.3. Начин прикупљања података

а) анкете, упитници, тестови

б) клиничке процене, медицински записи, електронски здравствени записи

в) генотипови: навести врсту _____

г) административни подаци: навести врсту _____

д) узорци ткива: навести врсту _____

ђ) снимци, фотографије: навести врсту _____

е) текст, навести врсту _____

ж) мапа, навести врсту _____

з) остало: описати _____

1.3 Формат података, употребљене скале, количина података

1.3.1 Употребљени софтвер и формат датотеке:

а) Excel фајл, датотека _____

b) SPSS фајл, датотека _____

c) PDF фајл, датотека _____

d) Текст фајл, датотека _____

e) JPG фајл, датотека _____

f) Остало, датотека _____

1.3.2. Број записа (код квантитативних података)

а) број варијабли _____

б) број мерења (испитаника, процена, снимака и сл.) _____

1.3.3. Поновљена мерења

а) да

б) не

Уколико је одговор да, одговорити на следећа питања:

а) временски размак између поновљених мера је _____

б) варијабле које се више пута мере односе се на _____

в) нове верзије фајлова који садрже поновљена мерења су именоване као _____

Напомене: _____

Да ли формати и софтвер омогућавају дељење и дугорочну валидност података?

а) да

б) не

Ако је одговор не, образложити _____

_____

2. Прикупљање података

2.1 Методологија за прикупљање/генерисање података

2.1.1. У оквиру ког истраживачког нацрта су подаци прикупљени?

а) експеримент, навести тип _____

б) корелационо истраживање, навести тип _____

ц) анализа текста, навести тип _____

д) остало, навести шта _____

2.1.2 Навести врсте мерних инструмената или стандарде података специфичних за одређену научну дисциплину (ако постоје).

_____

_____

2.2 Квалитет података и стандарди

2.2.1. Третман недостајућих података

а) Да ли матрица садржи недостајуће податке? Да Не

Ако је одговор да, одговорити на следећа питања:

а) Колики је број недостајућих података? _____

б) Да ли се кориснику матрице препоручује замена недостајућих података? Да Не

в) Ако је одговор да, навести сугестије за третман замене недостајућих података

_____

2.2.2. На који начин је контролисан квалитет података? Описати

_____

_____

2.2.3. На који начин је извршена контрола уноса података у матрицу?

_____

_____

3. Третман података и пратећа документација

3.1. Третман и чување података

3.1.1. Подаци ће бити депоновани у _____

3.1.2. URL адреса _____

3.1.3. DOI _____

3.1.4. Да ли ће подаци бити у отвореном приступу?

а) Да

б) Да, али после ембарга који ће трајати до _____

в) Не

Ако је одговор не, навести разлог _____

3.1.5. Подаци неће бити депоновани у репозиторијум, али ће бити чувани.

Образложење

_____

_____

3.2 Метаподаци и документација података

3.2.1. Који стандард за метаподатке ће бити примењен? _____

3.2.2. Навести метаподатке на основу којих су подаци депоновани у репозиторијум.

_____

_____

Ако је потребно, навести методе које се користе за преузимање података, аналитичке и процедуралне информације, њихово кодирање, детаљне описе варијабли, записа итд.

_____

3.3 Стратегија и стандарди за чување података

3.3.1. До ког периода ће подаци бити чувани у репозиторијуму? _____

3.3.2. Да ли ће подаци бити депоновани под шифром? Да Не

3.3.3. Да ли ће шифра бити доступна одређеном кругу истраживача? Да Не

3.3.4. Да ли се подаци морају уклонити из отвореног приступа после извесног времена? Да Не

Образложити

4. Безбедност података и заштита поверљивих информација

Овај одељак МОРА бити попуњен ако ваши подаци укључују личне податке који се односе на учеснике у истраживању. За друга истраживања треба такође размотрити заштиту и сигурност података.

4.1 Формални стандарди за сигурност информација/података

Истраживачи који спроводе испитивања с људима морају да се придржавају Закона о заштити података о личности (`https://www.paragraf.rs/propisi/zakon_o_zastiti_podataka_o_licnosti.html`) и одговарајућег институционалног кодекса о академском интегритету.

4.1.2. Да ли је истраживање одобрено од стране етичке комисије? Да Не

Ако је одговор Да, навести датум и назив етичке комисије која је одобрила истраживање

4.1.3. Да ли подаци укључују личне податке учесника у истраживању? Да Не

Ако је одговор да, наведите на који начин сте осигурали поверљивост и сигурност информација везаних за испитанике:

а) Подаци нису у отвореном приступу

б) Подаци су анонимизирани

ц) Остало, навести шта

5. Доступност података

5.1. Подаци ће бити

а) јавно доступни

б) доступни само уском кругу истраживача у одређеној научној области

ц) затворени

Ако су подаци доступни само уском кругу истраживача, навести под којим условима могу да их користе:

Ако су подаци доступни само уском кругу истраживача, навести на који начин могу приступити подацима:

5.4. Навести лиценцу под којом ће прикупљени подаци бити архивирани.

6. Улоге и одговорност

6.1. Навести име и презиме и мејл адресу власника (аутора) података

6.2. Навести име и презиме и мејл адресу особе која одржава матрицу с подацима

6.3. Навести име и презиме и мејл адресу особе која омогућује приступ подацима другим истраживачима