Simuleringsbaserad verifiering av fordonsbeteenden

Ett CAN-baserat verktyg för tidig testning i labbmiljö

Examensarbete inom högskoleprogrammet i Mekatronik

SIMON CARLÉN
ALVIN STALLGÅRD

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg 2026
www.chalmers.se

Examensarbete 2026

Simuleringsbaserad verifiering av fordonsbeteenden

Ett CAN-baserat verktyg för tidig testning i labbmiljö

SIMON CARLÉN
ALVIN STALLGÅRD

CHALMERS

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
Göteborg 2026

Simuleringsbaserad verifiering av fordonsbeteenden
Ett CAN-baserat verktyg för tidig testning i labbmiljö

SIMON CARLÉN
ALVIN STALLGÅRD

Handledare: Jonas Almström Duregård, CSE
Examinator: Wolfgang Ahrendt, CSE

Examensarbete 2026
Institutionen för data- och informationsteknik
Chalmers tekniska högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslagsbild: En Volvo-buss i stadsmiljö som visualiserar projektets koppling till fordonsutveckling, simulering och CAN-baserad verifiering.

Skriven i LaTeX
Göteborg 2026

Simuleringsbaserad verifiering av fordonsbeteenden
Ett CAN-baserat verktyg för tidig testning i labbmiljö
Simon Carlén & Alvin Stallgård
Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola

Sammanfattning

Fordonsindustrin befinner sig i en utveckling där kortare utvecklingscykler, ökad mjukvarukomplexitet och högre krav på verifiering driver ett växande behov av simuleringsbaserade testmetoder. Ett centralt arbetssätt i denna utveckling är shift-left testing, där verifiering och validering flyttas till tidigare faser i utvecklingsprocessen. Detta kan minska beroendet av fysiska testfordon och bidra till mer flexibla, repeterbara och kostnadseffektiva tester.

Syftet med detta examensarbete är att utveckla ett CAN-baserat simuleringsverktyg för tidig verifiering av fordonsbeteenden i labbmiljö. Arbetet har genomförts i samarbete med Volvo Bussar och fokuserar på att undersöka hur en mjukvarubaserad fordonsmodell kan användas för att generera simulerade fordons- och sensorsignaler till en befintlig testmiljö.

Resultatet är en fungerande prototyp bestående av tre huvudsakliga delar: en fordonsmodell, ett grafiskt användargränssnitt och en modul för CAN-kommunikation. Fordonsmodellen simulerar centrala tillstånd såsom hastighet, acceleration, orientering, växel, motorvarvtal och hjulrelaterade variabler. Det grafiska användargränssnittet möjliggör styrning och övervakning av simuleringen i realtid, medan CAN-modulen gör det möjligt att koda och skicka simulerade signaler enligt DBC-definitioner till en testmiljö.

Arbetet visar att simulerade fordonsbeteenden kan överföras via CAN och observeras i en fysisk testmiljö. Därmed visar prototypen potential att stödja tidigare verifiering av fordonsmjukvara i ett shift-left-perspektiv. Samtidigt är verktyget fortfarande ett principbevis, där vidareutveckling krävs för mer realistisk fordonsdynamik, fler signaler och mer omfattande systemintegration.

Nyckelord: Fordonsdynamik, simulering, CAN, J1939, DBC, HIL, shift-left testing, testrigg, fordonsmjukvara, Volvo Bussar.

Simulation-based verification of vehicle behavior
A CAN-based tool for early testing in a laboratory environment
Simon Carlén & Alvin Stallgård
Institutionen för data- och informationsteknik
Chalmers Tekniska Högskola

Abstract

The automotive industry is undergoing a transition toward shorter development cycles, increased software complexity and higher demands on verification and validation. As a result, simulation-based testing has become increasingly important. One central approach in this development is shift-left testing, where verification activities are moved to earlier stages of the development process. This can reduce the dependency on physical test vehicles and enable more flexible, repeatable and cost-efficient testing.

The purpose of this bachelor's thesis is to develop a CAN-based simulation tool for early verification of vehicle behavior in a laboratory environment. The work has been carried out in collaboration with Volvo Buses and focuses on investigating how a software-based vehicle model can be used to generate simulated vehicle and sensor signals for an existing test environment.

The result is a functional prototype consisting of three main components: a vehicle model, a graphical user interface and a CAN communication module. The vehicle model simulates central states such as velocity, acceleration, orientation, gear selection, engine speed and wheel-related variables. The graphical user interface enables real-time control and monitoring of the simulation, while the CAN module allows simulated signals to be encoded according to DBC definitions and transmitted to a test environment.

The thesis demonstrates that simulated vehicle behavior can be transmitted via CAN and observed in a physical test environment. The prototype therefore shows potential to support earlier verification of vehicle software from a shift-left testing perspective. However, the tool should be considered a proof of concept, as further development is required to improve vehicle dynamics realism, increase the number of implemented signals and enable more comprehensive system integration.

Keywords: Vehicle dynamics, simulation, CAN, J1939, DBC, HIL, shift-left testing, test rig, vehicle software, Volvo Buses.

Förord

Detta examensarbete på kandidatnivå har genomförts vid Institutionen för data- och informationsteknik på Chalmers Tekniska Högskola under våren 2026. Arbetet har utförts i samarbete med Volvo Bussar i Göteborg och utgör det avslutande momentet i högskoleprogrammet i Mekatronik.

Syftet med arbetet har varit att undersöka hur simuleringsbaserade metoder kan användas för att möjliggöra tidig verifiering av fordonsbeteenden i en industriell testmiljö. Genom att kombinera akademiska kunskaper med praktiska tillämpningar i Volvo Bussars testmiljöer har arbetet gett värdefull erfarenhet av modern fordonsutveckling och verifieringsprocesser.

Ett särskilt tack riktas till vår handledare på företaget, Andreas Nilsson, samt till Robin Kaveh, Aria Lind och Helena Bentzel för värdefull vägledning, teknisk expertis och kontinuerligt stöd under arbetets gång.

Vidare vill vi tacka vår akademiska handledare vid Chalmers Tekniska Högskola, Jonas Duregård, för konstruktiv feedback, vägledning och akademisk kvalitetssäkring genom projektets olika faser.

Simon Carlén och Alvin Stallgård
Göteborg, 2026

Akronymer

6 DOF	Six Degrees of Freedom, sex frihetsgrader
CAN	Controller Area Network
DBC	CAN Database, databasfil för CAN-meddelanden och CAN-signaler
DCM	Direction Cosine Matrix, rotationsmatris för koordinattransformation
ECU	Electronic Control Unit, elektronisk styrenhet
GUI	Graphical User Interface, grafiskt användargränssnitt
HIL	Hardware-in-the-Loop
J1939	SAE J1939, standard för CAN-baserad kommunikation i tunga fordon
MVP	Minimum Viable Product
PI	Proportional Integral, proportionell integrerande reglering
PoC	Proof of Concept, principbevis
RPM	Revolutions Per Minute, varv per minut
SAE	Society of Automotive Engineers
SIL	Software-in-the-Loop
USB	Universal Serial Bus
CAPL	Communication Access Programming Language

Symboler

Nedan listas de symboler, parametrar och variabler som används i rapportens modeller, beräkningar och implementation.

Index och referensramar

$(\cdot)_B$	Storhet uttryckt i kroppsfast referensram, det vill säga fordonets lokala koordinatsystem.
$(\cdot)_I$	Storhet uttryckt i global referensram, det vill säga simuleringsmiljöns fasta koordinatsystem.
$(\cdot)_L$	Storhet som avser vänster sida av fordonet.
$(\cdot)_R$	Storhet som avser höger sida av fordonet.
k	Diskret tidsindex eller växelindex, beroende på sammanhang.

Fordonsparametrar och konstanter

m	Fordonets massa [kg].
g	Tyngdacceleration [m/s^2].
ρ	Luftdensitet [kg/m^3].
C_d	Luftmotståndskoefficient [–].
A	Fordonets frontarea [m^2].
L	Axelavstånd mellan framaxel och bakaxel [m].
r_{hjul}	Effektiv hjulradie [m].
C_{rr}	Rullmotståndskoefficient [–].
μ_{long}	Longitudinell friktionskoefficient mellan hjul och väg [–].
α_{grade}	Konstant väglutning som används i fordonsmodellen [rad].
ϕ_g	Markplanets rollvinkel [rad].
θ_g	Markplanets pitchvinkel [rad].

Tid, insignaler och styrvariabler

t	Tid [s].
Δt	Diskret tidssteg i simuleringen [s].

u_{th}	Gasinsignal, normaliserad till intervallet $[0, 1]$ [–].
u_{br}	Bromsinsignal, normaliserad till intervallet $[0, 1]$ [–].
u	Generell styrsignal, exempelvis regulatorns utsignal [–].
δ	Styrvinkel [rad].
v_{ref}	Referenshastighet, exempelvis målhastighet för farthållare [m/s].
e	Reglerfel, exempelvis skillnaden mellan referenshastighet och aktuell hastighet [–].
I	Integralled i PI regulatorn [–].
K_p	Proportionell förstärkning i PI regulatorn [–].
K_i	Integrerande förstärkning i PI regulatorn [–].

Tillstånd och 6 DOF kinematik

$\mathbf{p}_I = [x\ y\ z]^T$	Fordonets position i global referensram [m].
x, y, z	Position längs global referensrams tre axlar [m].
$\boldsymbol{\eta} = [\phi\ \theta\ \psi]^T$	Fordonets orientering uttryckt med Euler-vinklar [rad].
ϕ	Rollvinkel, rotation kring fordonets längdaxel [rad].
θ	Pitchvinkel, rotation kring fordonets tväraxel [rad].
ψ	Yawvinkel, rotation kring vertikal axel [rad].
$\mathbf{v}_B = [v_x\ v_y\ v_z]^T$	Linjär hastighet i kroppsfast referensram [m/s].
v_x	Longitudinell hastighet i fordonets färdriktning [m/s].
v_y	Lateral hastighet i kroppsfast referensram [m/s].
v_z	Vertikal hastighet i kroppsfast referensram [m/s].
v	Longitudinell hastighet, vanligtvis $v = v_x$ [m/s].
$\boldsymbol{\omega}_B = [\omega_x\ \omega_y\ \omega_z]^T$	Vinkelhastighet i kroppsfast referensram [rad/s].
$\mathbf{a}_B = [a_x\ a_y\ a_z]^T$	Linjär acceleration i kroppsfast referensram [m/s^2].
a_x	Longitudinell acceleration [m/s^2].
$\boldsymbol{\alpha}_B$	Vinkelacceleration i kroppsfast referensram [rad/s^2].
$\dot{\psi}_{\text{ref}}$	Referensvärde för yawhastighet från styrmodell [rad/s].

Transformationer och rotationsmatriser

R_{BI}	Rotationsmatris som transformerar en vektor från global referensram till kroppsfast referensram [–].

R_{IB}	Rotationsmatris som transformerar en vektor från kroppsfast referensram till global referensram [–].
$T(\phi, \theta)$	Transformationsmatris mellan kroppsfixa vinkelhastigheter och Euler-vinkelderivator [–].
$\dot{\boldsymbol{\eta}}$	Tidsderivata av Euler-vinklarna [rad/s].
$\dot{\mathbf{p}}_I$	Tidsderivata av position i global referensram [m/s].

Drivlina, hjul och varvtal

T_{eng}	Motorvridmoment [N m].
$T_{\text{eng,max}}$	Maximalt motorvridmoment [N m].
n_{eng}	Motorvarvtal [rpm].
$n_{\text{eng,max}}$	Maximalt motorvarvtal [rpm].
$i_{\text{gb},k}$	Utväxling i växellådan för växel k [–].
$i_{\text{gb,rev}}$	Utväxling för backväxel [–].
i_{fd}	Slutväxelutväxling [–].
i_{axle}	Axelutväxling [–].
η_{gb}	Verkningsgrad i växellådan [–].
η_{shaft}	Verkningsgrad i drivaxeln [–].
η_{axle}	Verkningsgrad i axeln [–].
λ_L	Andel av drivmomentet som fördelas till vänster hjul [–].
$T_{\text{hjul},L}$	Drivmoment på vänster hjul [N m].
$T_{\text{hjul},R}$	Drivmoment på höger hjul [N m].
$v_{\text{hjul},L}$	Markhastighet för vänster hjul [m/s].
$v_{\text{hjul},R}$	Markhastighet för höger hjul [m/s].
$n_{\text{hjul},L}$	Hjulvarvtal för vänster hjul [rpm].
$n_{\text{hjul},R}$	Hjulvarvtal för höger hjul [rpm].
$n_{\text{hjul,max}}$	Maximalt tillåtet hjulvarvtal [rpm].

Krafter och moment

$\sum F$	Summan av verkande krafter [N].
$\sum M_B$	Summan av verkande moment i kroppsfast referensram [N m].
F_{driv}	Total drivkraft från drivlinan [N].
F_{broms}	Total bromskraft applicerad på hjulen [N].

$F_{\text{broms,max}}$	Maximal bromskraft [N].
F_{drag}	Luftmotståndskraft [N].
F_{rull}	Rullmotståndskraft [N].
F_{grade}	Kraftkomponent från konstant väglutning [N].
F_{pitch}	Kraftkomponent från markplanets pitchvinkel [N].
F_{resist}	Samlad motståndskraft från luftmotstånd, rullmotstånd och lutning [N].
F_{net}	Resulterande longitudinell kraft som används för att beräkna acceleration [N].
T_{load}	Lastmoment som reflekteras från hjulen till drivlinan [N m].
I_{hjul}	Hjulens tröghetsmoment [kg m^2].
I_B	Fordonets tröghetsmatris uttryckt i kroppsfast referensram [kg m^2].

Diskreta uppdateringar och derivator

$\dot{x}$	Tidsderivata av tillståndet x [varierar beroende på storhet].
$x(k)$	Tillstånd vid diskret tidssteg k [varierar beroende på storhet].
$x(k+1)$	Tillstånd vid nästa diskreta tidssteg [varierar beroende på storhet].
$\dot{v}$	Tidsderivata av hastighet, det vill säga acceleration [m/s^2].
$\dot{\omega}$	Tidsderivata av vinkelhastighet [rad/s^2].

Innehåll

Figurer

Tabeller

1

Inledning

Fordonsindustrin befinner sig i en omställning där ökade krav på kortare utvecklingscykler, högre kvalitet och lägre kostnader driver ett allt större fokus på digitalisering och simulering. Ett centralt arbetssätt i denna utveckling är *shift-left testing*, vilket innebär att verifiering och validering av system flyttas till tidigare faser i utvecklingsprocessen. Genom att tidigt använda simuleringar och virtuella testmiljöer kan potentiella problem identifieras innan fysiska prototyper byggs, vilket kan minska både utvecklingsrisker och resursåtgång.

Inom fordonsutveckling används testfordon och testriggar traditionellt för att verifiera funktioner och systembeteenden under realistiska körförhållanden. Dessa tester är ofta tidskrävande och kostsamma, eftersom de kan kräva manuella inställningar, omfattande testkörningar och tillgång till fysisk hårdvara. Detta skapar ett behov av metoder som gör det möjligt att tidigare i utvecklingsprocessen testa och verifiera fordonsfunktioner under kontrollerade och repeterbara förhållanden.

1.1 Bakgrund

Volvo Bussar har identifierat simulering och virtuell validering som ett strategiskt område för att effektivisera sin utvecklingsprocess. Genom att använda simuleringsbaserade testmetoder kan beroendet av fysiska testfordon minska, samtidigt som verifiering och validering kan genomföras tidigare i utvecklingsarbetet. Detta gör det relevant att undersöka hur digitala modeller kan samverka med befintliga testmiljöer och i vilken utsträckning simulerade signaler kan användas som ersättning eller komplement till signaler från ett fysiskt fordon.

Trots tillgång till avancerade testmiljöer finns i dagsläget begränsade möjligheter att på ett effektivt och flexibelt sätt simulera realistiska körscenarier. Flertalet tester är fortfarande beroende av manuella inställningar, statiska insignaler eller fysisk hårdvara, vilket försvårar ett konsekvent arbetssätt i linje med ett *shift-left*-perspektiv.

Befintliga verktyg och testmiljöer förutsätter därmed i stor utsträckning fysisk testning, vilket kan leda till längre utvecklingscykler, begränsad flexibilitet vid iterationer och ökade kostnader i utvecklingsprocessen. Bakgrunden till detta examensarbete är därför behovet av att undersöka hur simuleringsbaserade metoder kan användas tillsammans med befintliga testriggar för att stödja tidig och effektiv verifiering av fordonsmjukvara i en industriell kontext.

1.2 Problemformulering

För att möjliggöra tidigare verifiering av fordonsmjukvara behövs ett verktyg som kan simulera relevanta fordonsbeteenden och översätta dessa till signaler som kan användas i en befintlig testmiljö. Utmaningen ligger inte enbart i att skapa en fordonsmodell, utan även i att koppla modellens tillstånd till CAN-baserad kommunikation så att simulerade fordons- och sensorsignaler kan skickas till en fysisk testrigg.

Ett sådant verktyg behöver därför kombinera fordonsdynamisk modellering, realtidsstyrning, grafisk visualisering och signalöverföring via CAN. Samtidigt behöver lösningen vara tillräckligt flexibel för att kunna vidareutvecklas med fler signaler, fler körscenarier och mer avancerade funktioner.

1.3 Syfte

Projektets syfte är att utveckla ett CAN-baserat simuleringsverktyg som möjliggör tidig verifiering av fordonsbeteenden i labbmiljö. Verktyget ska kunna simulera relevanta körscenarier, generera fordonsrelaterade signaler och överföra dessa till en befintlig testmiljö för att minska beroendet av fysiska tester.

1.4 Mål

Projektets mål är att utveckla ett simuleringsbaserat mjukvaruverktyg som kan användas för tidig och reproducerbar verifiering av fordonsbeteenden i labbmiljö. Arbetet syftar till att ta fram ett principbevis, *proof of concept*, som demonstrerar hur ett virtuellt fordon kan simuleras och användas för att testa fordonsmjukvara utan behov av ett fysiskt fordon.

Det förväntade resultatet är en fungerande prototyp som kan simulera centrala fordonsrörelser såsom acceleration, inbromsning, styrning och hastighetsreglering i realtid. Simuleringen ska kunna styras manuellt genom ett grafiskt användargränssnitt och generera tillstånd som kan användas för vidare signalhantering.

Vidare ska simulerade fordons- och sensorsignaler kunna genereras, kodas och överföras via ett CAN-gränssnitt till en befintlig testmiljö. På så sätt ska verktyget demonstrera hur simulerade körscenarier kan användas som underlag för verifiering av fordonsmjukvara i ett *shift-left*-perspektiv.

1.5 Frågeställningar

Projektet ska besvara följande huvudfråga:

> Hur kan ett CAN-baserat simuleringsverktyg utvecklas och utvärderas för att möjliggöra tidig verifiering av fordonsbeteenden i en labb-baserad testmiljö?

Huvudfrågan konkretiseras genom följande delfrågor:

1. Hur kan en förenklad och utbyggbar fordonsmodell utformas för att simulera centrala körbeteenden i realtid?
2. Hur kan simulerade fordonsrelaterade signaler kopplas till CAN-meddelanden och överföras till en befintlig HIL-miljö?
3. Vilka möjligheter och begränsningar uppstår när ett simuleringsbaserat verktyg används för tidigare verifiering av fordonsmjukvara?

1.6 Avgränsningar

Projektet avgränsas till utveckling av ett simuleringsbaserat mjukvaruverktyg avsett för testning och verifiering av fordonsmjukvara i labbmiljö. Arbetet fokuserar på funktionalitet, struktur och integration snarare än maximal precision.

Följande avgränsningar tillämpas i projektet:

- Simuleringen begränsas till ett förenklat fordonsdynamiskt modellantagande, där syftet är att återskapa realistiskt beteende på systemnivå snarare än exakt fysisk representation.
- Endast ett urval av körmanövrer behandlas, såsom acceleration, inbromsning, konstant hastighet och svängar. Avancerade manövrer och extrema körfall, exempelvis kontrollerad sladdning, ingår inte.
- Arbetet omfattar inte modellbaserad parameteridentifiering eller kalibrering mot mätdata från fysiska fordon.
- Det grafiska användargränssnittet utvecklas för att stödja konfiguration, styrning och övervakning av simuleringen, men omfattar inte användartester eller vidare utvärdering av användarupplevelse.
- CAN-kommunikationen begränsas till ett definierat urval av fordonssignaler relevanta för verifiering av körbeteende.
- Integration mot testrigg avser funktionell verifiering i kontrollerad labbmiljö och omfattar inte långtidstester, prestandavalidering eller certifiering.

2

Teknisk bakgrund

Detta kapitel presenterar den tekniska bakgrund som ligger till grund för projektet. Kapitlet behandlar utvecklings- och testmetodik inom fordonsutveckling, grundläggande fordonsdynamik, realtidssimulering, reglering samt CAN-baserad kommunikation. Avslutningsvis presenteras de programvaruverktyg och bibliotek som varit relevanta för utvecklingen av simuleringsverktyget.

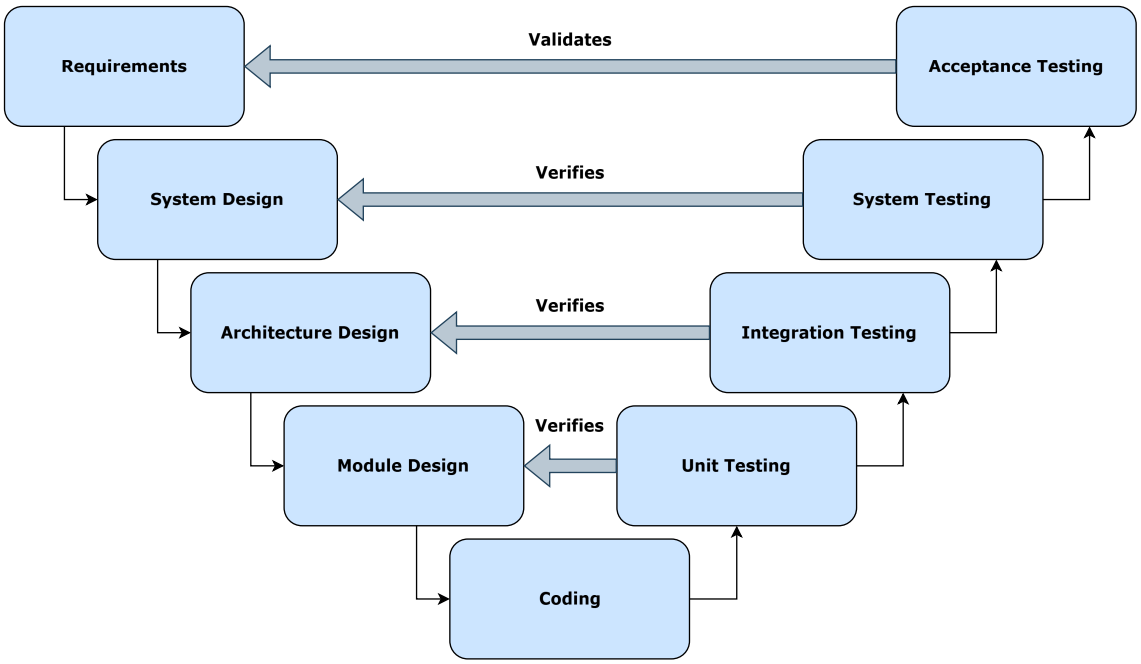
Syftet med kapitlet är att ge den teoretiska och tekniska kontext som krävs för att förstå projektets modellering, implementation och integration mot testmiljö. För symboler, variabler och enheter hänvisas även till symbolförteckningen i rapportens inledande del.

2.1 Utveckling och testmetodik

Modern fordonsutveckling präglas av ökande systemkomplexitet, kortare utvecklingscykler och ett växande beroende av mjukvara. Detta ställer höga krav på strukturerade utvecklingsprocesser och effektiva verifieringsmetoder. I detta avsnitt beskrivs V-modellen och *Shift-left testing*, vilka utgör viktiga utgångspunkter för projektets mål att möjliggöra tidigare verifiering av fordonsbeteenden i labbmiljö.

2.1.1 V-modellen

V-modellen är en utvecklingsmodell som ofta används inom system- och mjukvaruutveckling, särskilt i säkerhetskritiska och tekniskt komplexa system. Modellen beskriver hur utvecklingsarbetet bryts ned från övergripande krav till implementation, samt hur dessa nivåer sedan verifieras och valideras genom motsvarande testnivåer. Figur 2.1 visar V-modellens principiella struktur.



Figur 2.1: V-modellen med utvecklingsfaser till vänster och verifierings- och valideringsfaser till höger.

Som framgår av Figur 2.1 representerar den vänstra sidan av modellen en successiv nedbrytning av systemet. Arbetet börjar med övergripande krav och går därefter vidare till systemkrav, systemarkitektur, detaljdesign och slutligen implementation. Den högra sidan av modellen beskriver hur systemet därefter verifieras och valideras på motsvarande nivåer. Detaljerad implementation verifieras genom enhetstester, komponenter och delsystem genom integrationstester, och det färdiga systemet genom systemtester och validering mot de ursprungliga kraven.

En central princip i V-modellen är att varje utvecklingsnivå har en motsvarande testnivå. På detta sätt skapas spårbarhet mellan krav, designbeslut och verifiering. Detta är särskilt viktigt vid utveckling av fordonsmjukvara, där flera tekniska delsystem samverkar och där fel kan vara kostsamma att upptäcka sent i utvecklingsprocessen [1].

I projektets kontext är V-modellen relevant eftersom simuleringsverktyget syftar till att stödja tidigare verifiering av fordonsbeteenden. Genom att använda en simulerad fordonsmodell och generera CAN-signaler kan vissa tester tidigareläggas från senare fysiska testfaser till tidigare, labbaserade testmiljöer. Verktyget kan därmed ses som ett sätt att stödja ett mer *shift-left*-orienterat arbetssätt, där verifiering sker tidigare i utvecklingskedjan.

2.1.2 Shift-left testing

Shift-left testing innebär att verifiering och validering flyttas till tidigare faser i utvecklingsprocessen. I stället för att huvuddelen av testningen sker sent, exempelvis först när ett komplett fordon eller en färdig testrigg finns tillgänglig, används simuleringar, virtuella miljöer och automatiserade tester för att identifiera fel i ett tidigare

skede [2], [3].

Syftet med *Shift-left testing* är att minska utvecklingsrisker och kostnader genom att problem upptäcks innan de förs vidare i utvecklingskedjan. Tidig testning gör det möjligt att identifiera brister innan de blir mer komplexa att analysera och mer resurskrävande att åtgärda [2]. Detta är särskilt relevant inom fordonsindustrin, där sena fel ofta kan kräva omtestning, tillgång till fysisk hårdvara eller omfattande systemintegration.

2.2 Fordonsdynamik

Fordonsdynamik beskriver hur fordon rör sig under påverkan av krafter och moment. Området omfattar bland annat hastighet, acceleration, orientering, hjulrörelser och samspelet mellan drivlina, däck och vägunderlag.

Rörelsen kan beskrivas med mekaniska modeller där krafter och moment analyseras i relation till fordonets massa, geometri och aktuella körförhållanden. Beroende på modellens syfte kan olika förenklingsnivåer användas, från enkla longitudinella modeller till mer detaljerade modeller som även beskriver sidodynamik, vertikalrörelse och rotationer kring fordonets axlar.

2.2.1 Grundläggande rörelseekvationer

Fordonets translationsrörelse kan beskrivas med Newtons andra lag [4],

$$\sum \mathbf{F} = m\mathbf{a}, \tag{2.1}$$

där m är fordonets massa, $\mathbf{a}$ är acceleration och $\sum \mathbf{F}$ är summan av de yttre krafter som verkar på fordonet.

I projektets modell används framför allt den longitudinella riktningen, det vill säga fordonets färdriktning. Kraftbalansen kan då förenklas till

$$ma_x = F_{\text{net}}, \tag{2.2}$$

där a_x är fordonets longitudinella acceleration och F_{net} är den resulterande kraften i färdriktningen.

För rotationsrörelser kan en styv kropps dynamik generellt beskrivas med Newton-Eulers ekvationer [4],

$$\sum \mathbf{M}_B = \mathbf{I}_B \dot{\boldsymbol{\omega}}_B + \boldsymbol{\omega}_B \times (\mathbf{I}_B \boldsymbol{\omega}_B), \tag{2.3}$$

där $\sum \mathbf{M}_B$ är summan av moment i kroppsfast referensram, $\mathbf{I}_B$ är tröghetsmatrisen och $\boldsymbol{\omega}_B$ är vinkelhastigheten i kroppsfast referensram. I projektets modell används denna formulering som teoretisk grund, men rotationsuppdateringen hanteras huvudsakligen kinematiskt.

2.2.2 Longitudinell fordonsdynamik

Longitudinell fordonsdynamik beskriver fordonets rörelse i färdriktningen. Den omfattar bland annat drivkraft, bromskraft, luftmotstånd, rullmotstånd och krafter från väglutning. I projektets modell antas den longitudinella hastigheten vara den dominerande hastighetskomponenten, vilket innebär att

$$v \approx v_x. \tag{2.4}$$

Den longitudinella kraftbalansen kan skrivas som [4]

$$m\dot{v} = F_{\text{driv}} + F_{\text{broms}} + F_{\text{resist}}, \tag{2.5}$$

där F_{driv} är drivkraften från drivlinan, F_{broms} är bromskraften och F_{resist} är den samlade motståndskraften. Teckenhantering är viktig eftersom motstånds- och bromskrafter ska verka i motsatt riktning mot fordonets rörelse.

Drivkraft

Drivkraften uppstår genom att motorns vridmoment överförs via växellåda, drivaxel och hjul till vägplanet. En idealiserad relation mellan hjulmoment och drivkraft kan skrivas som

$$F_{\text{driv}} \approx \frac{T_{\text{hjul},L} + T_{\text{hjul},R}}{r_{\text{hjul}}}, \tag{2.6}$$

där $T_{\text{hjul},L}$ och $T_{\text{hjul},R}$ är drivmoment på vänster respektive höger hjul, och r_{hjul} är hjulradien.

Bromskraft

Bromskraften modelleras som en extern kraft som verkar på hjulen och motverkar fordonets rörelse. Med en normaliserad bromsinsignal $u_{\text{br}} \in [0, 1]$ kan bromskraften per hjul beskrivas som

$$F_{\text{broms,each}} = u_{\text{br}} F_{\text{broms,max,each}}, \tag{2.7}$$

där $F_{\text{broms,max,each}}$ är maximal bromskraft per hjul. Den totala bromskraften för två drivande hjul kan då skrivas som

$$F_{\text{broms}} = 2F_{\text{broms,each}} s_{\text{br}}, \tag{2.8}$$

där s_{br} anger bromskraftens riktning så att den motverkar aktuell eller avsedd rörelse.

Motståndskrafter

De huvudsakliga motståndskrafterna i modellen är luftmotstånd, rullmotstånd och lutningskrafter. Dessa krafter används för att beskriva hur externa motstånd påverkar fordonets acceleration i färdriktningen [4]. Luftmotståndets storlek kan beskrivas som

$$|F_{\text{drag}}| = \frac{1}{2}\rho C_d A v^2, \tag{2.9}$$

där ρ är luftdensitet, C_d är luftmotståndskoefficient och A är fordonets frontarea.

Rullmotståndet kan approximeras som

$$|F_{\text{rull}}| = C_{rr} mg \cos(\alpha_{\text{grade}}), \tag{2.10}$$

där C_{rr} är rullmotståndskoefficient och α_{grade} är väglutningen.

Kraftkomponenten från väglutning ges av

$$F_{\text{grade}} = -mg \sin(\alpha_{\text{grade}}). \tag{2.11}$$

En positiv lutning i färdriktningen ger därmed en negativ kraft, vilket motsvarar att fordonet kör uppför.

2.2.3 Styrning och yaw-rörelse

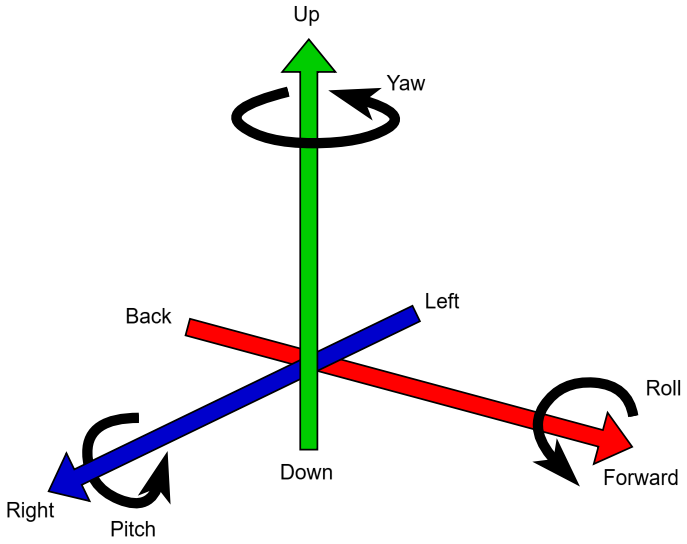
För att beskriva fordonets rotationsrörelse kring vertikalaxeln används en förenklad styrmodell. Med framåthastighet v, styrvinkel δ och axelavstånd L kan en referens för yawhastighet beräknas enligt

$$\dot{\psi}_{\text{ref}} = \frac{v}{L} \tan(\delta). \tag{2.12}$$

Detta samband bygger på en förenklad cykelmodell där fordonets rörelse approximeras genom ett styrande hjulpar och ett bakre hjulpar. Modellen är inte avsedd att beskriva avancerad lateral dynamik, men ger en tillräcklig approximation för att simulera svängrörelser i det aktuella projektet.

2.2.4 Sex frihetsgrader

Ett fordons rörelse i rummet kan beskrivas med sex frihetsgrader, vilket omfattar translation längs tre axlar samt rotation kring tre axlar [5]. Figur 2.2 illustrerar de sex frihetsgraderna samt de kroppsfixa axlar som används i modellen.



Figur 2.2: Egen illustration av fordonets sex frihetsgrader. Translation sker längs axlarna x, y och z, medan rotation sker kring samma axlar i form av roll ϕ, pitch θ och yaw ψ.

I modellen beskrivs fordonets position i global referensram som

$$\mathbf{p}_I = [x \ y \ z]^T, \tag{2.13}$$

och orienteringen med Euler-vinklar som

$$\boldsymbol{\eta} = [\phi \ \theta \ \psi]^T, \tag{2.14}$$

där ϕ är roll, θ är pitch och ψ är yaw.

Den linjära hastigheten i kroppsfast referensram skrivs som

$$\mathbf{v}_B = [v_x \ v_y \ v_z]^T, \tag{2.15}$$

och vinkelhastigheten som

$$\boldsymbol{\omega}_B = [\omega_x \ \omega_y \ \omega_z]^T. \tag{2.16}$$

2.2.5 Referensramar och koordinattransformation

Två referensramar används i modellen. Den globala referensramen $\{I\}$ beskriver fordonets position i simuleringsmiljön, medan den kroppsfixa referensramen $\{B\}$ följer fordonets orientering. Hastigheter, accelerationer och krafter uttrycks ofta i kroppsfast referensram, medan positionen uttrycks i global referensram. Denna uppdelning mellan referensramar är central vid modellering av rörelse med sex frihetsgrader [5].

Transformationen mellan referensramarna kan beskrivas med en rotationsmatris. Om $\mathbf{R}_{BI}$ transformerar en vektor från global referensram till kroppsfast referensram gäller

$$\mathbf{x}_B = \mathbf{R}_{BI}\mathbf{x}_I. \tag{2.17}$$

Den inversa transformationen ges av

$$\mathbf{R}_{IB} = \mathbf{R}_{BI}^T. \tag{2.18}$$

Positionens kinematik kan därmed skrivas som

$$\dot{\mathbf{p}}_I = \mathbf{v}_I, \qquad \mathbf{v}_I = \mathbf{R}_{IB}(\boldsymbol{\eta})\mathbf{v}_B. \tag{2.19}$$

I diskret tid kan hastighet och position uppdateras med explicit Euler-integration,

$$\mathbf{v}_B(k+1) = \mathbf{v}_B(k) + \mathbf{a}_B(k)\Delta t, \tag{2.20}$$

$$\mathbf{p}_I(k+1) = \mathbf{p}_I(k) + \mathbf{R}_{IB}(\boldsymbol{\eta}(k))\mathbf{v}_B(k+1)\Delta t. \tag{2.21}$$

2.2.6 Euler-vinklar och rotationsuppdatering

Sambandet mellan Euler-vinklarnas derivator och vinkelhastigheten i kroppsfast referensram kan uttryckas som [5]

$$\dot{\boldsymbol{\eta}} = \mathbf{T}(\phi, \theta)\boldsymbol{\omega}_B, \tag{2.22}$$

där $\mathbf{T}(\phi, \theta)$ är transformationsmatrisen mellan kroppsfixa vinkelhastigheter och Euler-vinkelderivator.

I diskret tid kan vinkelhastigheten uppdateras enligt

$$\boldsymbol{\omega}_B(k+1) = \boldsymbol{\omega}_B(k) + \boldsymbol{\alpha}_B(k)\Delta t, \tag{2.23}$$

och orienteringen enligt

$$\boldsymbol{\eta}(k+1) = \boldsymbol{\eta}(k) + \dot{\boldsymbol{\eta}}(k+1)\Delta t. \tag{2.24}$$

Euler-vinklar är intuitiva och användbara för visualisering, men har en singularitet vid $\theta \approx \pm\pi/2$. I praktiska implementationer behöver därför numeriska skydd användas för att undvika instabilitet nära dessa lägen [5].

2.3 Reglersystemdesign och realtidssimulering

Reglering används för att påverka ett systems beteende baserat på återkoppling från dess aktuella tillstånd. I detta projekt används reglering framför allt för hastighetsreglering, exempelvis i form av farthållare. Eftersom simuleringen körs med diskreta tidssteg är modellen naturligt kopplad till tidsdiskret reglering [6].

2.3.1 Diskret tidsmodellering

Ett kontinuerligt system kan generellt beskrivas som

$$\dot{x} = f(x, u), \tag{2.25}$$

där x är systemets tillstånd och u är styrsignalen. I en simulering approximeras detta i diskret tid, exempelvis med explicit Euler-metod,

$$x(k+1) \approx x(k) + \dot{x}(k)\Delta t. \tag{2.26}$$

Valet av tidssteg Δt påverkar både noggrannhet och stabilitet. Ett för stort tidssteg kan ge orealistiska förändringar mellan uppdateringar, medan ett mycket litet tidssteg ökar beräkningsbehovet. Vid tidsdiskret modellering behöver därför tidssteget väljas med hänsyn till både systemets dynamik och simuleringens beräkningskrav [6].

2.3.2 Återkopplade system

Ett återkopplat system använder information om systemets aktuella tillstånd för att beräkna en styrsignal. Reglerfelet definieras vanligtvis som

$$e = r - y, \tag{2.27}$$

där r är referensvärdet och y är det uppmätta eller beräknade systemvärdet. Vid hastighetsreglering kan referensen vara en önskad hastighet v_{ref}, medan systemvärdet är nuvarande hastighet v [6].

2.3.3 PI-reglering

En PI-regulator består av en proportionell och en integrerande del. Den proportionella delen reagerar på det aktuella reglerfelet, medan den integrerande delen tar hänsyn till det ackumulerade felet över tid. I kontinuerlig tid kan regulatorn skrivas som [6]

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau, \tag{2.28}$$

där K_p är proportionell förstärkning och K_i är integrerande förstärkning.

I diskret tid kan integralledet approximeras som

$$e(k) = r(k) - y(k), \tag{2.29}$$
$$I(k) = I(k-1) + e(k)\Delta t, \tag{2.30}$$
$$u(k) = K_p e(k) + K_i I(k). \tag{2.31}$$

För fysiska system behöver styrsignalen ofta begränsas till ett tillåtet intervall. I en fordonsmodell kan detta exempelvis motsvara att gaspådraget begränsas till intervallet $[0, 1]$. Sådana begränsningar är viktiga eftersom regulatorns beräknade styrsignal annars kan anta värden som inte är möjliga att realisera i det fysiska systemet.

2.4 CAN-baserad kommunikation

I detta avsnitt behandlas den CAN-baserade kommunikation som är relevant för projektet. Först beskrivs den grundläggande uppbyggnaden av Controller Area Network, CAN, därefter behandlas SAE J1939, DBC-filer och de gränssnitt som används för att koppla en dator till en CAN-buss.

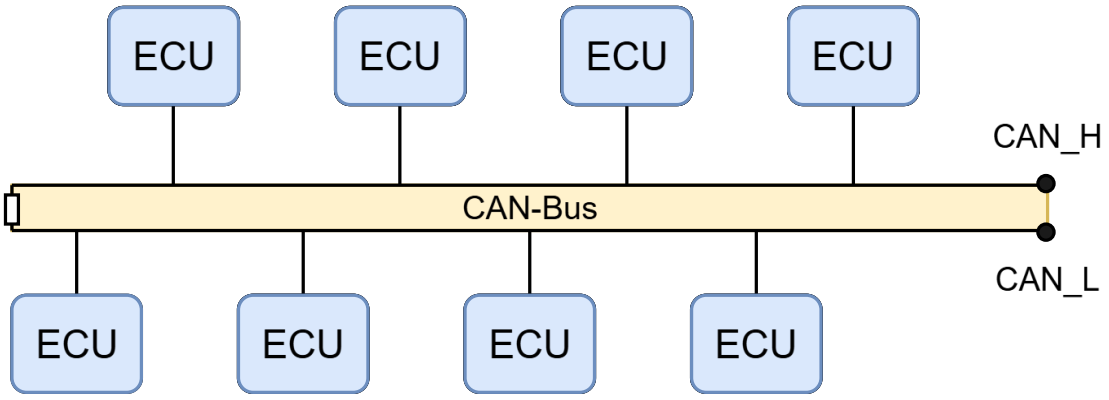
2.4.1 Controller Area Network

Controller Area Network, CAN, är ett kommunikationsnätverk där data överförs i form av meddelanden. Varje meddelande består av en identifierare och ett antal databytes. Identifieraren anger både meddelandets prioritet och dess innebörd, medan databytes innehåller de signaler som ska överföras [7].

En viktig egenskap hos CAN är att flera noder kan vara anslutna till samma databuss. När ett meddelande skickas på databussen kan samtliga noder läsa det, men varje nod avgör själv om meddelandet är relevant att behandla. Detta gör CAN särskilt lämpligt i fordonssystem, där flera styrenheter behöver kommunicera med varandra på ett tillförlitligt sätt [7].

Den fysiska kommunikationen sker via en databuss, vanligtvis kallad CAN-buss. CAN-bussen består av två ledare, CAN_H och CAN_L, där signalöverföringen är differentiell. Det innebär att mottagaren tolkar spänningsskillnaden mellan CAN_H och CAN_L i stället för att mäta signalerna mot jord. Eftersom störningar ofta påverkar båda ledarna på liknande sätt kan störningarna till stor del undertryckas vid den differentiella avläsningen. På så sätt blir kommunikationen mer robust mot exempelvis elektromagnetisk interferens, brus och signalförluster [7].

För att kommunikationen på CAN-bussen ska fungera korrekt måste samtliga noder använda samma bitrate, det vill säga samma överföringshastighet. Bitrate anger hur många bitar som överförs per sekund. Om noderna använder olika bitrate kan meddelandena inte tolkas korrekt, vilket leder till kommunikationsfel. Figur 2.3 visar en principiell illustration av CAN-kommunikation.

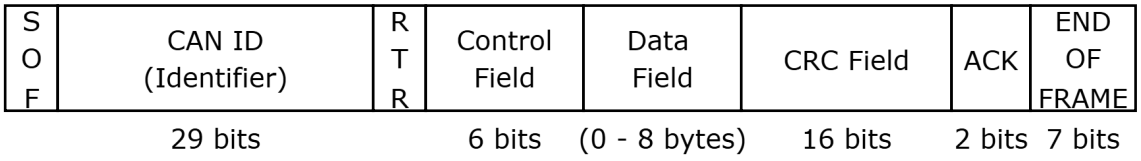


Figur 2.3: Principiell illustration av CAN-kommunikation mellan flera noder på samma databuss.

2.4.2 SAE J1939

SAE J1939 är en högre lager-kommunikationsstandard som bygger på CAN och används i stor utsträckning inom tunga fordon, exempelvis bussar, lastbilar och arbetsmaskiner. Standarden definierar hur fordonsrelaterad information ska struktureras, adresseras och överföras mellan elektroniska styrenheter, Electronic Control Units, på ett gemensamt CAN-nätverk. Exempel på information som kan överföras är fordonshastighet, motorvarvtal, temperaturer, diagnostiska data och andra signaler från drivlina och chassi.

CAN stöder två identifierarformat: standardformat med 11-bitars identifierare och extended-format med 29-bitars identifierare [7]. SAE J1939 använder extended-formatet, vilket innebär att den 29-bitars CAN-identifieraren får en standardiserad struktur. Identifieraren delas upp i flera fält, bland annat prioritet, reserved bit, data page, PDU-format, PDU specific och source address [8]. Den principiella uppbyggnaden visas i Figur 2.4.



Figur 2.4: Principiell uppbyggnad av en J1939-baserad CAN-frame med 29-bitars extended identifierare och databytes [8].

Den 29-bitars identifieraren används inte enbart för CAN-arbitrering, utan innehåller även information om meddelandets typ, prioritet och avsändande nod. Fältet *priority* omfattar tre bitar och används vid databussarbitrering, där ett lägre numeriskt värde motsvarar högre prioritet. Fälten *reserved bit*, *data page*, *PDU format* och *PDU specific* används tillsammans för att bilda ett Parameter Group Number, PGN, vilket anger vilken typ av information meddelandet innehåller. Fältet *source address* omfattar åtta bitar och anger vilken styrenhet som skickat meddelandet [8].

Tabell 2.1: Struktur för den 29-bitars identifieraren i SAE J1939.

Fält	Antal bitar	Funktion
Priority	3	Anger meddelandets prioritet vid CAN-arbitrering
Reserved	1	Reserverad bit för protokollets struktur
Data Page	1	Används för att utöka PGN-området
PDU Format	8	Anger meddelandets format och typ
PDU Specific	8	Anger destination eller grupputökning beroende på PDU-format
Source Address	8	Anger avsändande styrenhet på nätverket

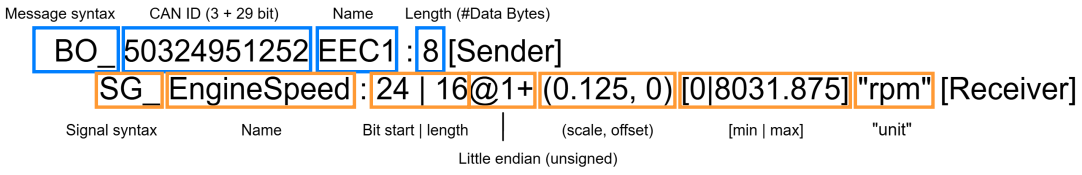
I J1939 organiseras signaler i så kallade parametergrupper. Varje parametergrupp identifieras av ett PGN och kan innehålla en eller flera signaler, ofta benämnda Suspect Parameter Numbers, SPN. En J1939-frame kan normalt innehålla upp till åtta databytes vid klassisk CAN-kommunikation. För större datamängder används transportprotokoll, där informationen delas upp över flera CAN-meddelanden.

I projektets kontext är SAE J1939 relevant eftersom simuleringsverktyget ska kunna generera fordonsrelaterade signaler som efterliknar de meddelanden som normalt förekommer i en verklig fordonsmiljö. Genom att strukturera simulerade signaler enligt J1939 kan testmiljön ta emot hastighet, motorvarvtal och andra fordonsdata på ett format som motsvarar kommunikationen mellan verkliga styrenheter. Detta gör det möjligt att använda simuleringsverktyget som en del av tidig verifiering och integration i en CAN-baserad testmiljö.

2.4.3 DBC-filer och signaldefinitioner

En DBC-fil används för att beskriva hur CAN-meddelanden och deras signaler är strukturerade. Filen fungerar som en databas över meddelanden, signaler, skalning, offset, enheter och gränsvärden. Därmed kan binär CAN-data översättas till fysikaliskt tolkbara signaler.

Syntaxen i en DBC-fil följer en tydlig struktur där varje CAN-meddelande definieras med ett meddelande-ID, ett namn och en datalängd, medan tillhörande signaler definieras separat med information om hur signalvärdet är placerat och hur det ska tolkas. Ett meddelande inleds med `BO_`, medan signaler i meddelandet definieras med `SG_`. Signaldefinitionen anger bland annat startbit, bitlängd, byteordning, teckentyp, skalning, offset, gränsvärden och enhet [9]. Figur 2.5 visar ett exempel på hur denna syntax kan se ut.



Figur 2.5: Egen illustration av syntaxen för ett CAN-meddelande och en signal i en DBC-fil.

I figuren definieras CAN-meddelandet med nyckelordet `BO_`. Därefter anges meddelandets identifierare, namn och längd i antal databytes. I exemplet har meddelandet namnet `EEC1` och en längd på 8 bytes. Under meddelandet definieras signalen med nyckelordet `SG_`. Signalen `EngineSpeed` beskriver motorvarvtalet och anges med startbit, bitlängd, byteordning, skalning, offset, tillåtna gränsvärden och enhet.

Skalning och offset används för att omvandla det råa signalvärdet från CAN-meddelandet till ett fysiskt värde enligt

$$y = x \cdot s + o, \tag{2.32}$$

där x är det råa signalvärdet, s är skalningsfaktorn, o är offset och y är det resulterande fysiska värdet. I exemplet i Figur 2.5 innebär skalningen 0.125 och offset 0 att det råa värdet multipliceras med 0.125 för att få motorvarvtalet i rpm. På detta sätt fungerar DBC-filen som ett gränssnitt mellan binär CAN-data och fysikaliskt tolkbara signaler.

2.4.4 CAN-gränssnitt och USB-anslutning

För att en dator ska kunna kommunicera med en CAN-buss krävs ett CAN-gränssnitt. Ett sådant interface fungerar som en brygga mellan datorns programvara och den fysiska eller virtuella CAN-bussen, och hanterar den underliggande signalöverföringen i enlighet med CAN-protokollets elektriska och logiska krav. Programvaran kommunicerar därmed inte direkt med databussen utan nyttjar interfacet som mellanled, vilket möjliggör en enhetlig programvarumiljö oavsett om kommunikationen sker mot en virtuell eller fysisk kanal [10].

Vector erbjuder ett flertal nätverksgränssnitt för fordonsapplikationer, däribland VN1630 och VN1640, vilka stöder CAN, CAN FD samt LIN och är vanligt förekommande i laboratorie- och testmiljöer [10]. Anslutningen mellan interface och dator sker via USB, och konfigurationen av hårdvara och kanaltilldelning hanteras via Vector Hardware Manager [11].

USB (Universal Serial Bus) är en standard för seriell kommunikation mellan datorer och externa enheter. Standarden definierar hur data och ström överförs mellan anslutna enheter och finns i flera versioner, där USB 3.0 idag är en av de vanligaste [12]

2.5 Programvaruverktyg och bibliotek

I projektet används flera programvaruverktyg och bibliotek för modellering, visualisering och kommunikation. Dessa verktyg är en förutsättning för att simuleringsverktyget ska kunna implementeras och användas i en realistisk testmiljö.

2.5.1 Python, NumPy och SciPy

Python är ett programmeringsspråk med brett användningsområde inom tekniska beräkningar, prototyputveckling och automatisering. Dess omfattande ekosystem av

bibliotek gör det särskilt lämpat för projekt där modellering, kommunikation och visualisering behöver integreras i ett och samma verktyg [13].

För numeriska beräkningar används NumPy, som erbjuder effektiv hantering av vektorer, matriser och matematiska operationer [14]. SciPy kompletterar NumPy med funktionalitet för mer avancerade vetenskapliga beräkningar, exempelvis inom linjär algebra och interpolation [15]. Tillsammans utgör dessa bibliotek grunden för implementationen av de matematiska modeller som beskriver fordonets rörelse och dynamik.

2.5.2 Matplotlib och PySide6

Matplotlib är ett etablerat visualiseringsbibliotek för Python som möjliggör skapandet av grafer och figurer med relativt lite kod [16]. Biblioteket lämpar sig väl för snabb datainspektion och enklare visualiseringsuppgifter, men saknar inbyggt stöd för interaktiva gränssnittselement och realtidsuppdatering i mer komplexa applikationer.

PySide6 bygger på Qt-ramverket och möjliggör utveckling av plattformsoberoende applikationer med avancerade gränssnittselement [17]. Ramverket erbjuder bred funktionalitet för interaktiva gränssnitt, men medför högre komplexitet i implementation och kräver mer omfattande kodstruktur jämfört med enklare visualiseringsbibliotek.

2.5.3 Python-CAN och cantools

För att hantera CAN-kommunikation från Python används biblioteket `python-can`, som erbjuder ett enhetligt gränssnitt mot olika CAN-gränssnitt. Det möjliggör konfiguration av databussar, konstruktion av meddelanden samt periodisk sändning [18]. Biblioteket `cantools` kompletterar detta genom att läsa DBC-filer och koda eller avkoda CAN-meddelanden enligt de signaldefinitioner som filerna innehåller [19].

Kombinationen av dessa två bibliotek är central för simuleringsverktygets funktion: `cantools` omvandlar fysiska signalvärden till korrekt binärt CAN-format, varefter `python-can` ansvarar för att överföra meddelandena på databussen. Detta gör det möjligt att simulera fordonssignaler på ett sätt som är kompatibelt med verkliga ECU:er och testverktyg.

2.5.4 Vector Hardware Manager och CANalyzer

Vector Hardware Manager är ett konfigurationsverktyg för Vector-baserade nätverksgränssnitt som möjliggör hantering av såväl virtuella som fysiska CAN-kanaler [11]. Verktyget tillhandahåller ett samlat gränssnitt för drivrutinsinstallation, kanaltilldelning och hårdvarukonfiguration, vilket reducerar behovet av manuell systeminställning.

CANalyzer är ett programverktyg för analys och stimulering av fordonsnätverkskommunikation, utvecklat av Vector Informatik [20]. Verktyget möjliggör observation och verifiering av datatrafik på CAN-bussar samt interaktiv ECU-diagnostik. Utöver passiv inspelning av nätverksdata erbjuder CANalyzer funktionalitet för aktiv stimulering, konfigurerbara analysblock samt skriptprogrammering via det proprietära

språket Communication Access Programming Language (CAPL), vilket ger stöd för avancerade och automatiserade analysuppgifter.

3

Metod

Detta kapitel beskriver den metod som planerades för att besvara rapportens frågeställningar och för att utveckla ett CAN-baserat simuleringsverktyg för tidig verifiering av fordonsbeteenden i labbmiljö. Metoden utformades inför projektets genomförande för att stödja ett utvecklingsorienterat arbete där fordonsmodellering, mjukvaruarkitektur, användarinteraktion och CAN-kommunikation behövde behandlas som sammanhängande delar av samma system.

Eftersom arbetet syftade till att ta fram ett principbevis planerades metoden med fokus på funktionell utveckling, stegvis integration och verifiering av systemets huvudkedja. Målet var inte att validera en fullständig fordonsmodell mot verklig kördata, utan att undersöka hur simulerade fordonsbeteenden kunde genereras, visualiseras och överföras som CAN-signaler till en testmiljö. Metoden utformades därför som en iterativ prototyputveckling, där systemets komplexitet skulle öka successivt.

3.1 Design och övergripande arbetssätt

Projektet planerades som ett utvecklingsorienterat examensarbete med målet att undersöka hur ett mjukvarubaserat simuleringsverktyg kan användas som stöd för tidigare verifiering av fordonsmjukvara. Den valda metoden kan beskrivas som iterativ prototyputveckling, där en första förenklad lösning skulle etableras och därefter stegvis byggas ut med högre funktionell komplexitet.

Det iterativa arbetssättet valdes eftersom projektets olika delar hade tydliga beroenden mellan varandra. Fordonsmodellen behövde kunna generera tillstånd som kunde användas av användargränssnittet, samtidigt som samma tillstånd behövde kunna översättas till signaler för extern kommunikation. Genom att utveckla systemet stegvis skulle delkomponenter kunna verifieras innan de integrerades i en större helhet.

Arbetet planerades i fyra huvudsakliga delar. Den första delen avsåg kravbild och behovsanalys. Den andra delen behandlade modellering och systemarkitektur. Den tredje delen omfattade implementation av prototypens huvudsakliga moduler. Den fjärde delen avsåg verifiering och utvärdering av verktygets funktionalitet i relation till projektets frågeställningar.

3.2 Kravbild och behovsanalys

Inför projektets genomförande skulle en inledande kravbild formuleras utifrån projektets syfte, den industriella kontexten och behovet av tidig verifiering i labbmiljö.

Kravbilden skulle användas för att avgränsa verktygets funktionalitet och för att styra val av modellnivå, systemarkitektur och verifieringsstrategi.

De övergripande kraven delades in i funktionella och strukturella krav. De funktionella kraven avsåg att verktyget skulle kunna simulera centrala körbeteenden, hantera användarstyrning i realtid och generera fordonsrelaterade tillstånd som kunde användas för signalöverföring. De strukturella kraven avsåg att systemet skulle vara modulärt, utbyggbart och möjligt att verifiera stegvis.

Eftersom arbetet avgränsades till ett principbevis planerades utvecklingen med fokus på funktionell korrekthet och tydlig systemstruktur snarare än fullständig fysikalisk noggrannhet. Detta innebar att modellen skulle vara tillräckligt realistisk för att generera relevanta signaler, men inte nödvändigtvis motsvara ett verkligt fordons beteende med hög precision.

3.3 Modelleringsmetod

Fordonsmodellen planerades med utgångspunkt i förenklade fordonsdynamiska samband. Syftet var att skapa en modell som kunde generera rimliga fordonsbeteenden på systemnivå, snarare än att återskapa ett verkligt fordons dynamik med hög detaljnoggrannhet. Detta metodval motiverades av att projektet främst skulle undersöka möjligheten att använda simulerade tillstånd som underlag för verifiering i en labb-baserad testmiljö.

Modelleringsarbetet skulle genomföras stegvis. Först skulle centrala tillstånd definieras, exempelvis position, hastighet, acceleration, orientering, hjulhastigheter och motorrelaterade variabler. Därefter skulle matematiska samband formuleras för hur dessa tillstånd skulle uppdateras över tid. Modellen skulle köras i diskreta tidssteg för att möjliggöra realtidsnära simulering och interaktiv styrning.

En viktig metodprincip var att modellens komplexitet skulle kunna ökas utan att den grundläggande strukturen behövde förändras. Därför planerades modellens olika delar som separata delsystem, exempelvis rörelsemodell, drivlina, bromsning, styrning och reglering. Detta skulle göra det möjligt att utveckla och verifiera delmodeller var för sig innan de kopplades samman.

3.4 Arkitekturmetod

Systemarkitekturen planerades enligt principen om separation av ansvar. Fordonsmodell, användargränssnitt och CAN-kommunikation skulle utformas som separata moduler, men samtidigt kunna integreras till en gemensam simuleringsmiljö. På så sätt skulle systemet bli lättare att vidareutveckla, felsöka och verifiera.

En objektorienterad struktur valdes som metod för att skapa tydliga gränssnitt mellan systemets komponenter. Centrala tillstånd skulle samlas i gemensamma datastrukturer, medan beräkningslogik, visualisering och kommunikation skulle hållas separerade. Detta skulle göra det möjligt att använda samma simulerade tillstånd för flera syften: numerisk uppdatering, grafisk visualisering och signalgenerering.

Arkitekturen planerades utifrån tre huvudsakliga principer. Den första var modularitet, vilket innebar att varje delsystem skulle ha ett tydligt ansvar. Den andra var utbyggbarhet, vilket innebar att nya komponenter och signaler skulle kunna läggas till utan omfattande omstrukturering. Den tredje var verifierbarhet, vilket innebar att varje del av systemet skulle kunna testas isolerat innan fullständig integration.

3.5 Integrationsmetod

Integrationen mellan simuleringsverktyget och den externa testmiljön planerades som en stegvis process. Syftet med detta var att minska risken vid övergången från ren mjukvarusimulering till kommunikation med en hårdvarunära testmiljö. I stället för att direkt koppla hela systemet mot fysisk utrustning skulle integrationen delas upp i flera verifieringsnivåer.

Den första nivån avsåg intern verifiering av att modellens tillstånd kunde omvandlas till signalvärden med rätt format och enheter. Den andra nivån avsåg verifiering av att CAN-meddelanden kunde skapas och hanteras i en kontrollerad mjukvarumiljö. Den tredje nivån avsåg verifiering av kommunikation mot en testmiljö där signalerna kunde observeras och analyseras.

Den stegvisa integrationsmetoden valdes för att kunna särskilja fel i modellberäkningar, signalmappning, DBC-kodning och fysisk kommunikation. Därmed skulle integrationsarbetet kunna genomföras mer kontrollerat och med tydligare felsökning mellan respektive nivå.

3.6 Verifieringsmetod

Verifieringen planerades på flera nivåer för att bedöma systemets funktionalitet och stabilitet. På den lägsta nivån skulle avgränsade beräkningar och funktioner kontrolleras, exempelvis hantering av insignaler, begränsningar, teckenriktning och grundläggande fysikaliska samband.

Därefter skulle integrationstester användas för att kontrollera samspelet mellan delsystem. Detta omfattade bland annat att modellens tillstånd skulle uppdateras konsekvent över tid och att beräknade storheter skulle kunna användas av både visualisering och signalhantering. Särskild vikt skulle läggas vid den beräkningskedja som återkommer vid varje tidssteg, där indata påverkar krafter, acceleration, hastighet, position och därefter genererade signaler.

För CAN-relaterad verifiering planerades en stegvis strategi där signalernas struktur, kodning, mottagning och observerbarhet skulle kontrolleras. Syftet var att inte enbart undersöka om meddelanden kunde skickas, utan även om de kunde tolkas med rätt innehåll efter överföring. Verifieringen skulle därför omfatta både virtuell testning och, om möjligt, fysisk anslutning till testmiljö.

3.7 Utvärderingskriterier

Projektets resultat planerades att utvärderas mot rapportens tre frågeställningar. Fordonsmodellen skulle bedömas utifrån om den kunde generera centrala tillstånd för realistiska körbeteenden på systemnivå. Användargränssnittet skulle bedömas utifrån om det möjliggjorde styrning, övervakning och felsökning av simuleringen i realtid. CAN-kommunikationen skulle bedömas utifrån om simulerade tillstånd kunde kopplas till definierade signaler, kodas och överföras till en testmiljö.

Utöver funktionell verifiering skulle verktygets relevans analyseras i ett shift-left-perspektiv. Detta skulle göras genom att bedöma i vilken utsträckning prototypen kunde bidra till tidigare, mer kontrollerbar och mer repeterbar verifiering av fordonsmjukvara. Samtidigt skulle kvarstående begränsningar analyseras för att avgöra vilka delar som behöver vidareutvecklas innan ett sådant verktyg kan användas i större industriell omfattning.

3.8 Metodens begränsningar

Metoden var anpassad för att utveckla och utvärdera ett principbevis, inte för att validera en fullständig fordonsmodell mot verklig kördata. Utvärderingen planerades därför med fokus på funktionell korrekthet, systemintegration och möjligheten att använda simulerade signaler i en labb-baserad testmiljö.

Detta innebar att modellens exakta fysikaliska noggrannhet, långtidsprestanda, signalernas latens och jitter samt generaliserbarhet till fler fordonstyper eller testfall inte skulle behandlas fullständigt inom projektets ramar. Dessa aspekter betraktades i stället som möjliga områden för framtida utveckling och vidare verifiering.

4

Genomförande

Detta kapitel beskriver hur simuleringsverktyget utvecklades från en första förenklad fordonsmodell till en integrerad prototyp med grafiskt användargränssnitt och CAN-kommunikation mot Volvo Bussars testmiljö. Kapitlet är strukturerat efter utvecklingsflödet i projektet, där varje del byggde vidare på föregående steg. Samtidigt beskrivs den färdiga funktionaliteten inom respektive del för att tydliggöra hur verktyget slutligen kom att fungera.

En central utgångspunkt i genomförandet var att öka systemets komplexitet successivt. Arbetet inleddes med en avgränsad modell för att verifiera grundläggande samband mellan acceleration, hastighet, position och orientering. Därefter byggdes modellen ut med fler fordonsdynamiska egenskaper, komponentbaserad drivlina och objektorienterad tillståndshantering. Parallellt utvecklades ett grafiskt användargränssnitt för styrning, visualisering och felsökning i realtid. I det avslutande steget kopplades modellens tillståndsvariabler till CAN-signaler som kunde skickas till en virtuell och fysisk testmiljö.

Den färdiga prototypen består därmed av tre huvudsakliga delar: en fordonsmodell, ett grafiskt användargränssnitt och en CAN-modul. Fordonsmodellen beräknar fordonets dynamiska tillstånd, användargränssnittet används för att styra och övervaka simuleringen, och CAN-modulen överför utvalda tillståndsvariabler till testmiljön. Den slutliga runtime-arkitekturen är GUI-ledd, vilket innebär att huvudfönstret äger simuleringsobjekten, driver simuleringens tidssteg och vidarebefordrar modellens resultat till både visualisering och CAN-kommunikation.

4.1 Initial fordonsmodell och grundläggande simulering

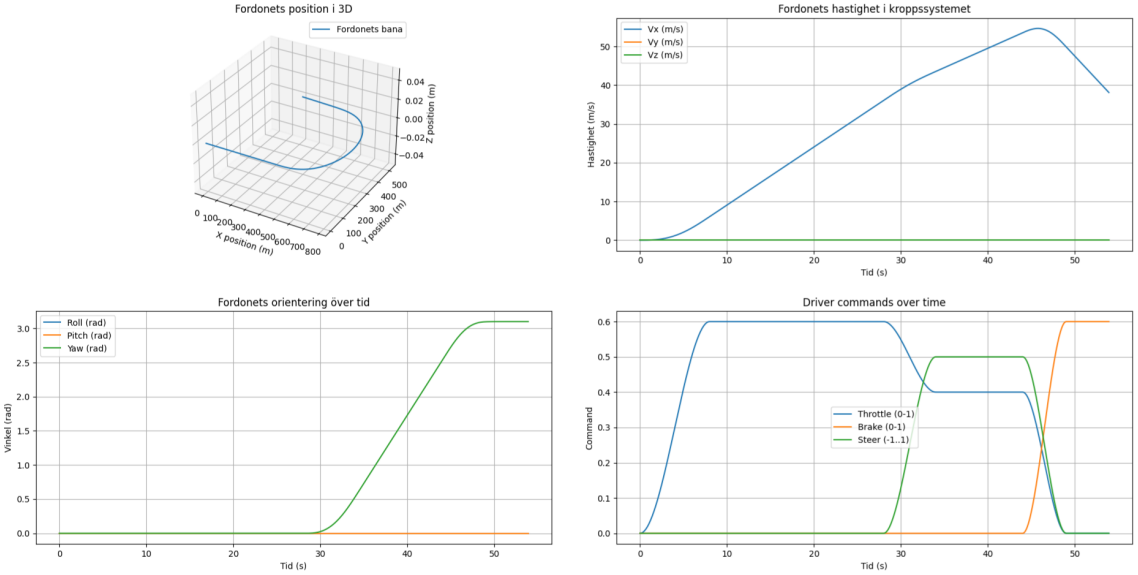
Den första versionen av simuleringsverktyget utvecklades för att etablera en matematisk grund för fordonets rörelse. Modellen utgick från ramverket med sex frihetsgrader beskrivet i avsnitt 2.2.4, men förenklades inledningsvis till rörelse i ett plant koordinatsystem. Rörelse i höjdled (z) samt rotation kring längdaxeln (ϕ) och tväraxeln (θ) låstes därför i den första implementationen, vilket innebar att $v_z = \phi = \theta = 0$. Modellen beskrev därmed longitudinell rörelse, lateral förflyttning och rotation kring den vertikala axeln, yaw (ψ).

Syftet med denna förenkling var att skapa en överskådlig simuleringsmiljö där grundläggande samband kunde kontrolleras innan mer avancerade delar infördes. Den longitudinella accelerationen beräknades utifrån kraftbalansen i ekvation 2.5, där nettokraften användes som grund för acceleration. Hastighet och position uppdaterades därefter med explicit Euler-integration enligt ekvationerna 2.20 och 2.21, med tidssteget $\Delta t = 0{,}01$ s. Styrningen implementerades med yawhastighetsmodellen i

ekvation 2.12, varvid orienteringen uppdaterades enligt ekvation 2.24.

För att beskriva fordonets rörelse användes den globala referensramen $\{I\}$ och den kroppsfixa referensramen $\{B\}$ enligt avsnitt 2.2.5. Storheter som acceleration och hastighet beräknades i $\{B\}$, medan fordonets position uttrycktes i $\{I\}$. Omvandlingen mellan referensramarna gjordes med rotationsmatrisen $\mathbf{R}_{IB}$ enligt ekvation 2.19.

Till den första modellen utvecklades ett enkelt grafiskt gränssnitt med Matplotlib. Syftet med detta gränssnitt var inte att skapa den slutliga användarmiljön, utan att snabbt kunna verifiera att rörelseekvationerna gav rimliga resultat under simulering. Den första versionen visas i Figur 4.1.



Figur 4.1: Första versionen av det grafiska gränssnittet.

Figur 4.1 visar den första visuella representationen av fordonets rörelse. Fordonet visualiserades som en punkt i ett tredimensionellt koordinatsystem, vilket gjorde det möjligt att följa position, hastighet och acceleration under simuleringens gång. Denna enkla visualisering var särskilt användbar i början av projektet eftersom fel i koordinattransformationer, teckenriktningar eller numerisk integration snabbt kunde upptäckas.

Den initiala modellen användes för att verifiera att ekvationerna 2.20–2.24 implementerades korrekt. Acceleration, hastighet och position vid konstant ingångssignal jämfördes med analytiskt beräknade referensvärden. Detta fungerade som ett tidigt verifieringssteg och gjorde det möjligt att identifiera fel i grundläggande beräkningar innan modellen byggdes ut med fler funktioner.

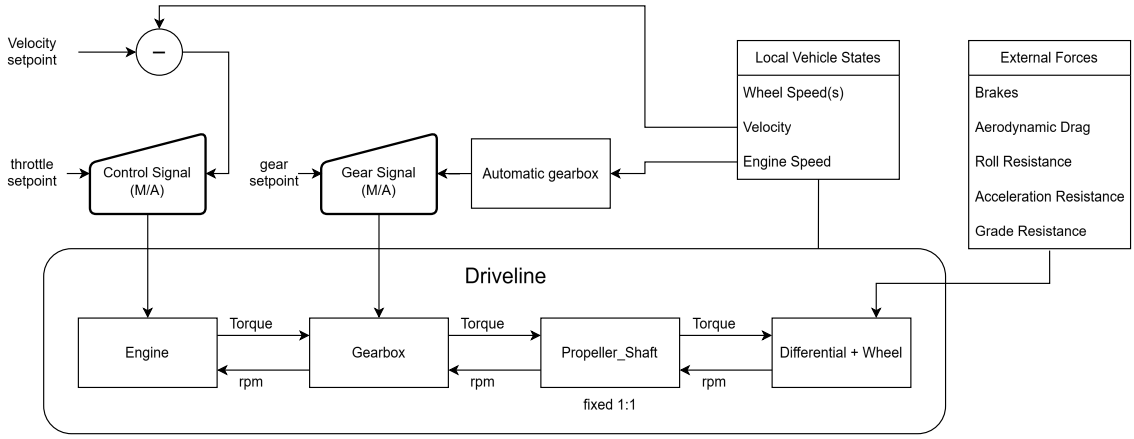
4.2 Utökad fordonsmodell och tillståndshantering

Efter att den grundläggande simuleringsloopen hade etablerats vidareutvecklades modellen för att hantera fler fordonsdynamiska samband. Den tidigare förenklade modellen byggdes ut till en mer generell struktur där fordonets rörelse kunde beskrivas

med sex frihetsgrader. Detta innebar att modellstrukturen kunde hantera rörelse längs tre axlar samt rotation kring dessa.

Utökningen innebar att rörelse längs z-axeln samt roll ϕ och pitch θ kunde inkluderas i modellens tillstånd. Även om alla frihetsgrader inte behövde användas i varje testfall skapade denna struktur en mer generell grund för fortsatt utveckling. På så sätt kunde modellen behålla enkelheten i grundläggande körfall, samtidigt som den kunde byggas ut för mer avancerade scenarier.

För att ge en övergripande bild av den vidareutvecklade fordonsmodellen visas modellens huvudstruktur i Figur 4.2. Figuren introducerar först modellens principiella flöde, från förarinput till uppdaterat fordonstillstånd, innan den mer detaljerade objektorienterade strukturen beskrivs.



Figur 4.2: Övergripande struktur för fordonsmodellen i simuleringsverktyget. Figuren visar hur förarinput, drivlina, kraftberäkningar och kinematisk uppdatering samverkar för att uppdatera fordonets tillstånd.

Figur 4.2 visar hur fordonsmodellen är uppbyggd kring ett flöde från användarens kommandon till fordonets rörelse. Förarens gas, broms och styrning behandlas först som indata till modellen. Dessa påverkar drivlinan och de krafter som verkar på fordonet, vilket därefter används för att beräkna acceleration, hastighet, position och orientering. Figuren fungerar därmed som en systemöversikt över modellens huvudprincip och visar hur olika beräkningsdelar kopplas samman i simuleringsloopen.

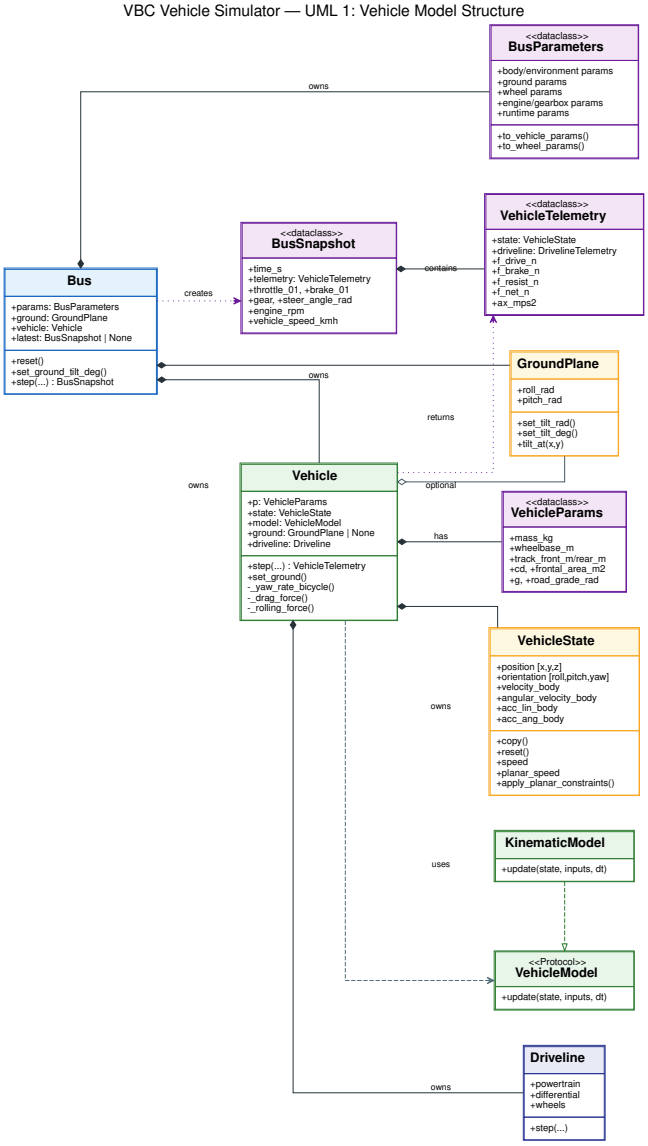
För att få mer realistiska hastighetsförlopp infördes motståndskrafter i modellen. Luftmotstånd och rullmotstånd modellerades enligt ekvationerna 2.9 och 2.10, och kraften från väglutning enligt ekvation 2.11. Dessa krafter påverkar fordonets acceleration och gör att fordonets hastighet förändras på ett mer realistiskt sätt än i en modell där rörelsen enbart styrs av gaspådrag.

I samband med utökningen utvecklades en mer systematisk tillståndshantering. Centrala fordonsstorheter såsom position $\mathbf{p}_I$, hastighet $\mathbf{v}_B$, acceleration, vinkelhastighet $\boldsymbol{\omega}_B$ och Euler-vinklar $\boldsymbol{\eta}$ samlades i ett gemensamt tillståndsobjekt, `VehicleState`. Tillståndsobjektet innehåller även relaterade storheter som används i simuleringen, exempelvis hastighet, orientering och acceleration i fordonets kroppsfixa koordinatsystem.

Den centraliserade tillståndshanteringen blev en viktig del av systemets arkitektur. Fordonsmodellen uppdaterar tillstånden vid varje tidssteg, medan användargränssnittet använder samma tillstånd för visualisering. CAN-modulen använder därefter simuleringsresultatet som grund för signalgenerering. På så sätt minskade risken för att olika delar av programmet skulle arbeta med olika versioner av samma fordonsdata.

Programvaran implementerades objektorienterat. Gemensamma egenskaper och metoder kunde därmed samlas i separata klasser, medan mer specifika komponenter kunde isoleras i egna moduler. Denna struktur minskade behovet av upprepad kod och gjorde det enklare att lägga till nya komponenter eller förändra enskilda delar av modellen.

Efter den övergripande modellstrukturen kan den objektorienterade implementationen beskrivas mer detaljerat. Figur 4.3 visar relationen mellan den övergripande bussmodellen, tillståndsobjektet, markmodellen och den kinematiska modellen. Diagrammet visar därmed hur simuleringsverktygets fordonsmodell byggdes upp som en separat del av systemet, frikopplad från både användargränssnitt och CAN-kommunikation.

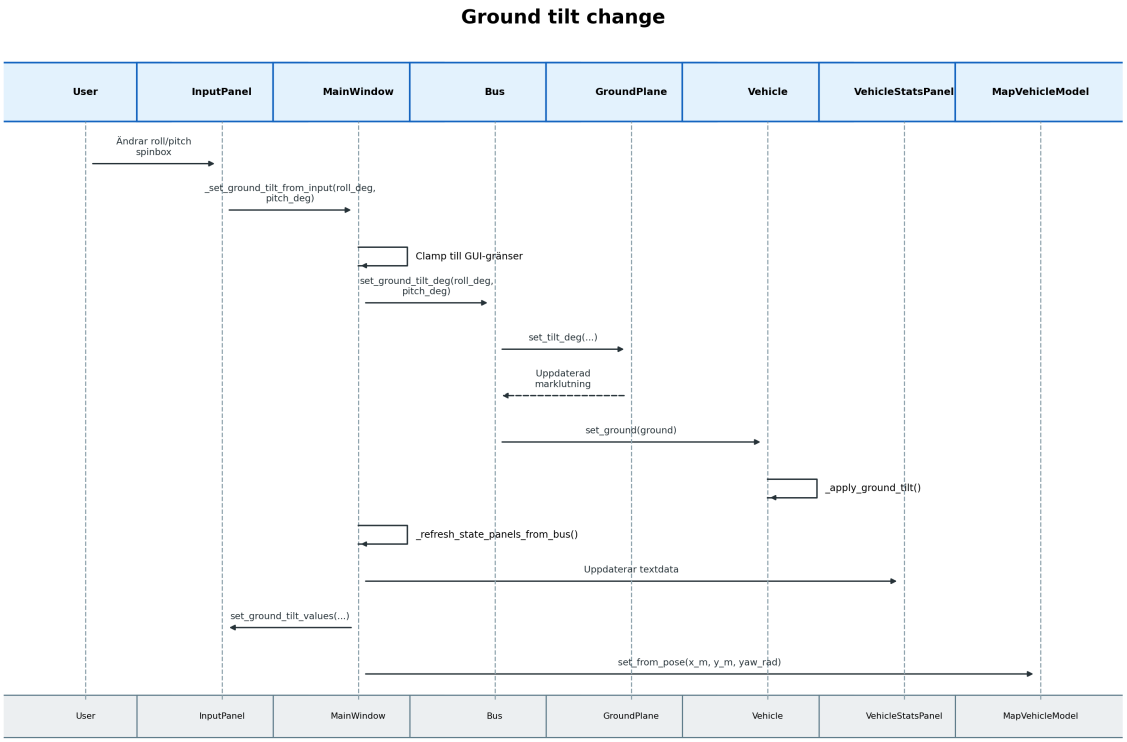


Figur 4.3: UML-klassdiagram över fordonsmodellens centrala struktur. Diagrammet visar hur `Bus` fungerar som fasad mot simuleringen, hur `Vehicle` hanterar fordonsdynamiken och hur `VehicleState` samlar fordonets dynamiska tillstånd.

Figur 4.3 visar att `Bus` fungerar som modellens huvudsakliga gränssnitt mot övriga systemet. Klassen äger konfigurationen i `BusParameters`, markmodellen `GroundPlane` och den underliggande `Vehicle`-instansen. Vid varje simuleringssteg returneras ett `BusSnapshot`, vilket gör att användargränssnittet och CAN-modulen kan läsa modellens resultat utan att direkt behöva känna till hur beräkningarna sker internt. Detta var en viktig arkitektonisk avgränsning eftersom fordonsmodellen då kunde utvecklas och testas oberoende av visualisering och kommunikation.

För att kunna testa fordonets respons vid lutande mark infördes en enkel markmodell, `GroundPlane`. Denna modell beskriver konstant roll och pitch för markplanet och kan uppdateras från användargränssnittet. Flödet vid ändring av roll och pitch visas i Figur 4.4. Figuren visar hur användarens inmatning via gränssnittet uppdaterar

markmodellen och därefter fordonets tillstånd och visualisering.



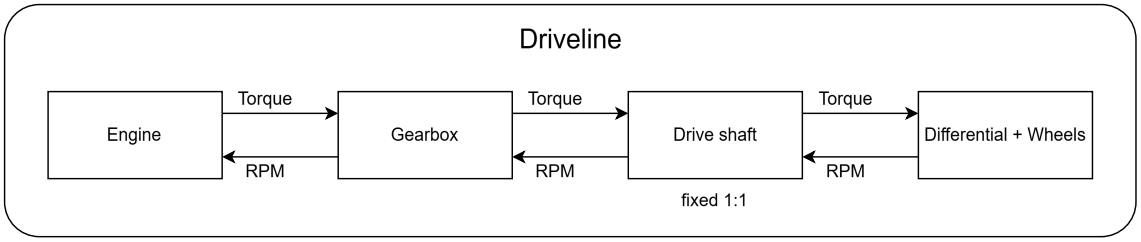
Figur 4.4: Sekvensdiagram över ändring av marklutning. Användarens roll- och pitchvärden skickas från `InputPanel` till `Bus`, som uppdaterar `GroundPlane` och kopplar den till `Vehicle`.

Figur 4.4 visar att marklutningen hanteras som en del av modellens omgivning snarare än som en direkt ändring av förarinput. Detta gjorde det möjligt att låta samma gas-, broms- och styrkommandon ge olika fordonsrespons beroende på markens lutning, vilket ökade modellens användbarhet för enkla scenariotester.

4.3 Implementation av drivlina och körlogik

För att skapa en mer realistisk koppling mellan förarens indata och fordonets rörelse implementerades en komponentbaserad drivlina. I stället för att låta gaspådraget direkt ge upphov till acceleration delades kraftöverföringen upp i flera delmodeller. Dessa representerade centrala delar av fordonet, såsom motor, växellåda, drivaxel, differential och hjul.

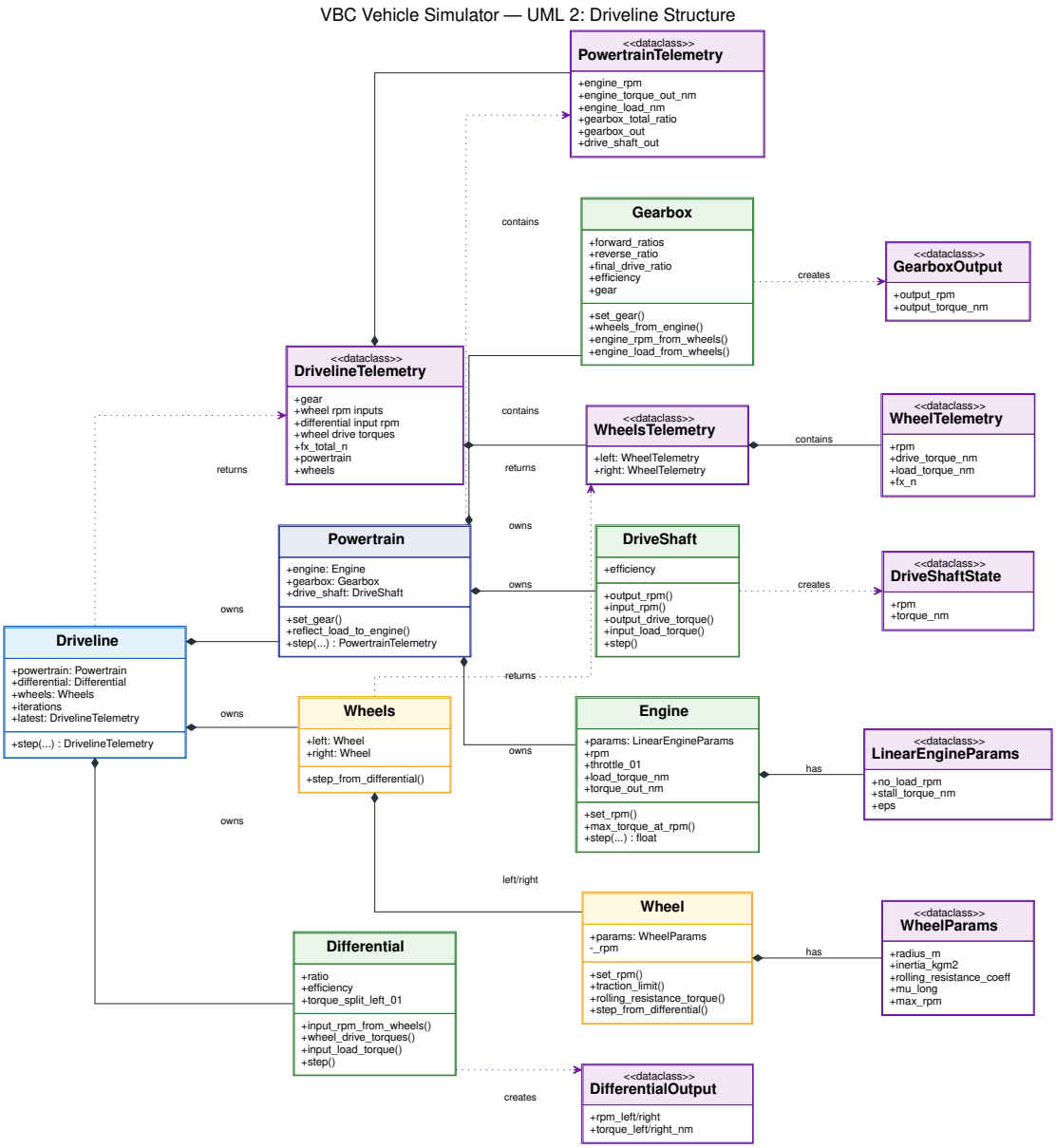
Som ett första steg visas drivlinans övergripande struktur i Figur 4.5. Figuren ger en principiell bild av hur gaspådrag omvandlas till drivkraft genom drivlinans huvudkomponenter. Den fungerar därmed som en översikt innan den mer detaljerade objektorienterade implementationen beskrivs.



Figur 4.5: Övergripande struktur för drivlinan i simuleringsverktyget.

Figur 4.5 visar drivlinans huvudsakliga kraftväg. Motorn genererar ett moment baserat på gaspådrag och motorvarvtal. Momentet förs därefter vidare genom växellådan och drivaxeln, innan det fördelas via differentialen till hjulen. På hjulnivå omvandlas momentet till longitudinell drivkraft, vilken sedan används i fordonets kraftbalans för att beräkna acceleration och hastighetsförändring.

För att implementera denna struktur i programvaran byggdes drivlinan upp med separata klasser för respektive delkomponent. Den objektorienterade strukturen visas i Figur 4.6. Syftet med figuren är att tydliggöra hur drivlinans komponenter är organiserade i koden och hur telemetri från varje del kan samlas och användas vidare i modellen.



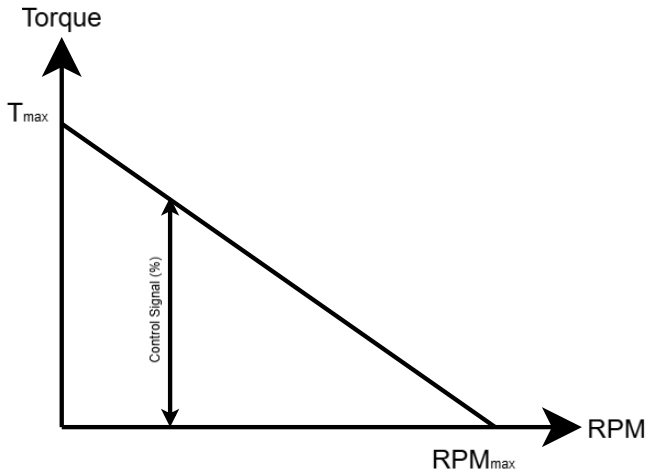
Figur 4.6: UML-klassdiagram över drivlinans objektorienterade struktur. Diagrammet visar hur `Driveline` samlar `Powertrain`, `Differential` och `Wheels`, samt hur `Powertrain` i sin tur består av `Engine`, `Gearbox` och `DriveShaft`.

Av Figur 4.6 framgår att `Powertrain` avgränsades till motor, växellåda och drivaxel, medan differential och hjul placerades utanför denna klass men fortfarande inom `Driveline`. Denna uppdelning gjorde modellen mer modulär. Exempelvis kan motormodellen bytas ut utan att hjulmodellen behöver förändras, och differentialens momentfördelning kan justeras utan att växellådans beräkningar påverkas. Strukturen gjorde det också möjligt att samla telemetri från varje delsteg, vilket senare användes för visualisering i gränssnittet och för signalgenerering.

De olika delmodellerna utformades så att utdata från en komponent kunde användas som indata till nästa komponent. Motorns genererade moment kunde därmed föras

vidare genom växellådan och drivaxeln till hjulen. Hjulmodellen användes sedan för att koppla drivmoment och hjulvarvtal till krafter mot vägplanet. På så sätt kunde modellen beskriva hur moment, varvtal och krafter fortplantades genom drivlinan.

Motorn genererar ett utgående moment baserat på gaspådrag och aktuellt motorvarvtal. I motormodellen används en förenklad linjär approximation, inspirerad av en DC-motormodell, se Figur 4.7. Det innebär att det tillgängliga motormomentet minskar när motorvarvtalet ökar. Gaspådraget används som en skalningsfaktor mellan noll och ett, vilket innebär att användarens inmatning bestämmer hur stor del av det tillgängliga momentet som används i simuleringen.



Figur 4.7: Moment- och varvtalskarakteristik för den förenklade motormodellen.

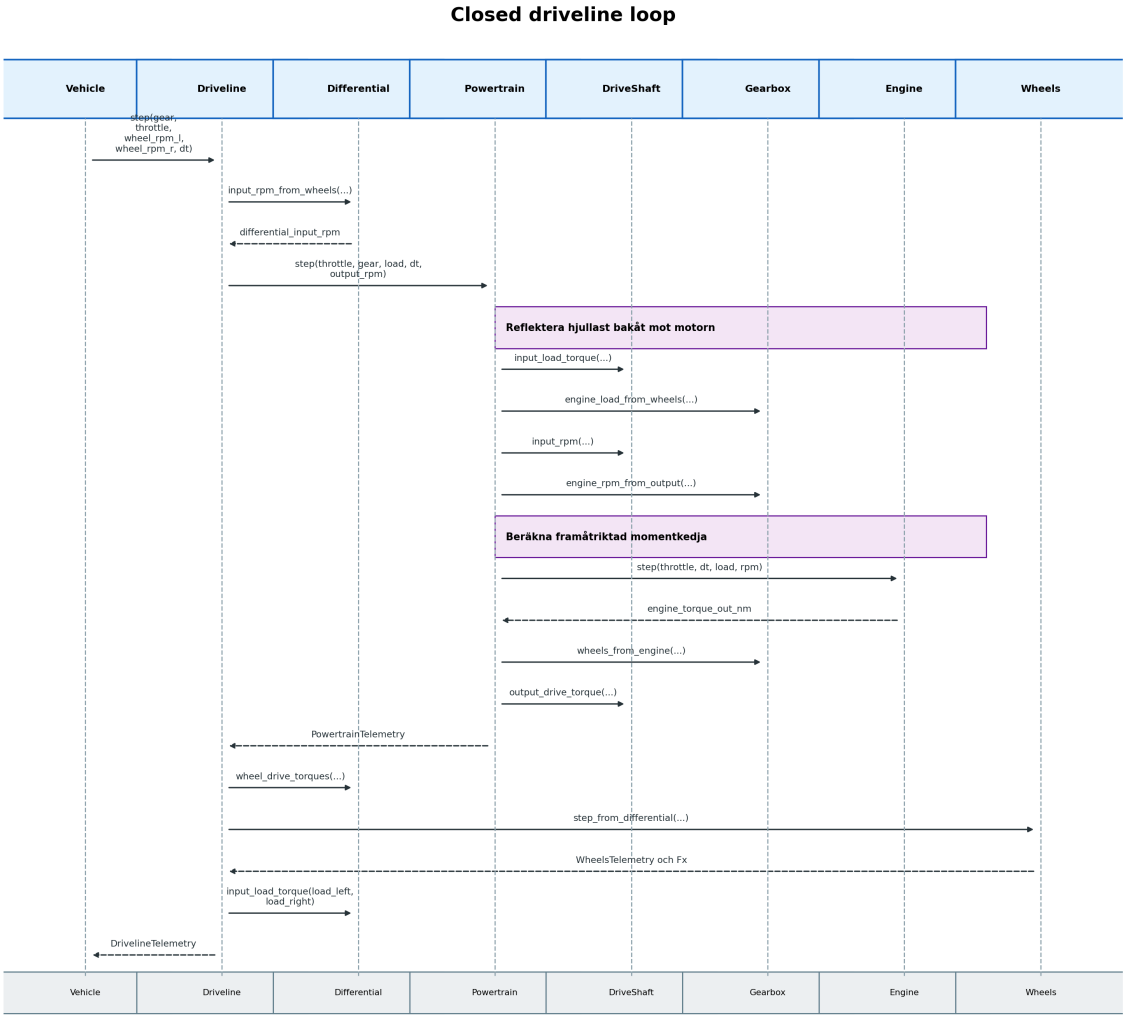
Figur 4.7 visar den förenklade motorkarakteristiken som används i modellen. Vid lågt varvtal kan motorn generera ett högre moment, medan det tillgängliga momentet minskar när varvtalet närmar sig modellens övre varvtalsgräns. Denna förenkling är inte avsedd att exakt beskriva en verklig bussmotor, men den ger en rimlig relation mellan gaspådrag, varvtal och utgående moment för funktionella simuleringstester.

Växellådan påverkar relationen mellan motorvarvtal och hjulhastighet genom vald växel. En låg växel ger högre momentförstärkning men lägre utgående varvtal, medan en högre växel ger lägre momentförstärkning men högre utgående varvtal. Drivaxeln för vidare momentet till differentialen, där momentet fördelas mellan vänster och höger hjul. Hjulmodellen omsätter därefter hjulmomentet till longitudinell kraft mot marken.

Samtidigt återkopplas hjulens rotationshastighet till drivlinan. Detta gör att motorns arbetspunkt påverkas av fordonets aktuella hastighet, vald växel och de motstånd som verkar på fordonet. Drivlinan modelleras därför inte enbart som en framåtriktad momentkedja, utan som en sluten beräkningsloop där last och varvtal också reflekteras tillbaka från hjulen mot motorn.

För att visa hur drivlinans komponenter samverkar under ett tidssteg visas denna slutna beräkningsloop i Figur 4.8. Figuren visar hur hjulens varvtal först används för

att bestämma reflekterat varvtal och last uppströms i drivlinan, och hur motorns genererade moment därefter förs framåt till hjulen.



Figur 4.8: Sekvensdiagram över drivlinans slutna beräkningsloop. Diagrammet visar hur hjulvarvtal först används för att reflektera last bakåt mot motorn, och hur motorns genererade moment därefter förs framåt genom växellåda, drivaxel, differential och hjul.

Figur 4.8 visar varför drivlinan inte kan betraktas som en enkel envägskedja. Hjulens rörelse påverkar det varvtal som reflekteras till motorn, samtidigt som motorns moment påverkar hjulens drivkraft. Denna återkoppling gjorde att drivlinan kunde beskriva relationen mellan fordonshastighet, växel, motorvarvtal och drivmoment på ett mer realistiskt sätt än i den initiala modellen.

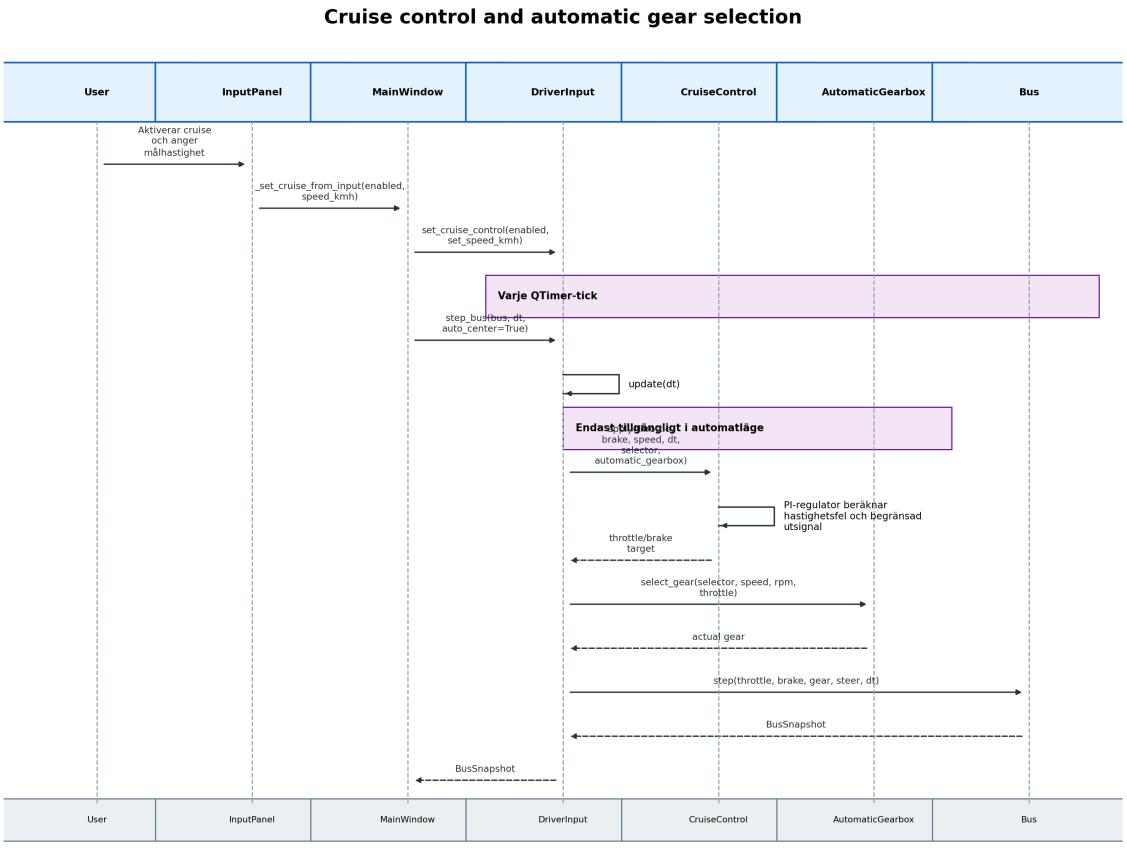
Den komponentbaserade strukturen gjorde det möjligt att följa både moment och varvtal genom hela drivlinan. Detta var viktigt eftersom modellen inte enbart skulle ge ett hastighetsvärde, utan även generera interna tillstånd som kunde användas för visualisering och senare CAN-signalering. Exempel på sådana tillstånd är motorvarvtal, motormoment, växellådans utgående moment, drivaxelns moment och hjulens

varvtal.

Utöver drivlinan implementerades körlogik för bromsning, växling och farthållare. Bromsmodellen användes för att skapa en kraft som motverkar fordonets rörelseriktning. Växlingslogiken användes för att ändra växel baserat på fordonets aktuella tillstånd. I prototypen implementerades både manuell växling och automatisk växling, vilket gjorde det möjligt att använda modellen i olika typer av testfall.

Farthållaren implementerades som en PI-regulator enligt avsnitt 2.3.3, där reglerfelet definierades som $e = v_{\text{ref}} - v$. Den proportionella termen reagerade på aktuell hastighetsskillnad, medan integralledet reducerade kvarstående fel vid konstant referenshastighet. Regulatorns utsignal begränsades till intervallet $[0, 1]$ för att motsvara ett fysiskt realiserbart gaspådrag, i enlighet med diskussionen om begränsning av styrsignaler i avsnitt 2.3.3.

Flödet för farthållare och automatväxling visas i Figur 4.9. Figuren visar hur användarens inställning av målhastighet går via `InputPanel` och `MainWindow` till `DriverInput`, samt hur `CruiseControl` och `AutomaticGearbox` används vid varje tidssteg.



Figur 4.9: Sekvensdiagram över farthållare och automatväxling. Farthållaren justerar gas och broms utifrån hastighetsfelet, medan automatväxlingen väljer faktisk växel baserat på selectorläge och aktuellt driftläge.

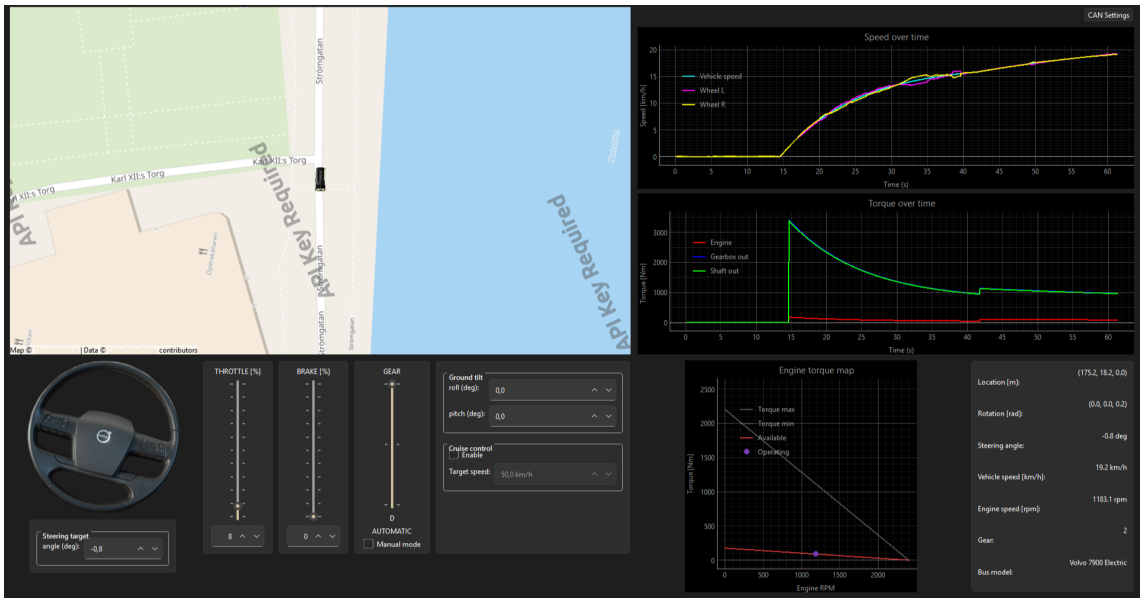
Figur 4.9 visar att farthållaren inte är en separat fordonsmodell, utan en del av körlogiken före själva fordonssteget. Den påverkar alltså de kommandon som skickas till `Bus.step()`, snarare än att ändra fordonstillståndet direkt. Detta gjorde implementationen tydligare eftersom reglering, förarinput och fordonsdynamik kunde hållas åtskilda.

Genom denna körlogik kunde verktyget hantera både manuell och delvis automatiserad körning. Användaren kunde styra fordonet fritt genom gas, broms och ratt, men också aktivera funktioner som automatisk växling och farthållare för mer kontrollerade körförlopp.

4.4 Utveckling av grafiskt användargränssnitt

Det grafiska användargränssnittet vidareutvecklades parallellt med fordonsmodellen. Den första Matplotlibbaserade visualiseringen var användbar för att kontrollera grundläggande rörelse, men blev begränsad när modellens komplexitet ökade. Därför utvecklades ett mer avancerat gränssnitt med PySide6 och Qt Widgets. Detta gjorde det möjligt att skapa ett mer interaktivt gränssnitt med reglage, statuspaneler, grafer, karta och CAN-inställningar.

Den vidareutvecklade versionen av gränssnittet visas i Figur 4.10. Figuren visar hur visualiseringen gick från en enkel punktmodell till en mer komplett användarmiljö där fordonets rörelse och interna tillstånd kunde övervakas samtidigt.



Figur 4.10: Vidareutvecklad version av det grafiska användargränssnittet med visualisering av fordonets rörelse och tillstånd.

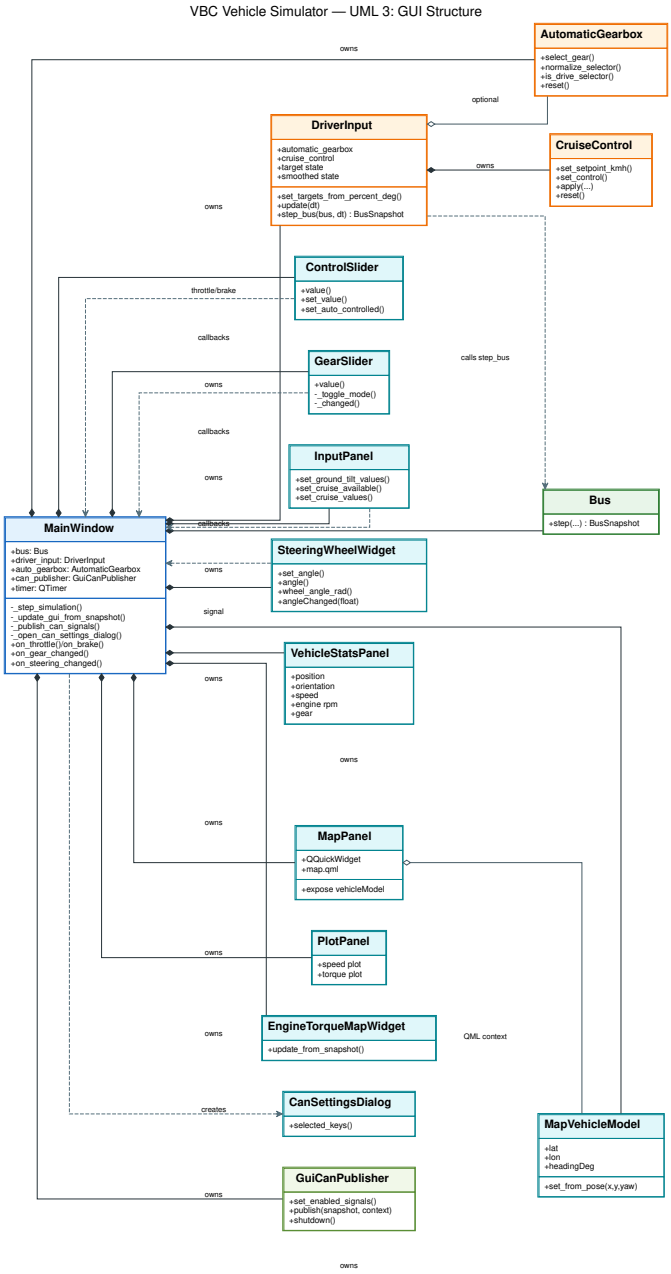
I det vidareutvecklade gränssnittet representerades fordonet av en bussikon som rörde sig över en kartliknande vy. Position och orientering från fordonsmodellen omvandlades till gränssnittets koordinatsystem, vilket gjorde det möjligt att följa fordonets rörelse visuellt. Detta gav en mer intuitiv bild av modellens beteende än

den första punktbaserade visualiseringen.

Gränssnittet kompletterades även med grafer och statusfält för centrala tillstånd och telemetridata. Typiska exempel på sådana värden är fordonets hastighet, acceleration, motorvarvtal, växel, styrvinkel, koordinater, rotationsvinklar och moment i drivlinans komponenter. Dessa visualiseringar gjorde det möjligt att följa modellens beteende mer detaljerat än vad kartvyn ensam möjliggjorde.

För att möjliggöra interaktiv styrning infördes kontroller för gas, broms, rattutslag, växelval och farthållare. Användaren kunde därmed påverka simuleringen under körning och samtidigt observera hur fordonets tillstånd förändrades. Detta gjorde gränssnittet användbart både för demonstrationssyfte och för iterativ utveckling av modellen.

När gränssnittet vidareutvecklades fick `MainWindow` rollen som central koordinator för hela körningen. Figur 4.11 visar hur huvudfönstret äger fordonsmodellen, förarinput, automatväxling, CAN-publicering och de visuella panelerna. Detta var en medveten struktur eftersom den aktuella runtime-arkitekturen är GUI-ledd: gränssnittet startar simuleringen, äger tidssteget och vidarebefordrar modellens resultat till både visualisering och CAN-kommunikation.

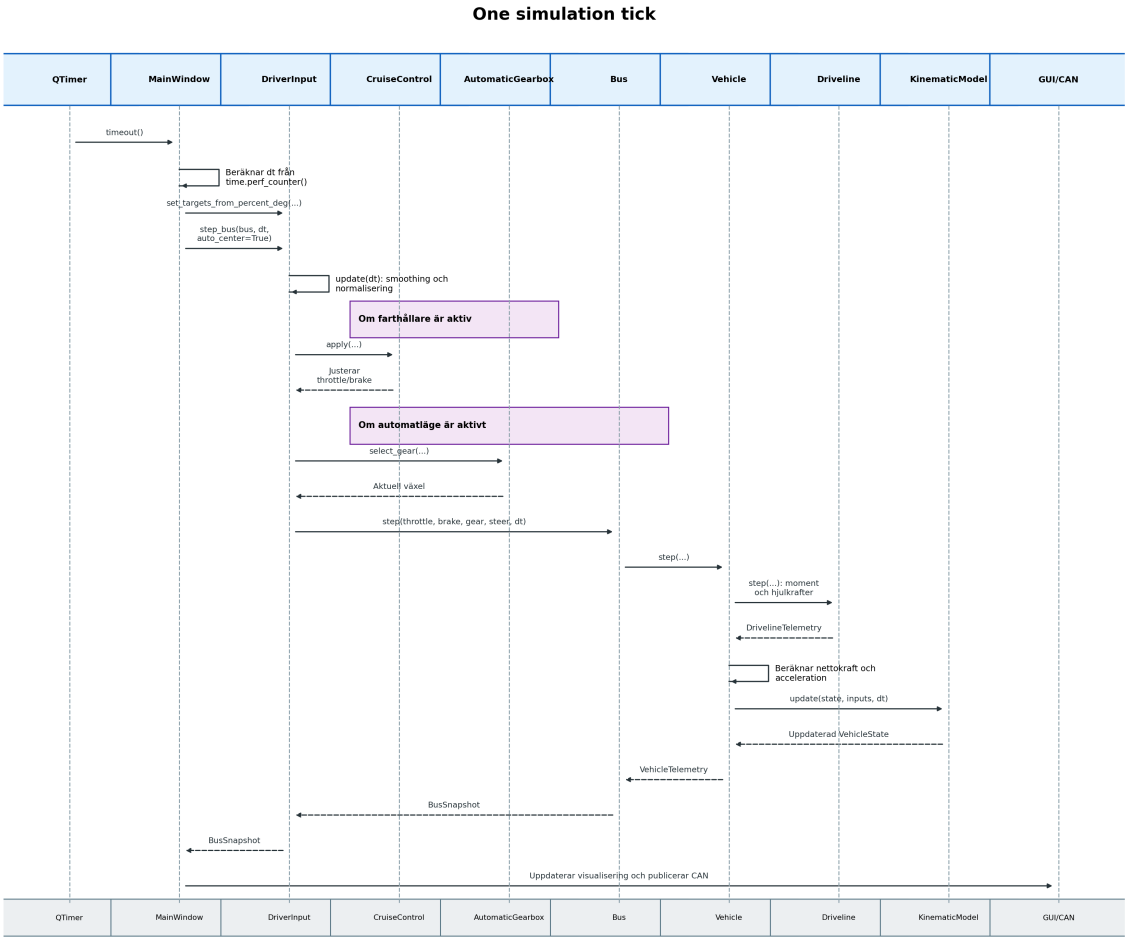


Figur 4.11: UML-klassdiagram över användargränssnittets struktur. Diagrammet visar hur `MainWindow` fungerar som central koordinator mellan GUI-komponenter, `DriverInput`, `Bus` och `GuiCanPublisher`.

Figur 4.11 visar att gränssnittet inte enbart är en visualiseringsyta, utan även den del som samordnar simuleringens körning. `QTimer` triggar återkommande simuleringssteg, `DriverInput` omvandlar användarens reglage till modellkommandon, `Bus` beräknar ett nytt `BusSnapshot`, och därefter uppdateras kartvy, grafer, statuspaneler och CAN-signaler. Denna struktur gjorde det möjligt att använda GUI:t både som styrpanel och som felsökningsverktyg.

Hela tidssteget från GUI-ingång till uppdaterad visualisering och CAN-publicering

sammanfattas i Figur 4.12. Figuren visar det viktigaste runtime-flödet i prototypen och förklarar hur de tre huvuddelarna, fordonsmodell, GUI och CAN-kommunikation, kopplas samman vid körning.



Figur 4.12: Sekvensdiagram över ett simuleringssteg. `MainWindow` triggas av `QTimer`, förarinput omvandlas till modellkommandon, `Bus` uppdaterar fordonsmodellen och resultatet används för visualisering och CAN-publicering.

Som framgår av Figur 4.12 är `BusSnapshot` den centrala datapunkten mellan modellen och övriga systemet. Snapshot-objektet innehåller de beräknade tillstånd som behövs för att uppdatera användargränssnittet och för att generera CAN-signaler. Detta minskar beroendet mellan komponenterna, eftersom CAN-modulen inte behöver anropa fordonsmodellen direkt utan endast konsumerar resultatet från det senaste simuleringssteget.

Gränssnittet användes även som stöd vid felsökning. Eftersom modellens respons kunde observeras direkt i kartvyn och i telemetrigraferna blev det enklare att upptäcka avvikelser i exempelvis rotationsriktning, koordinattransformationer eller hastighetsförändringar. På så sätt fungerade gränssnittet inte enbart som ett sätt att styra simuleringen, utan även som ett verktyg för validering av modellens beteende.

I den senare versionen kompletterades gränssnittet med funktioner kopplade till CAN-kommunikation. Användaren kunde där välja vilka signaler som skulle aktiveras för sändning mot testmiljön. Därmed blev användargränssnittet en länk mellan fordonsmodellens interna tillstånd och den externa hårdvaruintegrationen.

4.5 Implementation av CAN-kommunikation

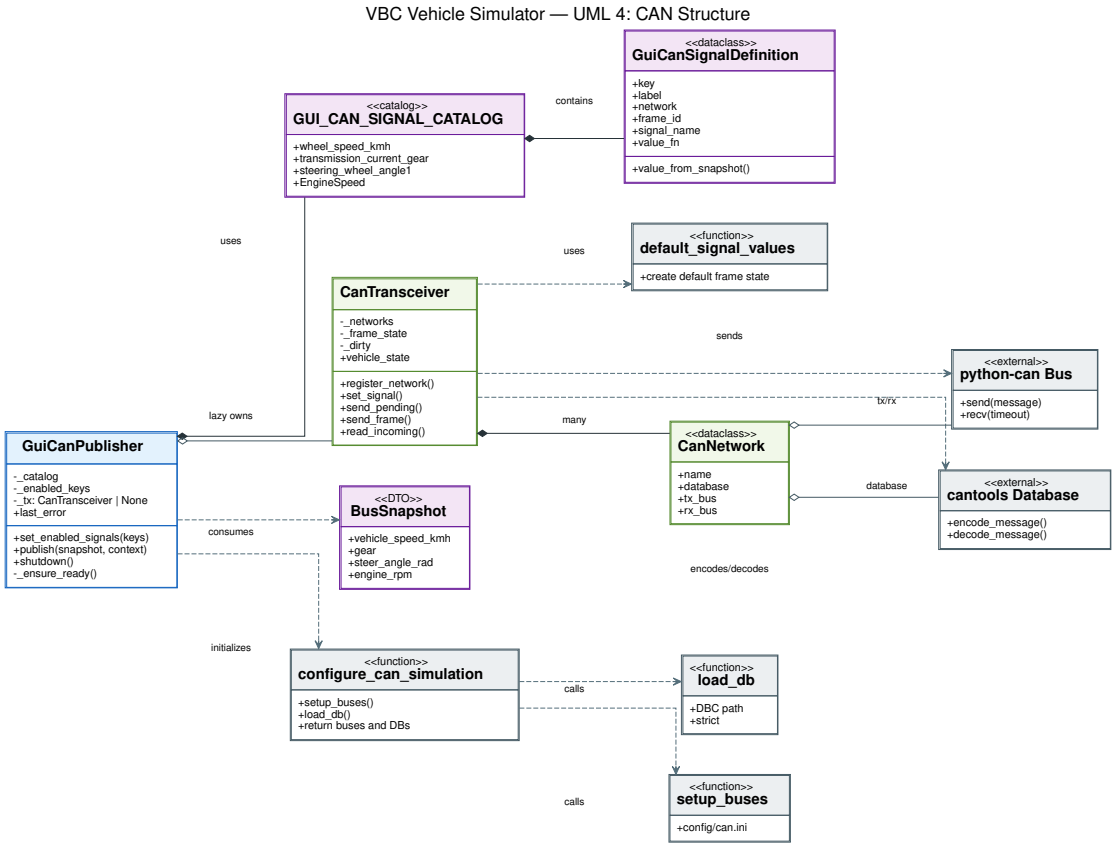
Efter att fordonsmodellen och användargränssnittet hade etablerats implementerades CAN-kommunikation. Syftet var att koppla modellens interna tillstånd till fordonsrelaterade signaler som kunde skickas från simuleringsverktyget till en extern testmiljö.

Kommunikationen implementerades med hjälp av biblioteket `python-can`, som användes för att konfigurera CAN-gränssnitt, skapa meddelanden och skicka signaler på CAN-nätverket. För att hantera signalernas struktur användes fördefinierade DBC-filer enligt beskrivningen i avsnitt 2.4.3. Varje signal i DBC-filen var definierad med bitposition, längd, skalning och offset. Detta gjorde det möjligt att koda modellens fysikaliska värden, exempelvis fordonshastighet, hjulhastighet eller styrvinkel, till rätt binärt format innan sändning på CAN-bussen.

Biblioteket `cantools` användes för att läsa in DBC-filerna samt koda och avkoda signaler enligt deras definitioner. Genom att använda DBC-filer kunde simuleringsverktyget generera signaler med rätt format, skalning och identifierare. Detta gjorde att verktyget inte enbart producerade interna simuleringsvärden, utan även kunde översätta dessa till ett format som är relevant för en fordonsnära testmiljö.

För att etablera kommunikation med Vector Hardware Manager konfigurerades ett CAN-gränssnitt i en initieringsfil. Konfigurationen omfattade bland annat val av interface, kanal, applikationsnamn och bitrate. Den använda konfigurationen visas i Appendix A.2, Listing A.1. Applikationsnamnet behövde överensstämma mellan simuleringsverktyget och Vector Hardware Manager för att anslutningen skulle kunna etableras korrekt. Bitraten anpassades till testmiljöns konfiguration, eftersom samtliga noder på en CAN-buss måste använda samma bitrate för att kunna tolka varandras meddelanden.

CAN-modulens struktur visas i Figur 4.13. Figuren visar hur `GuiCanPublisher` fungerar som adapter mellan modellens `BusSnapshot` och den mer generella CAN-transceivern. Denna uppdelning infördes för att hålla mappningen från simulerade värden till GUI-valda CAN-signaler separerad från den lägre nivån som hanterar DBC-kodning och fysisk sändning.



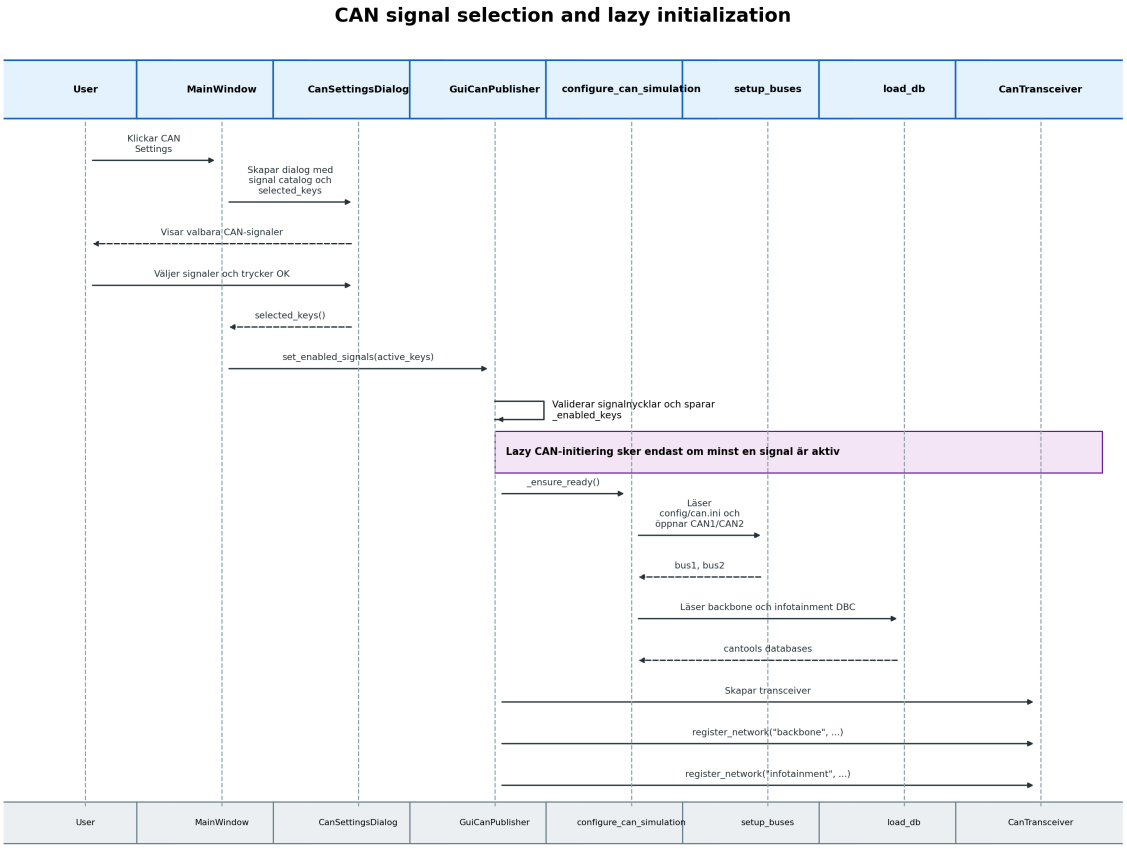
Figur 4.13: UML-klassdiagram över CAN-modulens struktur. `GuiCanPublisher` översätter värden från `BusSnapshot` till definierade GUI-CAN-signaler, medan `CanTransceiver` hanterar DBC-baserad kodning och sändning via `python-can`.

Figur 4.13 visar att CAN-kommunikationen delades upp i två nivåer. På den övre nivån finns signaldefinitioner som anger vilken modellvariabel som ska användas, vilket nätverk signalen tillhör, vilket meddelande som ska skickas och vilket DBC-signalnamn som ska uppdateras. På den lägre nivån hanterar `CanTransceiver` frame state, dirty frames, DBC-kodning och sändning på CAN-bussen. Denna uppdelning gjorde det enklare att lägga till nya signaler utan att ändra den generella kommunikationslogiken.

CAN-modulen byggdes för att kunna hantera flera signaler på ett skalbart sätt. Vid uppstart kunde relevanta DBC-filer läsas in, varefter signaldefinitioner kunde användas för att koppla modellens interna variabler till motsvarande CAN-signaler. Nya signaler kunde därmed läggas till utan att den grundläggande kommunikationsstrukturen behövde förändras.

Eftersom flera signaler kan tillhöra samma CAN-meddelande utvecklades funktionalitet för att samla signaler per meddelande. Vid varje uppdatering kunde de signaler som hörde till samma meddelande grupperas, kodas tillsammans och skickas som ett komplett CAN-meddelande. Detta var nödvändigt för att signalerna skulle följa den struktur som definierades i DBC-filerna.

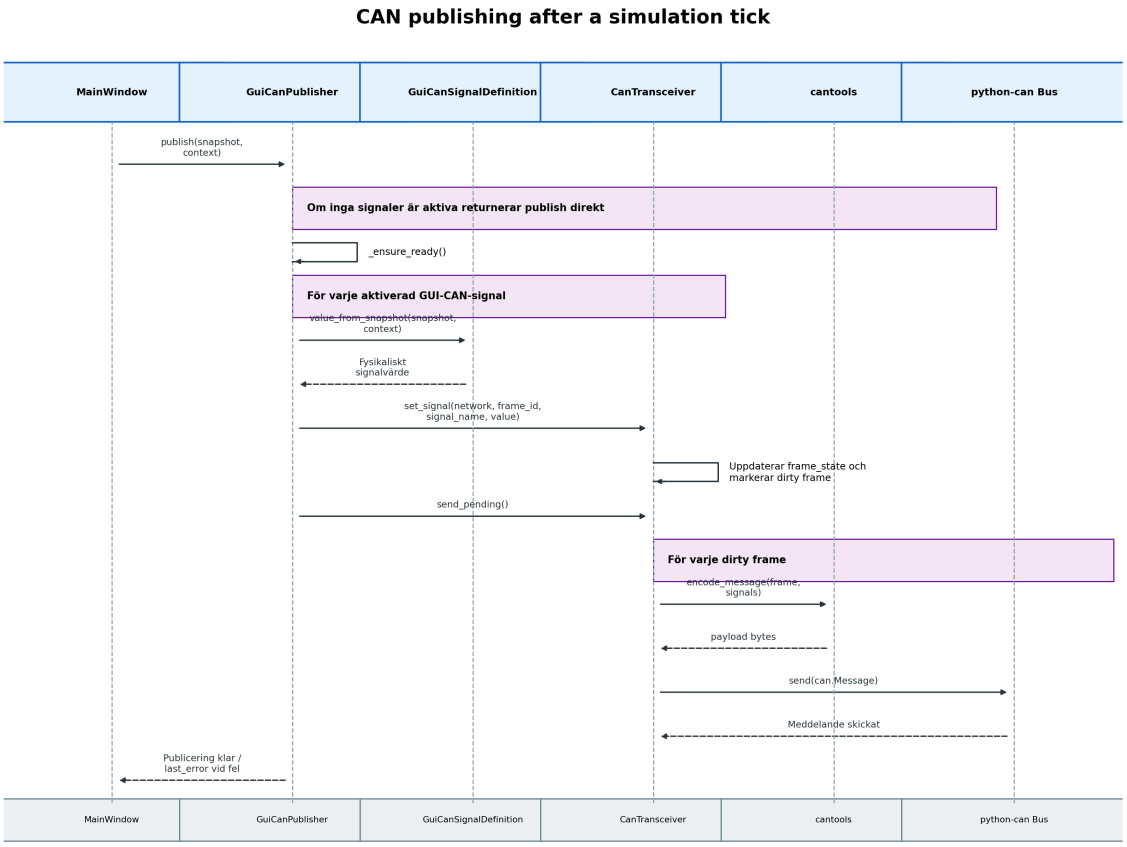
Eftersom CAN-kommunikationen inte behöver startas förrän användaren faktiskt har valt signaler, implementerades initieringen som en fördröjd initiering. Detta flöde visas i Figur 4.14. Där framgår hur användaren väljer signaler i `CanSettingsDialog`, varefter `GuiCanPublisher` initierar CAN-bussar, läser in DBC-filer och registrerar nätverk i `CanTransceiver`.



Figur 4.14: Sekvensdiagram över val av CAN-signaler och fördröjd initiering av CAN-kommunikation. CAN-konfigurationen laddas först när minst en signal har aktiverats av användaren.

Figur 4.14 visar hur initieringen av CAN-kommunikationen hålls separerad från programmets uppstart. Detta minskade risken för att simulatorn skulle sluta fungera om CAN-hårdvara saknades eller om DBC-filer inte kunde läsas in. Användaren kunde därmed fortfarande använda modellen och gränssnittet utan aktiv CAN-kommunikation, medan CAN-delen kunde startas först vid behov.

När CAN-modulen är initierad publiceras valda signaler efter varje simuleringssteg. Detta visas i Figur 4.15. Figuren visar hur `GuiCanPublisher` hämtar värden från `BusSnapshot`, låter varje `GuiCanSignalDefinition` beräkna sitt signalvärde och därefter uppdaterar rätt frame i `CanTransceiver`.



Figur 4.15: Sekvensdiagram över CAN-publicering efter ett simuleringssteg. Aktiva signaler hämtas från `BusSnapshot`, kodas med `cantools` enligt DBC-definitioner och skickas via `python-can`.

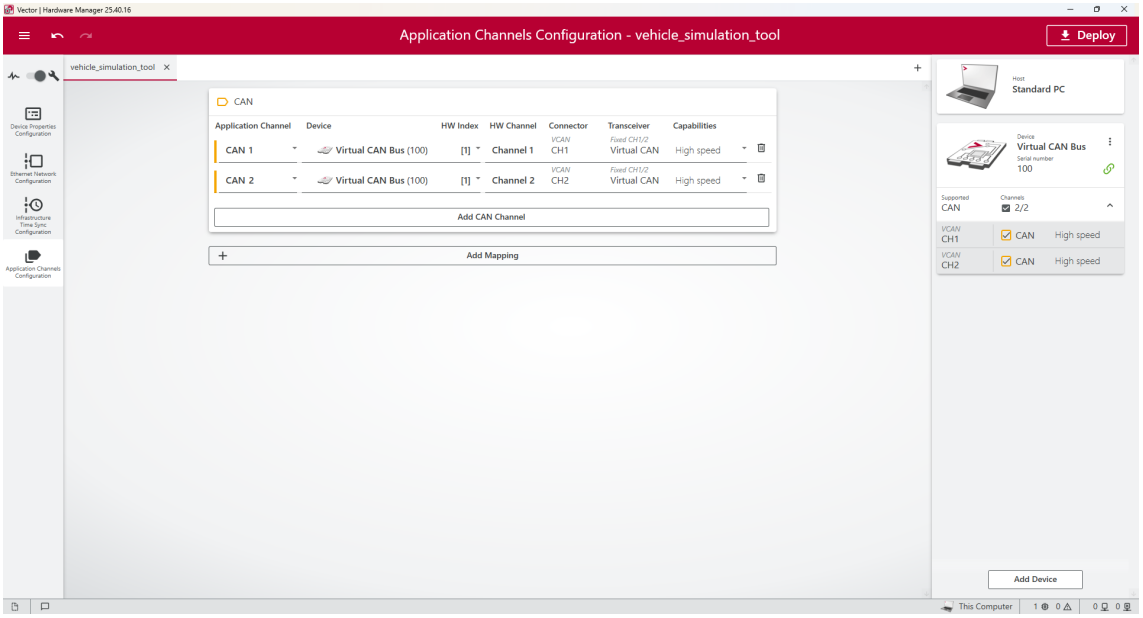
Av Figur 4.15 framgår att flera signaler kan uppdatera samma CAN-meddelande innan sändning. `CanTransceiver` håller därför ett internt frame state och markerar endast ändrade frames som dirty. När `send_pending()` anropas kodas och skickas de frames som faktiskt har uppdaterats. Detta gör signalhanteringen mer skalbar än om varje signal hade skickats som ett separat meddelande.

För att förenkla kopplingen mellan modell och CAN-kommunikation utvecklades även en separat mappningsstruktur. I denna kunde en signal kopplas till en specifik tillståndsvariabel i modellen, en vald DBC-databas, ett meddelande och ett signalnamn. Strukturen fungerade därmed som en brygga mellan fordonsmodellens interna tillstånd och de externa CAN-signalerna.

4.6 Anslutning och testning mot HIL-rigg

Innan verktyget anslöts till den fysiska testmiljön genomfördes testning i en virtuell CAN-miljö. En virtuell CAN-lina sattes upp i Vector Hardware Manager, vilket gjorde det möjligt att testa sändning och mottagning utan att koppla in fysisk hårdvara. Detta minskade risken för fel under integrationsarbetet och gav en kontrollerad miljö för felsökning.

Den virtuella uppkopplingen i Vector Hardware Manager visas i Figur 4.16. Figuren visar den testmiljö som användes för att verifiera att simuleringsverktyget kunde ansluta till en virtuell CAN-buss innan fysisk hårdvara användes.



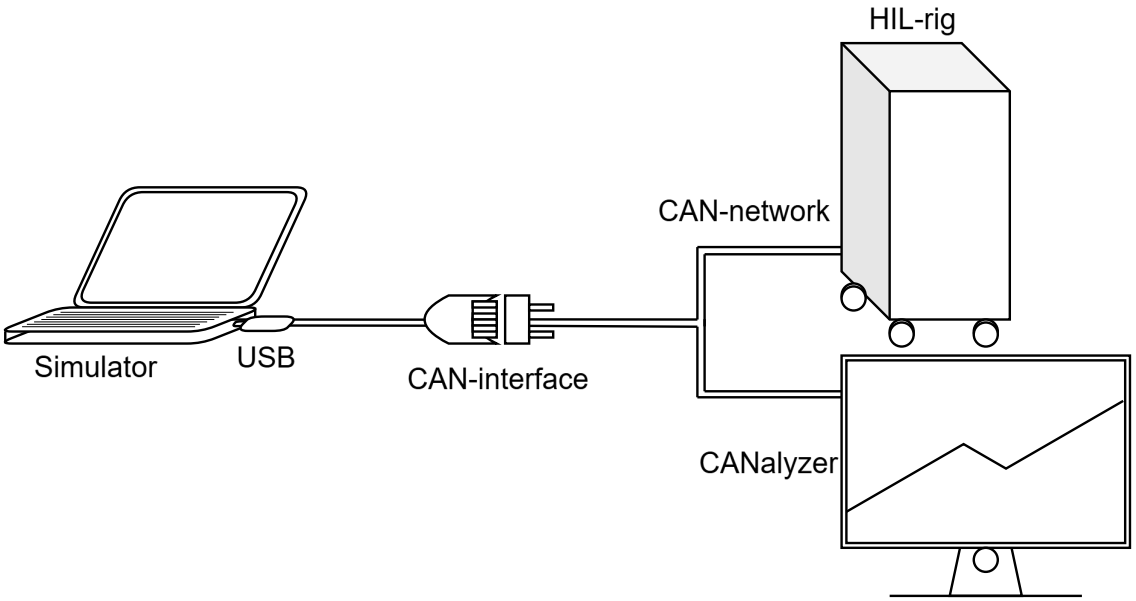
Figur 4.16: Vector Hardware Manager med en virtuell CAN-buss uppkopplad för testning av simuleringsverktygets CAN-kommunikation.

Figur 4.16 visar den virtuella CAN-konfigurationen som användes i ett tidigt integrationssteg. Genom att först arbeta mot en virtuell databuss kunde grundläggande problem med kanalval, applikationsnamn, bitrate och drivrutinsanslutning hanteras innan systemet kopplades till HIL-riggen.

Först kontrollerades att CAN-gränssnittet kunde initieras och att meddelanden kunde skickas utan kommunikationsfel. Denna kontroll byggde på återkoppling från `python-can`, exempelvis fel vid felaktig kanal, fel bitrate, problem med drivrutin eller misslyckad sändning. Detta verifierade att kommunikationen kunde upprättas, men inte att signalernas innehåll var korrekt.

Vid den virtuella testningen användes de två kanalerna `CAN1` och `CAN2` från konfigurationen i Appendix A.2, vilket gjorde det möjligt att skicka signaler på en kanal och ta emot dem på den andra. Detta gjorde det möjligt att verifiera signalernas skalning, kodning och innehåll innan anslutning till fysisk testmiljö.

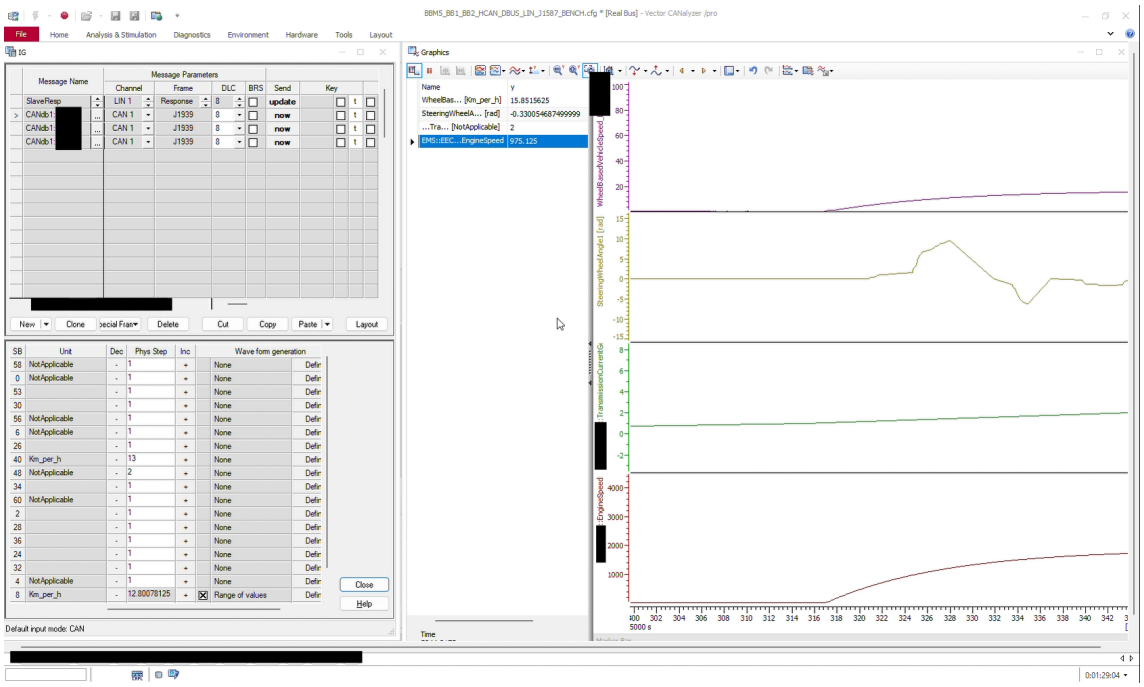
Efter den virtuella testningen konfigurerades Vector Hardware Manager om för kommunikation mot ett fysiskt CAN-case anslutet via USB. Simuleringsverktygets CAN-konfiguration anpassades därefter till testmiljöns förutsättningar, bland annat genom korrekt kanalval, applikationsnamn och bitrate. Figur 4.17 illustrerar den fysiska uppkopplingen mot HIL-riggen.



Figur 4.17: Fysisk uppkoppling av simuleringsverktyget mot HIL-riggen via CAN-gränssnitt och USB.

Figur 4.17 visar den fysiska integrationskedjan från simuleringsverktyget till testmiljön. Datorn som kör simulatorn ansluts via CAN-gränssnitt till riggens CAN-nätverk, där de simulerade signalerna kan tas emot av den ECU eller testdator som ingår i HIL-miljön. Figuren tydliggör därmed hur mjukvarumodellen kopplas till en fysisk verifieringsmiljö.

I den första fysiska testningen skickades en begränsad uppsättning signaler från simuleringsverktyget till CAN-nätverket. Signalerna övervakades med CANalyzer på en separat dator. Syftet var att kontrollera att signalerna inte enbart skickades från programmet, utan även kunde observeras i den fysiska testmiljön med förväntad struktur och rimliga värden. Figur 4.18 visar ett exempel på hur mottagna signaler kunde granskas i CANalyzer.



Figur 4.18: Exempel på mottagna CAN-signaler i CANalyzer vid testning av signalöverföring från simuleringsverktyget.

Figur 4.18 visar hur de skickade signalerna kunde verifieras i CANalyzer. Detta var ett viktigt teststeg eftersom det bekräftade att signalerna kunde observeras utanför simuleringsverktyget och att de därmed hade lämnat den interna mjukvarumiljön. Genom att jämföra mottagna värden med modellens genererade värden kunde även signalernas skalning och kodning kontrolleras.

Vid fysisk inkoppling anslöts verktyget mot den ECU som skulle ta emot signalerna. Den fysiska HIL-riggen som användes vid integrationen visas i Appendix A.1, Figur A.1. Eftersom vissa signaler normalt skickades av andra noder i testmiljön behövde motsvarande tidigare signalkälla kopplas bort under testningen. Detta gjordes för att undvika att flera noder skickade samma signal samtidigt på CAN-nätverket.

Efter att den första signalöverföringen hade fungerat utökades hanteringen till fler signaler. Exempel på signaler som kunde kopplas från modellen till CAN-kommunikation var hjulhastigheter, styrvinkel, motorvarvtal och växelläge. Därmed kunde verktyget användas som en simulerad signalkälla i en kontrollerad testmiljö.

En praktisk begränsning under integrationen var att det CAN-case som användes i testuppställningen endast kunde anslutas till en CAN-lina åt gången. Det innebar att signaler kopplade till samma databas kunde testas samtidigt, medan signaler från andra delar av riggens CAN-nätverk krävde fysisk omkoppling. Denna begränsning påverkade främst testförfarandet och inte den grundläggande mjukvarustrukturen.

Genom denna stegvisa anslutning kunde simuleringsverktyget gå från intern modelltestning, via virtuell CAN-kommunikation, till fysisk signalöverföring mot testrigg. Därmed verifierades hela kedjan från simulerade fordonsbeteenden till observerbara CAN-signaler i testmiljön.

5

Resultat

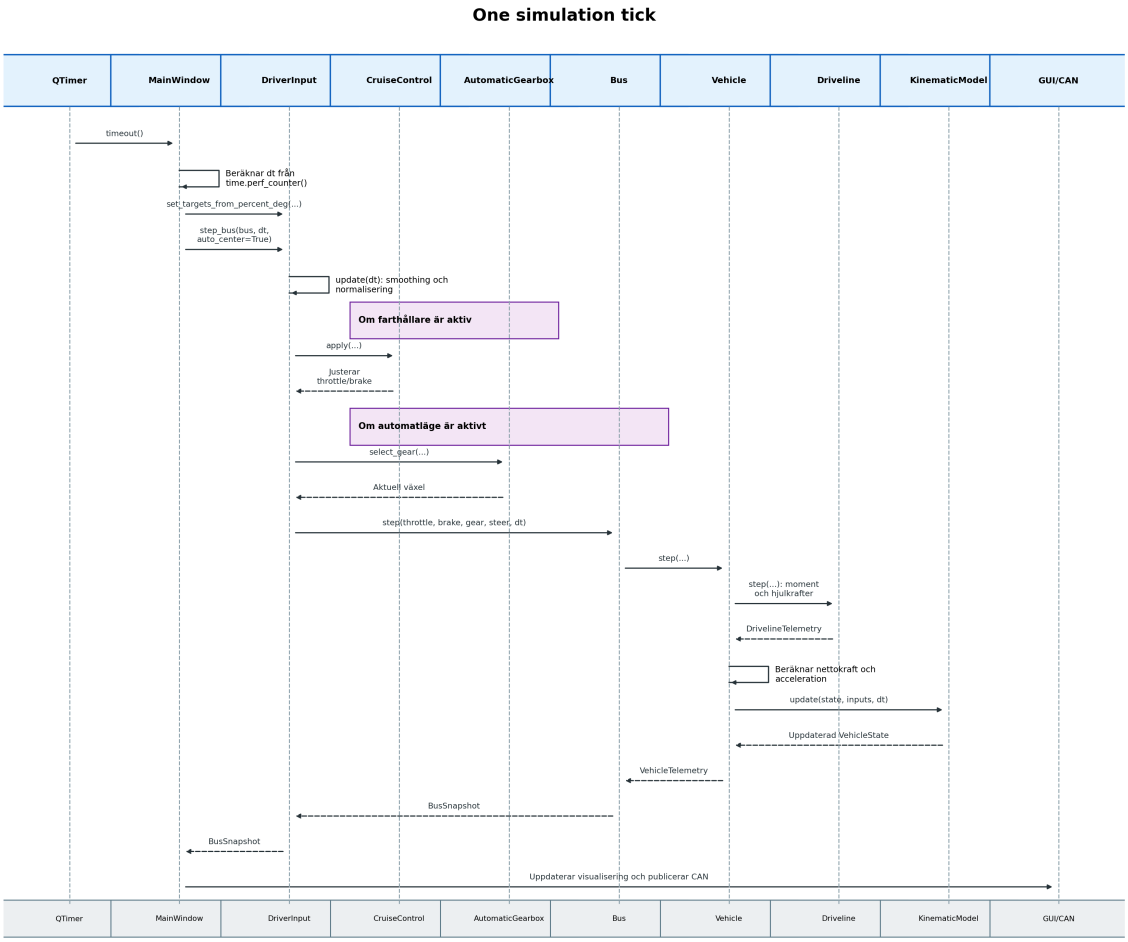
Detta kapitel presenterar projektets resultat i relation till syfte och frågeställningar. Fokus ligger på vad den utvecklade prototypen möjliggör, hur fordonsmodellen fungerar, hur simulerade signaler kopplas till CAN-meddelanden samt vilka möjligheter och begränsningar som identifierades vid användning av verktyget för tidig verifiering.

5.1 Resultat i relation till huvudfrågan

Projektets huvudfråga var hur ett CAN-baserat simuleringsverktyg kan utvecklas och utvärderas för att möjliggöra tidig verifiering av fordonsbeteenden i en labb-baserad testmiljö. Resultatet visar att detta kan genomföras genom att kombinera tre centrala delar: en mjukvarubaserad fordonsmodell, ett grafiskt användargränssnitt och en modul för CAN-kommunikation.

Den utvecklade prototypen kan simulera enklare körbeteenden i realtid, visualisera fordonets tillstånd och överföra utvalda simulerade signaler via CAN. Fordonsmodellen genererar tillstånd såsom hastighet, acceleration, orientering, växel, motorvarvtal och hjulrelaterade variabler. Det grafiska användargränssnittet gör det möjligt att styra och övervaka simuleringen, medan CAN-modulen kodar valda signaler enligt DBC-definitioner och skickar dem till en virtuell eller fysisk testmiljö.

Det färdiga systemets huvudflöde sammanfattas i Figur 5.1. Figuren visar hur prototypen binder ihop användarinput, fordonsmodell, visualisering och CAN-publicering. Därmed illustrerar den projektets huvudsakliga resultat: ett GUI-lett simuleringsverktyg där modellens tillstånd kan användas både för övervakning och extern signalgenerering.



Figur 5.1: Sammanfattande sekvensdiagram över prototypens huvudflöde. Figuren visar hur ett simuleringssteg leder till uppdaterad fordonsmodell, uppdaterat GUI och valfri CAN-publicering.

Figur 5.1 visar att prototypen uppfyller den centrala systemidén: användaren kan styra ett simulerat fordon i realtid, modellen beräknar nya tillstånd och samma tillstånd kan användas för både visualisering och CAN-kommunikation. Detta innebär att verktyget kan fungera som en sammanhållen prototyp för tidig verifiering, även om modellens fysikaliska noggrannhet och antalet implementerade CAN-signaler fortfarande är begränsade.

Tabell 5.1 sammanfattar resultatet i relation till projektets delfrågor. Tabellen visar även de viktigaste begränsningarna, vilket tydliggör att resultatet bör tolkas som ett funktionellt principbevis snarare än som ett färdigt industriellt verifieringsverktyg.

Tabell 5.1: Sammanfattning av projektets delfrågor, huvudsakliga resultat och identifierade begränsningar.

Delfråga	Huvudsakligt resultat	Begränsning
Fordonsmodell	En komponentbaserad och objektorienterad fordonsmodell utvecklades. Modellen kan simulera centrala körbeteenden såsom acceleration, inbromsning, styrning, lutning och hastighetsreglering.	Modellen är förenklad och inte kalibrerad mot mätdata från ett verkligt fordon.
CAN-koppling	Simulerade tillstånd kunde kopplas till DBC-definierade signaler, kodas till CAN-meddelanden och observeras i virtuell och fysisk testmiljö.	Endast ett begränsat urval av signaler och CAN-flöden implementerades.
Tidig verifiering	Verktyget visar potential att stödja tidig funktionell verifiering, felsökning och systemintegration i labbmiljö.	Prototypen saknar fullständigt stöd för automatiserade, repeterbara och industriellt robusta testscenarier.

Sammanfattningsvis visar resultatet att simulerade fordonsbeteenden kan användas som underlag för verifiering utan att ett komplett fysiskt fordon behöver användas. Prototypen kan därmed fungera som en mjukvarubaserad signalkälla i en HIL-miljö och användas för tidig verifiering av enklare fordonsrelaterade funktioner. Samtidigt bör resultatet ses som ett principbevis, där vidareutveckling krävs för mer realistisk fordonsdynamik, fler signaler och mer omfattande integration mot testmiljön.

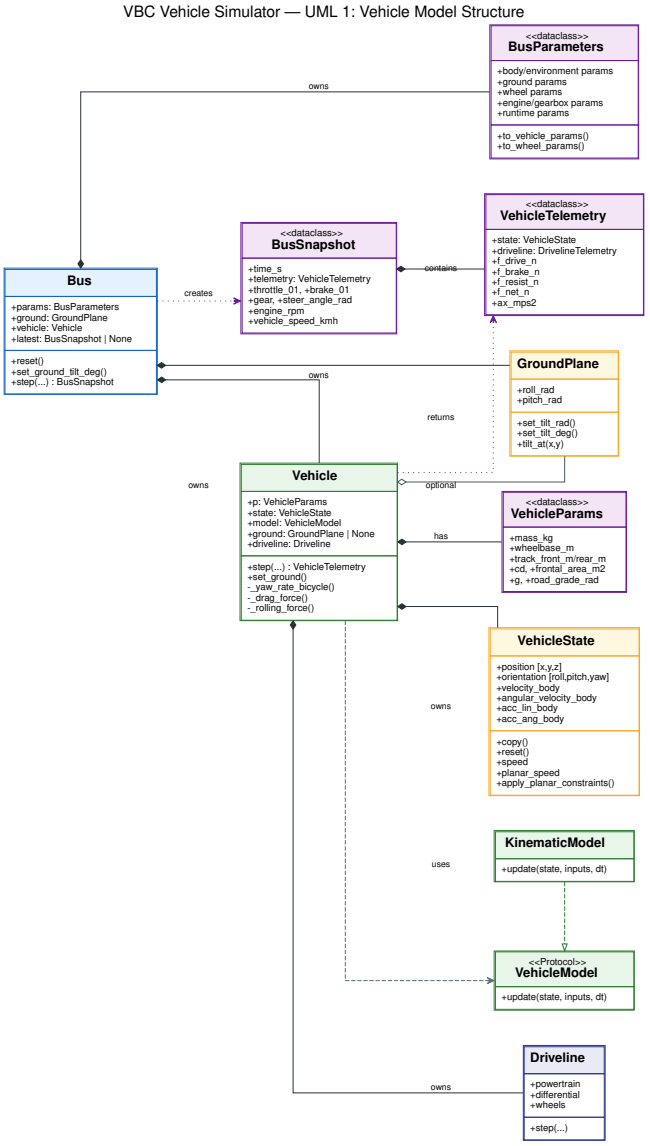
5.2 Förenklad och utbyggbar fordonsmodell

Den första delfrågan behandlade hur en förenklad men utbyggbar fordonsmodell kan utformas för att simulera centrala körbeteenden i realtid. Resultatet visar att detta kan uppnås genom en komponentbaserad och objektorienterad modellstruktur, där fordonets olika delar representeras av separata moduler.

En styrka med den valda modellstrukturen är att den möjliggör en tydlig uppdelning mellan fordonets tillstånd, drivlina, reglering och visualisering. Genom att samla centrala variabler i `VehicleState` kan samma data användas av flera delar av systemet. Detta gör modellen mer överskådlig och minskar risken för inkonsekvens mellan beräkningar, visualisering och signalgenerering.

Den objektorienterade modellstrukturen i Figur 5.2 sammanfattar en av projektets

viktigaste resultatdelar. Genom att separera bussens övergripande simuleringsgränssnitt, fordonets dynamiska tillstånd och den kinematiska uppdateringsmodellen kunde verktyget byggas ut stegvis utan att hela kodbasen behövde skrivas om.



Figur 5.2: Resultat av den objektorienterade modellstrukturen. Figuren visar hur fordonsmodellen organiserades kring `Bus`, `Vehicle`, `VehicleState` och `KinematicModel`.

Figur 5.2 visar att modellen fick en tydlig ansvarsfördelning. `Bus` fungerar som yttre gränssnitt, `Vehicle` ansvarar för fordonsdynamiska beräkningar, `VehicleState` samlar tillståndet och `KinematicModel` utför den kinematiska integrationen. Detta resultat är viktigt eftersom modellen inte är en monolitisk beräkningsloop, utan en struktur som kan vidareutvecklas med nya komponenter, mer avancerade delmodeller eller fler signalutgångar.

Den komponentbaserade strukturen gör även modellen utbyggbar. Eftersom motor,

växellåda, drivaxel, differential och hjul hanteras som separata delar kan enskilda komponenter förbättras utan att hela systemet behöver omarbetas. Detta är viktigt eftersom projektet inte syftade till att skapa en fullständig fysikalisk fordonsmodell, utan ett funktionellt simuleringsverktyg som kan vidareutvecklas över tid.

Den utvecklade modellen kan simulera centrala körbeteenden såsom acceleration, inbromsning, styrning, lutning och hastighetsreglering. Genom farthållare och automatisk växling kan modellen även skapa mer kontrollerade körförlopp än vid enbart manuell styrning. Detta gör modellen användbar för funktionella tester där syftet är att generera rimliga fordonsrelaterade signaler snarare än att exakt återskapa ett verkligt fordons dynamik.

Samtidigt finns begränsningar i modellens realism. Exempelvis saknas mer avancerad hantering av motorbromsning, tomgångsvarvtal, motorstopp, däckdynamik, koppling och detaljerade växlingsförlopp. Resultatet visar därmed att modellen är lämplig för funktionella tester och tidig verifiering av enklare fordonsbeteenden, men inte för tester där hög fysikalisk precision krävs. För sådana tillämpningar skulle modellen behöva kalibreras mot mätdata och kompletteras med mer detaljerade delmodeller.

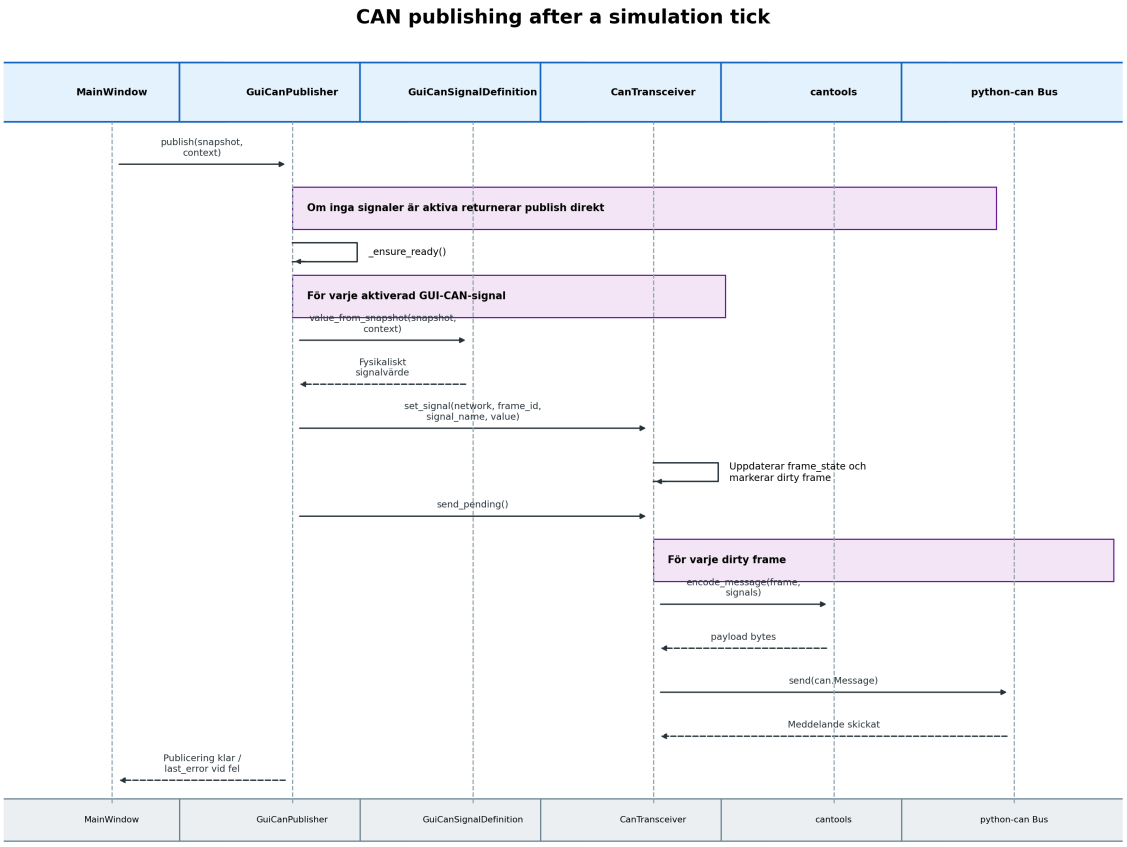
5.3 Koppling mellan simulerade signaler och CAN-meddelanden

Den andra delfrågan behandlade hur simulerade fordonsrelaterade signaler kan kopplas till CAN-meddelanden och överföras till en befintlig HIL-miljö. Resultatet visar att detta kan genomföras genom att koppla modellens interna tillstånd till signaldefinitioner i DBC-filer och därefter koda dessa signaler till CAN-meddelanden.

En viktig del av lösningen är användningen av DBC-filer. Genom att använda befintliga signaldefinitioner kan simuleringsverktyget generera signaler med rätt format, skalning och identifierare. Detta gör att verktyget inte enbart producerar interna simuleringsvärden, utan även översätter dessa till ett format som är relevant för en fordonsnära testmiljö.

De signaler som implementerades och testades omfattade bland annat fordonshastighet, aktuell växel, styrvinkel och motorvarvtal. Dessa signaler valdes eftersom de är centrala för att beskriva fordonets rörelse och samtidigt kan kopplas direkt till modellens beräknade tillstånd. Därmed var de lämpliga för att verifiera kedjan från simulering till CAN-kommunikation.

Resultatet av CAN-integrationen kan förstås genom publiceringsflödet i Figur 5.3. Figuren visar hur ett simulerat fordonsvärde omvandlas till en DBC-definierad CAN-signal och skickas via den konfigurerade CAN-bussen.



Figur 5.3: Resultatflöde för CAN-publicering. Figuren visar kopplingen från simulerat tillstånd i `BusSnapshot` till DBC-kodad CAN-frame och fysisk eller virtuell sändning via `python-can`.

Figur 5.3 visar att CAN-integrationen inte enbart består av att skicka råa värden, utan av en kedja där signalvärden hämtas från modellen, mappas mot DBC-signaler, kodas till rätt frame och därefter skickas på databussen. Detta är viktigt för resultatet eftersom det innebär att simuleringsverktyget kan agera som en mer realistisk signalkälla i en HIL-miljö än om värden hade skickats utan DBC-baserad struktur.

CAN-integrationen verifierades först i en virtuell CAN-miljö. Genom att skicka signaler på en kanal och ta emot dem på en annan kunde sändning, mottagning och signalinnehåll kontrolleras utan att fysisk hårdvara behövde anslutas. Detta var ett viktigt steg eftersom fel i kanalval, bitrate, DBC-kodning eller signalmappning kunde identifieras innan verktyget kopplades till testriggen.

När signalerna därefter skickades till den fysiska testmiljön kunde de observeras i CANalyzer. Detta visar att simuleringsverktyget kan fungera som en mjukvarubaserad signalkälla mot en HIL-miljö. Därmed verifierades inte enbart den interna modellen, utan även den externa kommunikationskedjan från simulerat tillstånd till mottagen CAN-signal.

Samtidigt är den nuvarande CAN-integrationen begränsad. Endast ett urval av signaler implementerades, vilket innebär att verktyget ännu inte representerar ett

komplett fordonsnätverk. Vid mer omfattande testning skulle fler signaler behöva inkluderas, och relationen mellan olika signaler skulle behöva hanteras mer systematiskt. Exempelvis kan vissa ECU:er förvänta sig flera samtidiga signaler med specifika uppdateringsfrekvenser och inbördes beroenden.

5.4 Möjligheter med simuleringsbaserad verifiering

Den tredje delfrågan behandlade vilka möjligheter som uppstår när ett simuleringsbaserat verktyg används för tidigare verifiering av fordonsmjukvara. Resultatet visar att verktyget kan bidra till tidigare testning av funktioner som annars hade krävt tillgång till ett fysiskt fordon eller en mer omfattande testmiljö.

Genom att simulera fordonsbeteenden och generera motsvarande CAN-signaler kan utvecklare och testare skapa kontrollerade körfall i labbmiljö. Detta kan bidra till att fel upptäcks tidigare i utvecklingsprocessen, vilket är centralt i ett *shift-left*-perspektiv. Verktyget kan även göra testningen mer flexibel, eftersom körscenarier kan ändras i mjukvara utan att fysiska förutsättningar behöver anpassas.

En annan möjlighet är ökad repeterbarhet. Fysiska tester påverkas ofta av yttre faktorer såsom vägförhållanden, förare, väder och tillgång till testfordon. Ett simuleringsbaserat verktyg kan däremot ge mer kontrollerade förutsättningar, särskilt om fördefinierade testscenarier implementeras. Detta gör det möjligt att köra samma scenario flera gånger och jämföra resultat mellan olika versioner av mjukvara.

Verktyget kan också minska tröskeln för tidig systemintegration. Genom att koppla simulerade signaler till en HIL-miljö kan vissa funktioner testas innan alla fysiska komponenter är tillgängliga. Detta kan vara värdefullt i fordonsutveckling, där hårdvara, mjukvara och systemintegration ofta utvecklas parallellt.

Ytterligare en möjlighet är att simuleringsverktyget kan användas som stöd vid felsökning. Eftersom både modellens interna tillstånd och de skickade CAN-signalerna kan observeras blir det enklare att avgöra om ett problem uppstår i fordonsmodellen, i signalmappningen, i DBC-kodningen eller i den externa testmiljön. Denna transparens är särskilt värdefull i tidiga utvecklingsfaser, där fel ofta kan bero på flera samverkande delar.

5.5 Identifierade begränsningar

Trots dessa möjligheter finns flera begränsningar. Den mest centrala är att en simuleringsmodell alltid är en förenkling av verkligheten. I detta projekt har modellen avgränsats till att återskapa realistiska beteenden på systemnivå snarare än att ge en exakt fysikalisk representation. Detta innebär att verktyget lämpar sig bäst för tidiga och funktionella tester, men inte för verifiering där mycket hög fysisk noggrannhet krävs.

En annan begränsning är att modellen inte har kalibrerats mot mätdata från ett verkligt fordon. Därmed går det inte att säkerställa att modellens respons exakt motsvarar

ett fysiskt fordons beteende. Resultaten bör därför tolkas som en demonstration av metod och systemarkitektur snarare än som validerad fordonsdynamik.

Vidare är den nuvarande prototypen begränsad när det gäller fördefinierade testscenarier. Även om realtidsstyrning fungerar väl för explorativ testning är den mindre lämplig för systematisk verifiering, eftersom manuell inmatning gör det svårt att upprepa exakt samma körförlopp. För att stärka verktygets verifieringsvärde bör framtida versioner innehålla stöd för scriptade och repeterbara scenarier.

En begränsning som framgår av systemarkitekturen är att GUI:t i den nuvarande implementationen är den centrala koordinatorn för simuleringen. Detta har varit ändamålsenligt för en prototyp, eftersom det gör verktyget enkelt att styra, visualisera och felsöka i realtid. Samtidigt innebär det att simuleringen i nuläget är starkt kopplad till den interaktiva körningen i gränssnittet.

Detta påverkar framför allt möjligheten till automatiserad och repeterbar verifiering. Även om användaren kan skapa realistiska körförlopp manuellt är manuella körningar svåra att upprepa exakt. För framtida utveckling bör därför simuleringsloopen kunna köras även utan GUI, exempelvis genom scriptade scenarier där gas, broms, styrning, växel och marklutning definieras i förväg. En sådan vidareutveckling skulle göra verktyget mer lämpligt för systematiska regressionstester och automatiserad HIL-verifiering.

CAN-integrationen är också begränsad till ett urval av signaler. I en verklig fordonsmiljö är signaler ofta beroende av varandra, och ECU:er kan kräva ett större sammanhang för att reagera på ett realistiskt sätt. Därför behöver fler signaler, fler CAN-linor och mer avancerad signalhantering implementeras innan verktyget kan användas för mer omfattande systemtester.

En praktisk begränsning är även att den fysiska testuppställningen krävde korrekt konfiguration av CAN-gränssnitt, kanal, bitrate, DBC-filer och aktiv signalkälla. Om flera noder skickar samma signal samtidigt kan konflikter uppstå på nätverket. Detta innebär att användning av verktyget i en fysisk HIL-miljö kräver noggrann kontroll av vilka signaler som simuleras och vilka noder som är aktiva.

Sammanfattningsvis visar resultatet att den utvecklade prototypen kan användas som ett principbevis för CAN-baserad simuleringsdriven verifiering i labbmiljö. Verktyget demonstrerar hela kedjan från simulerat fordonsbeteende till observerbar CAN-signal, men behöver vidareutvecklas med mer detaljerad fordonsdynamik, fler signaler, scriptade testscenarier och mer omfattande HIL-integration för att kunna användas i större industriell skala.

6

Diskussion

Detta kapitel diskuterar projektets metod, tillförlitlighet och möjliga vidareutveckling. Eftersom projektets huvudsakliga resultat presenteras i kapitel 5 fokuserar diskussionen här på hur resultaten bör tolkas, vilka metodmässiga begränsningar som finns och vilka utvecklingsområden som är mest relevanta för framtida arbete.

6.1 Kritisk diskussion

Arbetet ska ses mot bakgrund av den utveckling som beskrivs i inledningen, där fordonsindustrin i allt högre grad präglas av kortare utvecklingscykler, ökad mjukvarukomplexitet och ett växande behov av tidigare verifiering. I detta sammanhang är simuleringsbaserad testning relevant eftersom den kan minska beroendet av fysiska testfordon och göra det möjligt att undersöka fordonsbeteenden redan i labbmiljö. Projektets resultat visar att det är möjligt att utveckla ett CAN-baserat simuleringsverktyg som kopplar samman en mjukvarubaserad fordonsmodell, ett grafiskt användargränssnitt och CAN-kommunikation mot en testmiljö. Därmed bidrar arbetet med ett konkret exempel på hur ett shift-left-orienterat arbetssätt kan stödjas i praktiken.

En tydlig styrka med arbetet är att hela kedjan från simulerat fordonsbeteende till observerbar CAN-signal har demonstrerats. Verktyget stannar inte vid en fristående fordonsmodell, utan visar hur modellens tillstånd kan användas för att generera signaler som kodas enligt DBC-definitioner och skickas via CAN till en extern miljö. Detta är centralt eftersom nyttan i en industriell testmiljö inte enbart ligger i att kunna simulera rörelse, utan i att kunna omvandla simulerade tillstånd till signaler som kan tolkas av befintliga testverktyg och system. Att signalerna kunde observeras i CANalyzer visar därför att prototypen uppfyller sin roll som funktionellt principbevis.

En annan styrka är den modulära och objektorienterade programstrukturen. Genom att separera fordonsmodell, GUI och CAN-kommunikation blev systemet mer överskådligt och lättare att vidareutveckla. Fordonsmodellen kan generera tillstånd utan att direkt känna till hur dessa visualiseras eller skickas via CAN, medan CAN-modulen kan arbeta utifrån `BusSnapshot` och signaldefinitioner. Denna uppdelning är värdefull även i andra projekt, eftersom den minskar kopplingen mellan modellering, användarinteraktion och kommunikation. På så sätt blir systemet mer flexibelt om exempelvis fler signaler, en annan fordonsmodell eller ett annat gränssnitt ska införas.

Samtidigt finns viktiga begränsningar som påverkar hur resultaten bör tolkas. Fordonsmodellen är avsiktligt förenklad och syftar till att återskapa realistiska beteenden på systemnivå snarare än att ge en exakt fysikalisk representation av ett verkligt fordon. Detta innebär att verktyget är lämpligt för tidiga funktionella tester, demon-

strationsfall och grundläggande verifiering av signalkedjor, men inte för verifiering där hög fordonsdynamisk precision krävs. Eftersom modellen inte har kalibrerats mot mätdata från ett verkligt fordon går det inte att dra slutsatsen att dess respons motsvarar ett verkligt fordons beteende under alla körfall.

En ytterligare begränsning är att den nuvarande prototypen främst är GUI-ledd och bygger på manuell styrning i realtid. Detta gör verktyget användbart för interaktiv testning, demonstration och felsökning, men begränsar möjligheten till strikt repeterbara tester. I en verifieringsmiljö är repeterbarhet ofta avgörande, eftersom samma testfall behöver kunna köras flera gånger för att jämföra olika mjukvaruversioner eller analysera förändringar. För att stärka verktygets värde som verifieringsplattform skulle framtida versioner därför behöva stöd för scriptade testscenarier, där gas, broms, styrning, växel, farthållare och marklutning kan definieras i förväg.

CAN-integrationen bör också tolkas med viss försiktighet. Arbetet visar att simulerade signaler kan kodas och skickas via CAN, men endast ett urval av signaler har implementerats. I ett verkligt fordonsnätverk finns ofta många beroenden mellan signaler, uppdateringsfrekvenser och ECU:ers förväntade tillstånd. Att enskilda signaler kan skickas korrekt innebär därför inte automatiskt att hela fordonsnätverket kan simuleras på ett realistiskt sätt. För mer omfattande HIL-testning krävs fler signaler, mer systematisk hantering av signalberoenden och en tydligare strategi för vilka noder i testmiljön som ska ersättas eller kompletteras av simulatorn.

Det är även viktigt att diskutera relationen mellan simulering och fysisk testning. Verktyget bör inte ses som en ersättning för fysiska tester, utan som ett komplement som kan användas tidigare i utvecklingsprocessen. Fysiska tester behövs fortfarande för att verifiera verkligt fordonsbeteende, hårdvarupåverkan, sensorrespons och systemegenskaper som inte fångas av en förenklad modell. Däremot kan ett verktyg av denna typ bidra till att identifiera fel tidigare, testa grundläggande signalflöden och förbereda integration innan ett komplett fordon eller en fullständig testrigg är tillgänglig.

Utifrån arbetet kan några mer generella slutsatser dras för liknande projekt. För det första är det värdefullt att utveckla simuleringssystem stegvis, där en enkel modell först verifieras innan GUI, drivlina och kommunikation byggs på. För det andra bör gränssnittet mellan modell och kommunikation vara tydligt definierat, exempelvis genom ett snapshot-objekt eller motsvarande datalager. För det tredje är virtuell CAN-testning ett viktigt mellansteg innan fysisk inkoppling, eftersom fel i kanalval, bitrate, DBC-kodning eller signalmappning kan upptäckas utan risk för påverkan på hårdvara. Dessa principer är inte specifika för just bussimulering, utan kan vara relevanta även för andra projekt där simulerade systemtillstånd ska överföras till en befintlig HIL- eller labbmiljö.

Sammanfattningsvis visar arbetet att det utvecklade verktyget har ett tydligt värde som principbevis för simuleringsbaserad verifiering av fordonsbeteenden. Dess främsta styrka ligger i att det demonstrerar en fungerande systemkedja från modell till GUI och vidare till CAN-baserad signalöverföring. Den huvudsakliga svagheten är att verktyget ännu inte har den modellprecision, signalomfattning eller automatisering som krävs för bred industriell användning. Resultaten är därför mest relevanta

för tidiga utvecklingsfaser, där målet är att testa funktionella samband, verifiera signalflöden och minska tröskeln till senare fysisk integration.

6.2 Metodkritik och tillförlitlighet

Projektets metod var väl lämpad för att utveckla ett funktionellt principbevis, *proof of concept*. Det iterativa arbetssättet gjorde det möjligt att börja med en enkel fordonsmodell och därefter stegvis bygga ut systemet med drivlina, grafiskt användargränssnitt och CAN-kommunikation. På så sätt kunde delkomponenter testas innan de integrerades i en större helhet.

Tillförlitligheten stärks av att systemet verifierades på flera nivåer. Fordonsmodellen testades först internt, CAN-kommunikationen verifierades i en virtuell miljö och signalöverföringen kontrollerades därefter mot fysisk testmiljö. Detta ger stöd för att prototypens huvudsakliga funktioner fungerar enligt projektets avgränsade mål.

Samtidigt innebär metoden att utvärderingen främst blev funktionell. Fokus låg på att visa att verktyget kunde simulera fordonsbeteenden, visualisera dessa i realtid och skicka signaler via CAN. Däremot genomfördes ingen omfattande kvantitativ analys av modellens noggrannhet, signalernas latens, jitter, långtidsprestanda eller jämförelse mot verklig fordonsdata. Detta begränsar möjligheten att bedöma verktygets precision och robusthet i mer krävande testfall.

En ytterligare begränsning är att prototypen huvudsakligen styrs manuellt via det grafiska användargränssnittet. Detta är användbart för demonstration och felsökning, men gör det svårt att upprepa exakt samma körscenario flera gånger. För mer systematisk verifiering hade scriptade och repeterbara testfall varit en viktig komplettering.

CAN-integrationen testades också med ett begränsat urval av signaler och i en specifik testuppställning. Resultaten visar därför att den grundläggande kedjan från simulering till CAN-signal fungerar, men inte att verktyget kan ersätta ett komplett fordonsnätverk eller fysisk testning i alla sammanhang.

Sammanfattningsvis bedöms metoden vara tillräckligt tillförlitlig för att visa att konceptet fungerar som ett principbevis. Resultaten bör dock tolkas som en funktionell validering av systemkedjan, snarare än som en fullständig verifiering av modellens fysikaliska noggrannhet eller industriella robusthet.

6.3 Framtida utvecklingsområden

För att vidareutveckla verktyget bör flera områden prioriteras. Ett första område är att förbättra och validera fordonsmodellen. Detta kan omfatta mer realistisk drivlina, koppling, motorbromsning, tomgångsvarvtal, motorstopp och mer detaljerad däckmodellering. För att öka modellens tillförlitlighet bör modellen även kalibreras och jämföras mot mätdata från verkliga fordon. En sådan vidareutveckling skulle göra modellen mer användbar för fler typer av tester och ge bättre underlag för bedömning av dess noggrannhet.

Ett andra område är stöd för fördefinierade och repeterbara testscenarier. Genom att låta användaren definiera körförlopp över tid, exempelvis gaspådrag, bromsning, styrvinkel, växelval och lutning, skulle samma scenario kunna köras flera gånger med identiska indata. Detta skulle stärka verktygets värde för systematisk verifiering och göra det mer lämpligt för regressionstester.

Ett tredje område är att utöka CAN-integrationen. Fler signaler bör implementeras, och stöd för flera CAN-linor kan behövas för att bättre efterlikna en verklig fordonsmiljö. Verktyget skulle även kunna vidareutvecklas för att hantera inkommande CAN-meddelanden, vilket skulle möjliggöra mer interaktion mellan simulatorn och testmiljön. Även funktioner för filtrering, prioritering, loggning och felinjektion av signaler skulle förbättra användbarheten i en industriell testmiljö.

Ett fjärde område är att vidareutveckla det grafiska användargränssnittet och analysstödet. En mer flexibel testmiljö där användaren kan välja vilka signaler, grafer och kontrollfunktioner som ska visas skulle göra verktyget mer anpassningsbart. Detta bör kombineras med export av testdata, tydligare loggning och möjlighet att jämföra simulerade resultat mot förväntade referensvärden efter genomförda testfall.

Sammanfattningsvis visar projektet att ett CAN-baserat simuleringsverktyg har potential att stödja tidigare verifiering av fordonsmjukvara. För att uppnå högre användbarhet i en industriell kontext behöver verktyget dock vidareutvecklas med högre modellrealism, bättre repeterbarhet, mer omfattande CAN-integration och tydligare stöd för analys och automatiserad verifiering.

7

Slutsats

Examensarbetets syfte var att utveckla ett CAN-baserat simuleringsverktyg för tidig verifiering av fordonsbeteenden i labbmiljö. Arbetet visar att detta är möjligt genom att kombinera en mjukvarubaserad fordonsmodell, ett grafiskt användargränssnitt och en modul för CAN-kommunikation. Den utvecklade prototypen demonstrerar hur simulerade körscenarier kan användas som signalunderlag i en testmiljö utan att ett fysiskt fordon behöver vara anslutet.

Den första delfrågan behandlade hur en förenklad men utbyggbar fordonsmodell kan utformas för att simulera centrala körbeteenden i realtid. Resultatet visar att en komponentbaserad och objektorienterad modellstruktur är lämplig för detta ändamål. Fordonsmodellen kan generera centrala tillstånd såsom hastighet, acceleration, orientering, växel, motorvarvtal och hjulrelaterade variabler. Genom att separera tillståndshantering, drivlina, körlogik och kinematisk uppdatering skapades en struktur som både fungerar för enklare körfall och kan vidareutvecklas med mer detaljerade delmodeller.

Den andra delfrågan behandlade hur simulerade fordonsrelaterade signaler kan kopplas till CAN-meddelanden och överföras till en befintlig HIL-miljö. Arbetet visar att detta kan genomföras genom att koppla modellens interna tillstånd till signaldefinitioner i DBC-filer och därefter koda och skicka signalerna via CAN. Den fysiska integrationen visade att simulerade signaler kunde observeras i testmiljön, vilket verifierar hela kedjan från mjukvarubaserad simulering till extern signalöverföring.

Den tredje delfrågan behandlade vilka möjligheter och begränsningar som uppstår när ett simuleringsbaserat verktyg används för tidigare verifiering av fordonsmjukvara. Resultatet visar att verktyget kan bidra till ett mer shift-left-orienterat arbetssätt genom att möjliggöra tidig testning av enklare fordonsbeteenden i en kontrollerad labbmiljö. Verktyget kan därmed användas som stöd vid funktionell verifiering, felsökning och tidig systemintegration innan ett komplett fysiskt fordon finns tillgängligt.

Samtidigt bör prototypen betraktas som ett funktionellt principbevis snarare än ett färdigt industriellt verifieringsverktyg. Fordonsmodellen är förenklad och inte kalibrerad mot mätdata från ett verkligt fordon. CAN-integrationen omfattar endast ett urval av signaler, och verktyget saknar i nuläget stöd för fullständigt automatiserade och repeterbara testscenarier. Dessa begränsningar innebär att verktyget lämpar sig bäst för tidiga funktionella tester där realistiska, men inte fullständigt fysikaliskt exakta, signaler är tillräckliga.

Den utvecklade prototypen visar därmed att simuleringsbaserad CAN-signalering kan fungera som ett praktiskt stöd för tidig verifiering, men att fortsatt arbete krävs innan verktyget kan användas som en robust och generell verifieringsplattform i industriell fordonsutveckling.

Referenser

[1] J. O. Clark, "System of systems engineering and family of systems engineering from a standards, V-Model, and Dual-V Model perspective," in *2009 3rd Annual IEEE Systems Conference*. Vancouver, BC, Canada: IEEE, 2009.

[2] IBM, "Shift-left testing," https://www.ibm.com/think/topics/shift-left-testing, hämtad: 2026-03-01.

[3] Devopedia, "Shift Left," https://devopedia.org/shift-left, Feb. 2022, version 6. Hämtad: 2026-05-01.

[4] P.-Å. Jansson, R. Grahn, and M. Enelund, *Mekanik: statik och dynamik*, 4th ed. Lund, Sverige: Studentlitteratur AB, 2018.

[5] MathWorks, "6DOF (Euler Angles)," https://www.mathworks.com/help/aeroblks/6dofeulerangles.html, hämtad: 2026-03-01.

[6] B. Thomas, *Modern reglerteknik*, 3rd ed. Stockholm, Sverige: Liber AB, 2001.

[7] R. Buttigieg, M. Farrugia, and C. Meli, "Security issues in controller area networks in automobiles," in *2017 18th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*. Monastir, Tunisia: IEEE, 2017, pp. 93–98.

[8] RF Wireless World, "SAE J1939 frame format and message structure," https://www.rfwireless-world.com/terminology/sae-j1939-frame-format-message-structure, hämtad: 2026-05-08.

[9] M. Falch, "CAN DBC file explained – a simple intro," https://www.csselectronics.com/pages/can-dbc-file-database-intro, 2024, hämtad: 2026-05-02.

[10] Vector Informatik GmbH, "Network interfaces," https://www.vector.com/int/en/products/products-a-z/hardware/network-interfaces/, hämtad: 2026-05-07.

[11] ——, "Vector Hardware Manager," https://www.vector.com/int/en/products/products-a-z/hardware/network-interfaces/vector-hardware-manager/, hämtad: 2026-05-02.

[12] USB Implementers Forum, "USB.org," https://usb.org/, hämtad: 2026-05-07.

[13] Python Software Foundation, *Python 3.11.14 Documentation*, https://docs.python.org/3.11/, 2025, hämtad: 2026-05-03.

[14] NumPy Community, "NumPy documentation," https://numpy.org/doc/, 2025, hämtad: 2026-05-03.

[15] The SciPy Community, "SciPy documentation, version 1.17.0," https://docs.scipy.org/doc/scipy/, 2026, hämtad: 2026-05-03.

[16] The Matplotlib Development Team, “Matplotlib: Visualization with Python,” https://matplotlib.org/, 2024, hämtad: 2026-05-02.

[17] The Qt Company, “Qt for Python – PySide6.QtWidgets,” https://doc.qt.io/qtforpython-6/PySide6/QtWidgets/index.html, hämtad: 2026-05-07.

[18] python-can contributors, “python-can documentation,” https://python-can.readthedocs.io/en/stable/index.html, 2025, hämtad: 2026-05-02.

[19] E. Moqvist, “cantools – CAN bus tools in Python,” https://cantools.readthedocs.io/en/latest/, 2023, hämtad: 2026-05-02.

[20] Vector Informatik GmbH, “CANalyzer – ECU and network analysis,” https://www.vector.com/int/en/products/products-a-z/software/canalyzer/, 2026, hämtad: 2026-05-03.

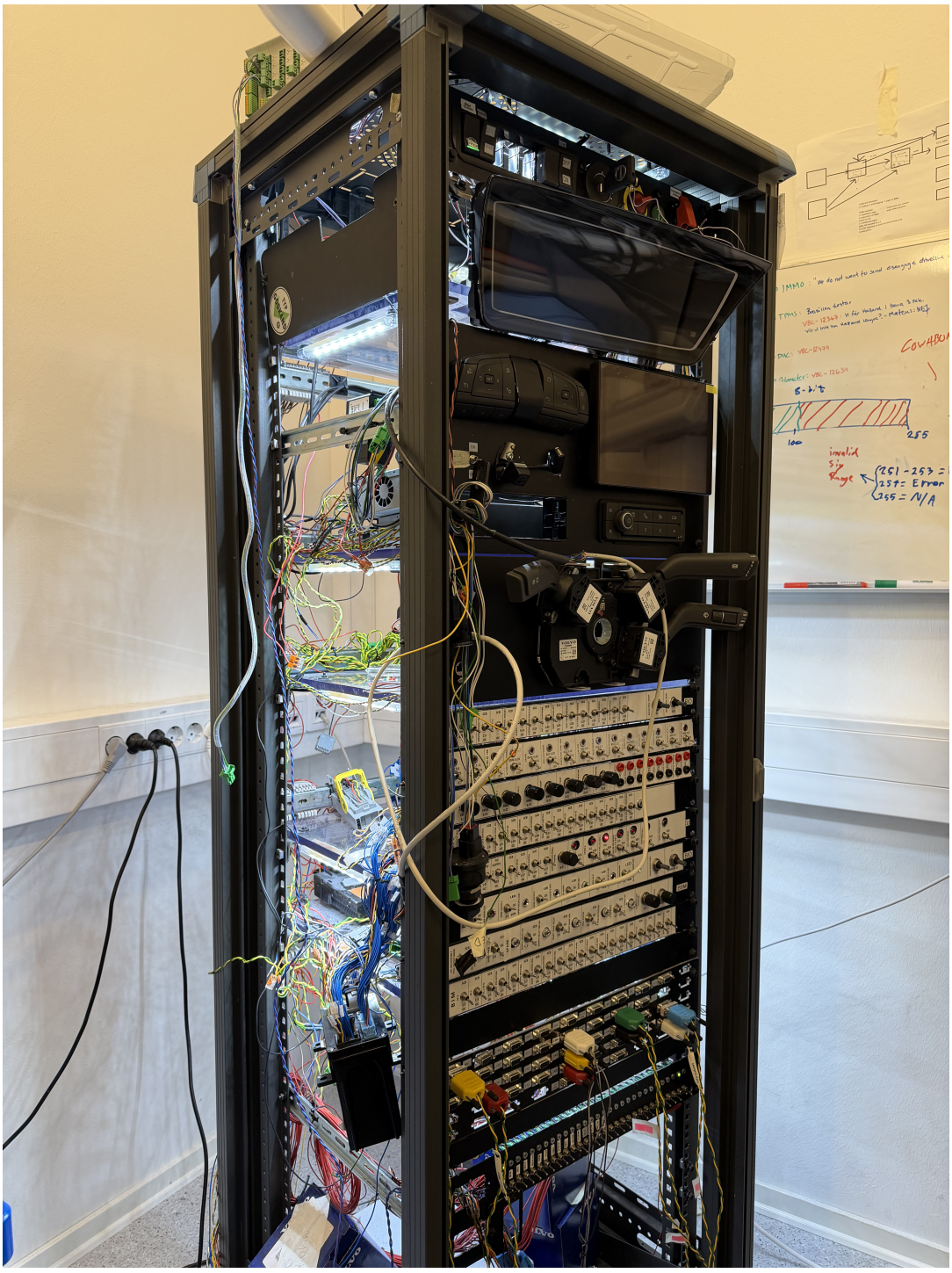
A

Kompletterande material

Detta appendix innehåller kompletterande material kopplat till integration och testning av simuleringsverktyget. Här redovisas dels den fysiska HIL-rigg som användes vid integrationen, dels den CAN-konfiguration som användes för att ansluta verktyget till virtuella och fysiska CAN-kanaler.

A.1 Fysisk HIL-rigg

Figur A.1 visar den fysiska HIL-rigg som användes vid integration och verifiering av simuleringsverktyget mot Volvo Bussars testmiljö. Riggen användes för att kontrollera att simulerade signaler från verktyget kunde skickas via CAN och observeras i den externa testmiljön.



Figur A.1: Den fysiska HIL-riggen som användes vid integration och verifiering av simuleringsverktyget mot Volvo Bussars testmiljö.

Den fysiska riggen användes efter att den virtuella CAN-kommunikationen hade verifierats. Genom att först testa signalöverföringen virtuellt kunde grundläggande fel i kanalval, bitrate och signalmappning hanteras innan anslutning till fysisk hårdvara. Vid den fysiska testningen användes riggen för att bekräfta att signalerna kunde tas emot och granskas utanför simuleringsverktygets interna mjukvarumiljö.

A.2 CAN-konfiguration

För att ansluta simuleringsverktyget till de virtuella och fysiska CAN-kanalerna användes en konfigurationsfil, `can.ini`. Filen definierar de CAN-kanaler som används av programmet, inklusive interface, applikationsnamn, kanalnummer, bitrate och om egna meddelanden ska tas emot. Konfigurationen som användes i projektet visas i Listing A.1.

```
[CAN1]
interface = vector
app_name = vehicle_simulation_tool
channel = 0
bitrate = 250000
receive_own_messages = False

[CAN2]
interface = vector
app_name = vehicle_simulation_tool
channel = 1
fd = False
bitrate = 250000
receive_own_messages = False
```

Listing A.1: CAN-konfiguration som användes för simuleringsverktygets Vector-baserade CAN-anslutning.

Konfigurationen innehåller två CAN-kanaler, `CAN1` och `CAN2`. Båda använder `vector` som interface och applikationsnamnet `vehicle_simulation_tool`, vilket behövde motsvara konfigurationen i Vector Hardware Manager. Kanal 0 och kanal 1 användes för att möjliggöra testning mellan två CAN-kanaler, exempelvis vid virtuell loopbaserad verifiering där signaler skickades på en kanal och togs emot på den andra.

Bitraten sattes till 250 kbit/s för att överensstämma med den aktuella testmiljöns konfiguration. Detta är viktigt eftersom samtliga noder på samma CAN-buss måste använda samma bitrate för att meddelanden ska kunna tolkas korrekt. Inställningen `receive_own_messages = False` användes för att undvika att programmet tog emot sina egna skickade meddelanden på samma kanal.

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS