TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Pavel Pavliv 233021IAIB
Rasmus Raasuke 222885IAIB

DBC-põhise Rusti ja C++-i koodigeneraatori arendamine

Bakalaureusetöö

Juhendaja: Tavo Annus
MSc

Tallinn 2026

Autorideklaratsioon

Kinnitame, et oleme koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autorid: Pavel Pavliv ja Rasmus Raasuke

01.06.2026

Lühikokkuvõte

CAN on sardsüsteemides levinud andmevahetuse protokoll. CAN-võrgus edastatavate sõnumite struktuuri ja tähendust kirjeldatakse sageli DBC-failidega. Selliste failide põhjal lähtekoodi genereerimine CAN-sõnumitega töö jaoks on levinud praktika, kuid olemasolevatel tööriistadel ilmneb mitmeid puudusi.

Ettevõttes Milrem AS genereeritakse koodi DBC-failide põhjal, kuid olemasolevate tööriistade puuduste tõttu tehakse lisaks manuaalset tööd. Käesoleva lõputöö eesmärk on luua tööriist, mis genereerib Milremi vajadustele vastavat koodi.

Töö käigus analüüsitakse põhjalikult DBC-formaati, uuritakse olemasolevate lahenduste puudusi ning sõnastatakse täpsemad nõuded uuele tööriistale. Nende põhjal valitakse projekti jaoks kompilaatoritest inspireeritud arhitektuur ning arendatakse uus tööriist.

Lahenduse praktilist väärtust hinnatakse võrdluses olemasoleva dbc-codegen tööriistaga ning Milremi tagasiside põhjal.

Töö tulemusena valmib avatud lähtekoodiga projekt dbc-codegen2. Loodud tööriist ei lahenda ainult Milremi konkreetseid probleeme, vaid on kasutatav ka üldotstarbelise DBC-põhise koodigeneraatorina, mis täidab lünga olemasolevates tööriistades.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 35 leheküljel, 8 peatükki, 9 joonist, 1 tabel.

Abstract

Development of a DBC-based Code Generator for Rust and C++

CAN is a widely used communication protocol in embedded systems. The structure and meaning of messages transmitted over a CAN network are often described using DBC files. Generating source code for working with CAN messages based on such files is a common practice. However, tooling is lacking.

At Milrem AS, code is generated from DBC files, but existing tooling requires manual work. The goal of this thesis is to create a tool that generates code meeting Milrem's needs.

In the thesis, the DBC format is analyzed in detail, the shortcomings of existing solutions are studied, and precise requirements for the new tool are formulated. Based on these requirements, a compiler-inspired architecture is selected for the project, and a new tool is developed.

The practical value of the solution is evaluated through comparison with the existing dbc-codegen tool. In addition, feedback from Milrem is considered.

As a result of the thesis, an open-source project named dbc-codegen2 is developed. The created tool not only solves Milrem's specific problems but can also be used as a general-purpose DBC-based code generator that fills the gap in existing tooling.

The thesis is in Estonian and contains 35 pages of text, 8 chapters, 9 figures, 1 table.

Lühendite ja mõistete sõnastik

Alignment	Joondamine
API	*Application Programming Interface*, rakendusliides
CAN	*Controller Area Network*, kontrollerite alavõrk
DBC	*CAN Database*, CAN-andmebaas
EBNF	*Extended Backus-Naur Form*, laiendatud Backus-Nauri vorm
Extended Multiplexing	Laiendatud multipleksimine
Getter	Lugemismeetod
Padding	Täidistus
Setter	Muutmismeetod

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Autotööstuses kasutatakse seadmete vaheliseks suhtluseks sageli CAN (*Controller Area Network*) protokolli [1]. CAN on sõnumipõhine mitme sõlmega suhtlusprotokoll, kus võrgus saadetud sõnumid on nähtavad kõigile samal siinil olevatele seadmetele. Protokoll määrab muu hulgas sõnumite edastamise viisi, kaadri üldise struktuuri, sõnumi identifikaatorite kasutamise, veatuvastuse ning selle, kuidas lahendatakse olukord, kus mitu seadet alustavad samal võrgul andmete saatmist samal ajal [2].

CAN-kaadri andmeväli sisaldab aga ainult toorbaite ning CAN-protokoll ise ei kirjelda, millist tähendust need baidid rakenduse tasemel kannavad. Selleks kasutatakse DBC (CAN *Database*) faile. DBC-fail kirjeldab CAN-võrgus kasutatavaid sõnumeid, nende identifikaatoreid ja nimesid ning annab eeskirja andmevälja tõlgendamiseks. Samuti võib DBC-fail kirjeldada võrgus olevaid sõlmi ning seda, millised sõnumid on konkreetsete sõlmedega seotud [3]. DBC-faili näidis on esitatud lisas 2.

Tegu on tellimustööga, kus tellijaks on Milrem AS. Milremi toodetes toimub erinevate seadmete vaheline suhtlus CAN-protokolli abil ning suurem osa sellest suhtlusest on kirjeldatud DBC-failidega.

1.1 Lahendatav probleem ja töö eesmärk

Tellijal on vaja DBC-failides kirjeldatud CAN-sõnumite jaoks Rusti ja C++-i koodi, mis võimaldaks neid sõnumeid koostada, kodeerida ja dekodeerida. Käsitsi DBC-faili põhjal koodi kirjutamine on aeganõudev ja veaohtlik tegevus [4], [5]. Seetõttu on vaja lahendust, mis suudab DBC-failis kirjeldatud sõnumitest ja signaalidest automaatselt genereerida korrektset ning tellija kasutusjuhtudele sobivat koodi [6].

Olemasolevad tööriistad lahendavad seda probleemi ainult osaliselt. Ei ole tööriista, mis genereeriks C++-i koodi ning olemasolevad Rusti koodi generaatorid ei vasta tellija ootustele. Olemasolevaid tööriistu analüüsitakse sektsioonis 4.3.

Lõputöö eesmärk on luua teek ning käsurea tööriist Rusti ja C++-i koodi genereerimiseks DBC-faili põhjal. Loodav lahendus peab vastama tellija vajadustele ja nõuetele, mis kujunevad välja projekti käigus koostöös tellijaga.

1.2 Lõputöö struktuur

Antud lõputöö on ülesehitatud järgnevalt:

- Peatükis 2 antakse ülevaade DBC-formaadist ja CAN-sõnumitest.
- Peatükis 3 kirjeldatakse töö metoodikat. Selgitatakse, kuidas kaardistati tellija nõuded, milliseid tööriistu ja tehnoloogiaid kasutati ning kuidas planeeriti loodud lahenduse valideerimist.
- Peatükis 4 analüüsitakse DBC-formaati, võrreldakse olemasolevaid lahendusi ja selgitatakse, miks need ei vasta täielikult tellija vajadustele. Samuti sõnastatakse projekti nõuded.
- Peatükis 5 tutvustatakse loodud tööriista arhitektuuri ja implementatsiooni. Kirjeldatakse, kuidas DBC-fail parsitakse, teisendatakse vaheesituseks, valideeritakse semantiliselt ning kasutatakse Rusti ja C++-i koodi genereerimiseks. Lisaks selgitatakse genereeritud koodi API (*Application Programming Interface*) disaini ja testimise põhimõtteid.
- Peatükis 6 hinnatakse loodud lahendust. Võrreldakse loodud tööriista olemasoleva dbc-codegen lahendusega, analüüsitakse genereeritud koodi ohutust ja kasutusmugavust, mõõdetakse jõudlust ning esitatakse valdkonna spetsialistidelt kogutud tagasiside tulemused.
- Peatükis 7 kirjeldatakse töö käigus ilmnenud piiranguid ja võimalikke edasiarendusi. Nende hulka kuuluvad vaheesituse lihtsustamine, tehnilise võla vähendamine ja uute funktsionaalsuste lisamine.

2 Taust

DBC-faili kasutatakse CAN-võrgu sõnumite kirjeldamiseks [3]. Autotööstuses on DBC-formaat laialt levinud ning seda peetakse valdkonna *de facto* standardiks [7]. Vaatamata laiale kasutusele ei ole DBC-faili struktuur ametlikult avalikustatud. Formaadi töötas välja Vector Informatik GmbH tööriista CANdb++ jaoks. Avalikult kättesaadavad formaadi kirjeldused põhinevad suuresti kogukonna tehtud pöördkonstrueerimisel [8]. DBC-faili peamisteks osadeks on sõnumite ja signaalide definitsioonid.

2.1 CAN-sõnumid ja signaalid

CAN-sõnum on võrgu kaudu edastatav andmeüksus. DBC-failis määratakse sõnumi nimi, identifikaator ning andmevälja pikkus. Klassikalise CAN-sõnumi andmeväli on kuni kaheksa baiti [2]. DBC-fail kirjeldab, kuidas see baitide jada jaguneb üksikuteks signaalideks [3].

Signaal kirjeldab ühte väärtust sõnumi andmeväljas. Iga signaali puhul määratakse selle algusbitt, pikkus bittides, baitide järjekord, märgilisus, skaleerimistegur, nihe ning füüsilise väärtuse minimaalne ja maksimaalne lubatud väärtus. Lisaks võib signaal olla seotud väärtuste loendiga [3].

Signaalil on kaks olulist esitusviisi. Esimene on toorväärtus, mis paikneb CAN-sõnumi andmeväljas bittide jadana. Teine on füüsiline väärtus, mida kasutab rakendus. Nende väärtuste vahel on lineaarne seos. Füüsilise väärtuse saamiseks rakendatakse toorväärtusele skaleerimistegur ja nihe [3]. Seda seost kirjeldab valem (2.1):

$$\text{füüsiline väärtus} = \text{toorväärtus} \cdot \text{tegur} + \text{nihe}. \tag{2.1}$$

2.2 Multipleksitud sõnumid

DBC-formaadis võivad sõnumid olla multipleksitud [9]. Multipleksitud sõnumis määrab üks signaal, mida nimetatakse multiplekseriks, millised teised signaalid on konkreetse sõnumi

korral aktiivsed [9]. Näiteks võib sõnum sisaldada multiplekserit, mille väärtused 0 ja 1 tähistavad kahte erinevat sõnumivarianti. Kui multiplekseri väärtus on 0, tuleb tõlgendada ainult neid signaale, mis kuuluvad variandi 0 alla. Kui väärtus on 1, tõlgendatakse variandi 1 signaale.

3 Metoodika

Töö metoodika põhineb tarkvaraarendusprojekti tavadel, kus oluline on nii tellija probleemide mõistmine kui ka loodud lahenduse süstemaatiline kontrollimine [10].

3.1 Eelanalüüs ja planeerimine

Enne arendust kaardistatakse koostöös tellijaga olemasolevate lahenduste probleemid ning tööriistale seatavad ootused. Lisaks uurivad autorid ka iseseisvalt olemasolevaid lahendusi ja nende poolt genereeritud koodi. Selline eelanalüüs aitab vältida juba teadaolevaid kitsaskohti ning annab aluse uue lahenduse arhitektuuriliste otsuste tegemiseks.

Kogutud info põhjal koostatakse arendusplaan, milles määratakse tööriista põhikomponendid, nende vastutusala ning omavahelised seosed. Planeerimisel lähtutakse põhimõttest, et lahendus peab olema lihtsasti edasiarendatav ja testitav.

3.2 Arendusprotsess

Arendus toimub iteratiivselt koostöös tellijaga [10]. Töö käigus täpsustatakse nõudeid vastavalt tellija tagasisidele ning arenduse käigus ilmnenud tehnilistele probleemidele. Selline töökorraldus võimaldab lahendust järk-järgult kohandada tegelikele kasutusvajadustele.

Töö jaotatakse autorite vahel väiksemateks loogilisteks ülesanneteks. Arendamisel toetutakse agiilse arenduse põhimõtetele: muudatusi tehakse väikeste sammudena, lahendust testitakse jooksvalt ning vajaduse korral muudetakse varasemaid otsuseid [10]. Projekti arendus toimub ühes salves ja ühes harus. Blokeerivaid koodiülevaatusi ei tehta, kuid autorid arutavad arenduse käigus tehtavad muudatused omavahel läbi ning langetavad olulisemad otsused koos.

3.3 Kasutatud tööriistad ja tehnoloogiad

Tellija soovi tõttu implementeeritakse tööriist programmeerimiskeeles Rust. Lisaks on Rust ka kiire ja pakub mugavaid võimalusi käsurea rakenduse kirjutamiseks [11]. Projekti lähtekoodi hallatakse GitHubi avalikus salves. Sõltuvuste haldamiseks kasutatakse Rusti standardset paketihaldurit Cargo. Võimaluse korral kasutatakse olemasolevaid teeke, et vältida korduva funktsionaalsuse käsitsi realiseerimist ja keskenduda töö põhiloogikale.

Olulisemad kasutatud teegid on:

1. can-dbc – DBC-failide parsimiseks.
2. quote, proc-macro2 ja syn – Rusti lähtekoodi genereerimise lihtsustamiseks.
3. prettyplease – genereeritud Rusti koodi vormindamiseks.
4. heck – nimede kirjaviisi teisendamiseks Rusti tavadele vastavasse kujusse.
5. clap – käsurea liidese argumentide kirjeldamiseks ja töötlemiseks.
6. anyhow – veakäsitluse lihtsustamiseks.
7. embedded-can – CAN-sõnumitega seotud tüüpide ja liideste kasutamiseks.

Testimisel kasutatakse lisaks teeke insta, arbitrary ja tempfile. insta võimaldab hetktõmmis-testimist, mille abil kontrollitakse genereeritud väljundi vastavust oodatud tulemusele [12]. arbitrary abil genereeritakse pseudojuhuslikke väärtusi ning tempfile abil luuakse testide käigus ajutisi faile ja katalooge.

3.4 Valideerimisplaan

Töö tulemuste valideerimiseks kasutatakse mitut meetodit. Võrreldakse uut lahendust tellija poolt hetkel kasutatud tööriistaga ning kogutakse eksperthinnanguid.

3.4.1 Võrdlus olemasoleva dbc-codegen tööriistaga

Autorid võrdlevad arendatavat tööriista olemasoleva dbc-codegen lahendusega. Võrdluses hinnatakse tööriistade kasutusmugavust, jõudlust, sisendi valideerimist ja genereeritud koodi mugavust ning tüübiohutust. Lisaks vaadeldakse, kui hästi kumbki lahendus toetab erinevaid DBC-failides esinevaid struktuure ning kui arusaadavad on kasutajale esitatavad veateated.

3.4.2 Ekspertide tagasisideuuring

Lisaks tellija pidevale tagasisidele huvitab autoreid ka laiema kogukonna arvamus. DBC-failide spetsiifilisuse tõttu ei saa aga küsida tagasisidet kõigilt, vaid on vaja leida inimesi, kes on selle valdkonna eksperdid.

Eksperthinnangute kogumiseks koostavad autorid kaks küsimustikku: ühe Rusti ja teise C++-i kasutajatele. Küsimustikes esitatakse erinevad kasutusjuhud, mida arendatud tööriistaga saab lahendada, ning küsitakse tagasisidet selle kohta, kui mugav ja arusaadav on vastavat ülesannet täita.

Küsimustikud saadetakse olemasolevate tööriistade aktiivsetele arendajatele, et saada nende arvamusi uue lahenduse kohta. Samuti tehakse tööriista kohta postitused kolmes Redditi kogukonnas: *r/embedded*, *r/rust* ja *r/cpp*. Postitustes kutsutakse kasutajaid tööriistaga tutvuma ning andma tagasisidet.

Saadud hinnangud aitavad tuvastada tööriista tugevusi ja puudusi ning annavad sisendit edasiseks parendamiseks.

3.5 Valideerimisandmete kogumine ja analüüs

Tagasisideuuringu tulemused kogutakse läbi Google Formsi. Analüüsi tulemuste põhjal tehakse vajaduse korral muudatusi tööriista API-s, veateadetes, dokumentatsioonis või genereeritud koodi struktuuris.

4 Eelanalüüs

Autoritel puudub varasem kokkupuude DBC-failidega, mistõttu alustatakse DBC-formaadi uurimisest. Samuti analüüsitakse olemasolevaid koodigeneraatoreid. Tulemuste põhjal sõnastatakse nõuded, millele uus lahendus peab vastama.

4.1 DBC-faili grammatika ja struktuur

DBC-faili võib käsitleda formaalse keelena, millel on oma grammatika ja lubatud lausetüübid [13]. Eelanalüüsi käigus koostavad autorid DBC-faili süntaksist EBNF-kujulise (*Extended Backus–Naur form*) kirjelduse. Selle eesmärk on saada ülevaade formaadis esinevatest konstruktsioonidest ning eristada koodigeneraatori jaoks olulised osad nendest osadest, mida loodav tööriist ei pea toetama.

Analüüsi tulemusena eristatakse DBC-formaadis 29 erinevat osa. Ükski neist ei ole formaadi tasemel absoluutselt kohustuslik, mistõttu võivad DBC-failid olla nii mahu kui ka sisu poolest väga erinevad.

Koodigeneraatori seisukohast on kõige olulisemad järgmised osad:

- sõnumite definitsioonid, mis kirjeldavad CAN-sõnumite identifikaatoreid, nimesid ja andmevälja pikkust;
- signaalide definitsioonid, mis kirjeldavad sõnumi andmeväljas paiknevaid väärtusi;
- kommentaarid, mida saab seostada sõnumite ja signaalidega;
- väärtuste loendid, mille abil saab numbrilistele väärtustele anda tähenduslikud nimetused.

Ülejäänud DBC-faili osad jäävad projekti ulatusest välja, kuna need on kas vananenud, seotud CANdb++ tööriista metaandmetega, ebapiisavalt dokumenteeritud või ei mõjuta otseselt genereeritava koodi funktsionaalsust.

4.2 DBC-formaadi semantilised probleemid

Selgub, et DBC-fail võib olla süntaktiliselt korrektne, kuid sisaldada siiski semantilisi vastuolusid või kirjeldada ohtlikku andmemudelit. Koodigeneraatori seisukohast saab DBC-failis esinevad probleemid jagada kaheks. Esimesse rühma kuuluvad probleemid, mille tõttu võib genereeritud kood muutuda kompileerimatuks. Sellised probleemid on näiteks nimekonfliktid või sihtkeeles keelatud võtmesõnade kasutamine.

Teise rühma kuuluvad sisulised vastuolud, mille korral genereeritud kood võib küll kompileeruda, kuid anda käitusajal vale või ebatäpse tulemuse. Selliste probleemide tuvastamise keerukus sõltub vastuolu laadist. Osa neist on tuvastatavad lihtsate kontrollidega. Näiteks võib DBC-fail sisaldada dubleeritud sõnumiidentifikaatoreid, kattuvaid signaalibitte, sõnumi andmeväljast välja ulatuvaid signaale, nulliga võrdset skaleerimistegurit või füüsilise väärtuse miinimumi, mis on maksimumist suurem.

Samas leidub ka semantilisi probleeme, mille tuvastamine ei piirdu üksikute väljade või lihtsate vastuolude kontrollimisega. Eelanalüüsi käigus eristusid kolm mittetriviaalset probleemiklassi, mis puudutavad DBC-formaadis kirjeldatud väärtuste esitatavust, teisendatavust ja aritmeetilist korrektsust.

Üks probleemide klass on seotud füüsiliste väärtuste esitatavusega. Näiteks võib signaal olla defineeritud 8-bitise märgita täisarvuna, mille tegur on 1 ja nihe 0. Sellisel juhul on toorväärtuste vahemik $[0, 255]$ ning ka füüsiliste väärtuste tegelik vahemik on $[0, 255]$. DBC-fail lubab signaalile siiski määrata muu minimaalse või maksimaalse füüsilise väärtuse. Kui kasutaja määratud vahemik ei lange kokku tegelikult esitatava vahemikuga, võib tekkida olukord, kus osa lubatud väärtustest ei ole sõnumi andmeväljas esitatavad.

Teine probleemide klass on seotud informatsioonikaoga. Kui füüsilise väärtuse teisendamisel toorväärtuseks tuleb teha täisarvuline jagamine, võib väärtuse salvestamisel osa informatsioonist kaduda. Samuti ei saa täisarvulise toorväärtusega alati täpselt esitada ujukomaarvulist füüsilist väärtust. Sellisel juhul on võimalik väärtust esitada ainult teatud täpsusega, mis võib mõnes rakenduses olla vastuvõetav, kuid mõnes teises mitte.

Kolmas probleemide klass on seotud aritmeetilise üle- ja alatäitumisega. Näiteks võib signaali toorväärtus olla defineeritud märgita 8-bitise täisarvuna, mille tegur on 1, nihe -128

ning füüsiliste väärtuste vahemik $[-128, 127]$. Järelikult füüsilise väärtuse esitamiseks sobib märgiga 8-bitine täisarv. Kui toorväärtuse teisendamisel füüsiliseks väärtuseks tehakse arvutus ainult toorväärtuse andmetüübis, võib negatiivse nihke rakendamisel tekkida alatäitumine, sest märgita täisarv ei võimalda esitada negatiivseid väärtusi. Kui aga toorväärtust tõlgendada kohe märgiga täisarvuna, võib tekkida ületäitumine juhul, kui toorväärtus on suurem kui valitud märgiga andmetüübi maksimaalne esitatav väärtus.

4.3 Olemasolevate lahenduste analüüs

Järgnevalt analüüsitakse olemasolevaid lahendusi, nende puudusi ning uuritakse, kuidas need käituvad vigaste DBC-failide korral.

4.3.1 Olemasolevad lahendused ja nende puudused

Kõige tuntum ja kasutatum tööriist on cantools [14], aga see genereerib koodi ainult C programmeerimiskeeles, mida tellija ei vaja.

Autoritele teadaolevalt ei paku ükski olemasolev tööriist DBC-faili põhjal C++-i koodi genereerimist. Küll aga leidub valmislahendusi CAN-sõnumite käitusaegseks kodeerimiseks ja dekodeerimiseks. Sellist funktsionaalsust pakuvad näiteks dbcppp [15] ja can_dbc_loader [16].

Rusti ökosüsteemis on CAN-protokolli ja DBC-failidega seotud arenduse ümber kujunenud OxiBUSi kogukond [17]. Selle GitHubi organisatsiooni all on kaks peamist projekti, mis genereerivad DBC-faili põhjal Rusti koodi.

Esimene neist on dbc-codegen [18], mida tellija peamiselt kasutab. Tellija jaoks on sellel lahendusel siiski mitu puudust:

- genereeritud kood lubab konstrueerida sõnumeid, mis ei vasta DBC-failis defineeritud reeglitele;
- genereeritud kood sisaldab dubleeritud konstruktsioone;
- DBC-failis esinevatest ebaloogilistest reeglitest ei anta kasutajale teada, vaid nende põhjal genereeritakse vigane kood.

Teine projekt on dbc-data [19]. See genereerib koodi ainult nende sõnumite kohta, mille

kasutaja käsitsi määrab, mitte kõigi DBC-failis defineeritud sõnumite kohta. Autorid suhtlesid projekti haldajaga, kelle sõnul selline töövoog on teadlik disainiotsus, mida tulevikus muuta ei plaanita. Tellija vaatepunktist ei ole selline lahendus mugav, kuna nende DBC-failid võivad sisaldada üle saja sõnumi.

4.3.2 Olemasolevate lahenduste käitumine vigaste sisendite korral

DBC-formaadi semantiliste probleemide tõttu uuritakse, kuidas käituvad olemasolevad tööriistad vigaste DBC-failide korral. Võrdluseks valitakse cantools ja dbc-codegen, kuna mõlemad genereerivad DBC-faili põhjal koodi ning on seetõttu oma tööpõhimõtte ja väljundi poolest kõige lähedasemad loodavale tööriistale.

Katsete käigus ilmneb, et vaadeldud lahendused ainult parsivad DBC-faili ja genereerivad koodi. Semantilisi vastuolusid need ei tuvasta. See tähendab, et tööriistad genereerivad koodi ka juhul, kui sisendfail sisaldab vastuolusid, mille tõttu genereeritud koodi kasutamine on ebatäpne või ebaturvaline.

Erandina käsitletakse olemasolevates lahendustes nimekonflikte. Mõlemad generaatorid lisavad vajaduse korral objektidele prefikseid või kasutavad muid nimemuutmise võtteid, et vältida samanimeliste tüüpide, funktsioonide või muutujate tekkimist genereeritavas koodis.

4.4 Projekti nõuded

Tellija vajaduste, DBC-formaadi probleemide ning olemasolevate lahenduste puuduste põhjal sõnastatakse nõuded, millele loodav tööriist peab vastama. Need jagunevad kaheks: tööriista puudutavad nõuded ja genereeritavat koodi puudutavad nõuded.

4.4.1 Nõuded lahenduse tööle

Lahendust on lihtne ja mugav kasutada nii käsurea programmina kui ka teegina. Programmi sisendiks on DBC-fail ning väljundiks on Rusti või C++-i kood.

Tööriist kontrollib DBC-faili enne koodi genereerimist. Kui sisendis leitakse viga, saab kasutaja arusaadava veateate koos piisava kontekstiga. Eelistatud on lahendus, kus tööriist suudab ühe käivituse jooksul tuvastada mitu viga, mitte ei katkesta tööd esimese vea

leidmisel. See lihtsustab kasutaja töövoogu, sest nii saab kõik leitud probleemid korraga parandada.

4.4.2 Nõuded genereeritavale koodile

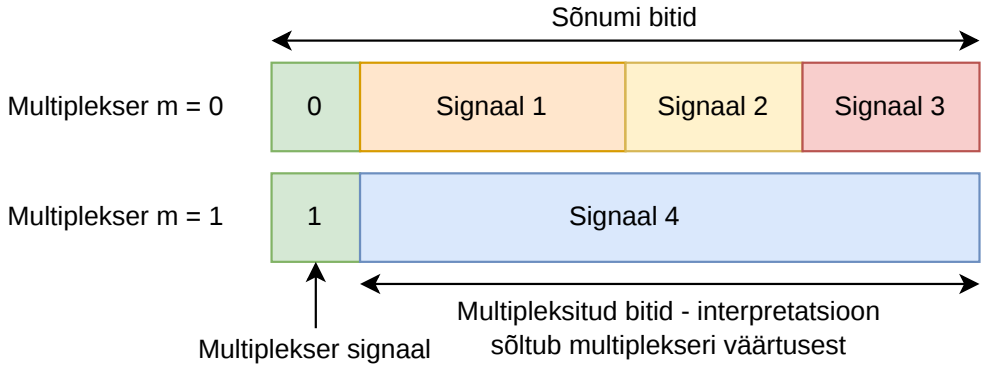
Genereeritud kood võimaldab koostada, kodeerida ja dekodeerida DBC-failis defineeritud CAN-sõnumeid. Iga sõnumi jaoks saab luua vastava kõrgetasemelise objekti, mille kaudu on võimalik lugeda ja muuta sõnumi signaalide füüsilisi väärtusi ning teisendada objekt CAN-sõnumi andmeväljaks. Lisaks saab sõnumi identifikaatori ja andmevälja põhjal konstrueerida vastava kõrgetasemelise objekti.

Genereeritud objektide konstruktorid ja muutmismeetodid (*setter*) ei võimalda luua objekte määramata ega vastuolulises olekus. Käesoleva töö kontekstis nimetatakse seda ohutuseks.

Ohutuse tagamiseks valib generaator signaalidele sobivad andmetüübid ning kontrollib väärtuste salvestamisel DBC-failis defineeritud piiranguid. Kui kasutaja antud väärtus jääb lubatud vahemikust välja, teavitab genereeritud kood sellest kasutajat.

Kui signaali väärtusega on DBC-failis seotud väärtuste loend, kajastub see genereeritud koodis eraldi tüübina. Sellisel juhul kasutavad vastava signaali lugemis- (*getter*) ja muutmismeetodid loenditüüpi, mitte täisarvulist väärtust.

Multipleksitud sõnumite puhul ei luba API olukorda, kus kasutaja saab muuta ainult multiplekseri väärtust, jättes ülejäänud andmevälja samaks. Selline operatsioon võib tekitada vastuolulise oleku, sest eri multiplekseri väärtustele vastavad signaalid võivad kasutada samu bitte. Seetõttu on multipleksitud sõnumi variant esitatud tervikliku alamstruktuurina. Kui kasutaja soovib varianti muuta, peab ta andma uue variandi tervikuna, mille põhjal uuendatakse nii multiplekseri väärtus kui ka ülejäänud andmeväli. Seda olukorda illustreerib joonis 1.



Joonis 1. Multipleksitud sõnumi näide.

Lisaks funktsionaalsusele on genereeritud kood loetav ja idiomaatiline. Kuna arendajad võivad genereeritud koodi vajaduse korral lugeda või siluda, on selle struktuur arusaadav ning kood korrektselt vormindatud.

Tellijale on oluline, et genereeritud kood töötab ka keskkonnas, kus ei saa kasutada standardteeki (no_std keskkond). See on oluline peamiselt Rusti puhul, kuna Rusti standardteek eeldab operatsioonisüsteemi, mida aga sardsüsteemi puhul ei ole [20]. C++-is saab aga kasutada osasid standardteegi päisefaile ilma probleemita [21], [22].

Samuti oskab generaator genereerida teste, mille abil saab kontrollida genereeritud koodi korrektsust.

5 Implementatsioon

Eelnevalt kirjeldatud nõuetest järeldub, et loodav tööriist ei saa olla vaid tekstipõhine koodigeneraator. Enne koodi genereerimist tuleb sisend parsida, semantiliselt analüüsida ning teisendada kujule, millest on mugav koodi genereerida.

5.1 Lahenduse arhitektuur

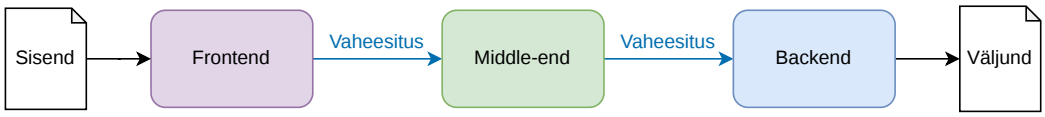
Arhitektuuri kujundamisel võetakse eeskujuks kompilaatorites levinud ülesehitus. Projekt jagatakse kolmeks põhiliseks osaks: *frontend*, *middle-end* ja *backend* [13].

Frontend'i ülesanne on lugeda sisendfail, kontrollida selle süntaksit ning koostada DBC-faili esmane sisemine vaheesitus. Kui sisendfailis esineb süntaktiline viga, ei ole edasine analüüs enam mõistlik ning tööriist peab kasutajale veast teada andma.

Middle-end vastutab vaheesituse valideerimise ja teisendamise eest. See osa töötab konveierina, kus andmemudel läbib järjest mitu töötlussõlme. Valideerimissõlmed kontrollivad semantilisi piiranguid, näiteks signaalide kattumist, väärtuste vahemikke ja nimekonflikte. Transformatsioonisõlmed teisendavad andmemudelit kujule, mis on koodi genereerimiseks sobivam. Näiteks seovad need omavahel sisuliselt seotud DBC-faili osi, arvutavad abiväärtusi ning valivad signaalidele sobivad andmetüübid.

Backend'i ülesanne on genereerida lõplik kood ja testid. See ei lahenda DBC-faili semantilisi vastuolusid, vaid eeldab, et vaheesitus on valideeritud ja genereerimiseks ette valmistatud.

Joonis 2 demonstreerib projekti töötamise printsiipi.



Joonis 2. Projekti andme- ning töövoo diagramm.

5.1.1 Vaheesitus

Arhitektuuri keskseks komponendiks on vaheesitus, mida võib käsitleda kui süsteemi sisemist suhtlusprotokolli. *Frontend* teisendab DBC-faili vaheesituseks ning kõik järgnevad etapid kasutavad omavaheliseks andmevahetuseks vaheesitust. Seetõttu ei pea *middle-end* olema teadlik sellest, kuidas andmed algsest failist parsiti, ning *backend* ei pea teadma, milliseid teisendusi või valideerimisi eelnevates etappides rakendati. Teisisõnu vähendab vaheesitus süsteemi põhikomponentide omavahelisi sõltuvusi.

Vaheesituse olemasolu muudab projekti arhitektuuri modulaarseks. Igal põhiosal on selgelt piiritletud vastutus ning muudatused ühes osas ei mõjuta kogu süsteemi. Lisaks soodustab vaheesitus koodi taaskasutust. Näiteks juhul, kui tulevikus on vaja lisada tugi mõnele teisele väljundkeelele, piisab *backend*'i uue generaatori lisamisest. Olemasolevat *frontend*'i ja *middle-end*'i ei pea muutma.

5.1.2 *Frontend*

Parseriks kasutatakse can_dbc teeki, mis loeb DBC-faili ning kontrollib selle süntaksit. Autorite ülesandeks oli can_dbc'st tagastatav andmestruktuur konverteerida vaheesituseks.

5.1.3 *Middle-end*

Middle-end töötab konveierina, mis järjest jooksutab töötlussõlmi. Sõlmed täidavad kahte ülesannet: andmemudeli muutmine ja valideerimine. Transformatsioonisõlmed saavad sisendina muudetavat vaheesitust ning rikastavad või normaliseerivad seda. Valideerimissõlmed töötavad muutumatu vaheesitusega. Need otsivad probleeme andmemudelis ning salvestavad leitud vead hiljema emiteerimise jaoks. Neid põhimõtteid väljendavad liidesed joonisel 3.

```
// IR stands for Intermediate Representation
trait TransformationNode {
    fn transform(ir: IR);
}

trait CheckNode {
    fn check(ir: IR, diagnostics: Diagnostics);
}
```

Joonis 3. *Middle-end*'i kahe sõlmetüübi liidesed pseudokoodina.

Liideste kasutamine lihtsustab uute sõlmede lisamist. Konveier käivitab need ühise liidese kaudu, mistõttu ei pea uue sõlme lisamisel muutma konveieri loogikat. *Middle-end*'i töövoogu illustreerib joonis 4.

```
pub fn middle_end_run(ir: IR) -> Result {
    let diagnostics = [];
    for checker in check_nodes {
        checker.run(ir, diagnostics);
    }

    if diagnostics.has_errors() {
        diagnostics.emit();
        return Err;
    }

    transform_nodes.run(ir);
    return Ok;
}
```

Joonis 4. *Middle-end*'i töövoog pseudokoodina.

Liideste kasutamise tõttu ei ole konveieri jaoks oluline, kui palju on töötlussõlmi kokku. Tänu sellele saab arendaja loogiliselt erineva vastutusega sõlmi jagada mitme koodifaili vahel. Kahe konkreetse sõlme implementatsioonid on näidatud joonistel 5 ja 6.

```
struct ComputeSignalStartEndBits;

impl TransformationNode for ComputeSignalStartEndBits {
    fn transform(ir: Ir) {
        for message in ir.messages {
            for signal in message.signals {
                let (start, end) = start_end_bits(signal);
                signal.start_bit = start;
                signal.end_bit = end;
            }
        }
    }
}
```

Joonis 5. Transformatsioonisõlme näide pseudokoodina.

```
struct CheckUniqueMessageIds;

impl CheckNode for CheckUniqueMessageIds {
    fn check(ir: Ir, diagnostics:Diagnostics) {
        let set = Set();
        for message in ir.messages {
            if message.id in set {
                diagnostics.error(// error description...);
            } else {
                set.add(message.id);
            }
        }
    }
}
```

Joonis 6. Valideerimissõlme näide pseudokoodina.

Liideste definitsioonis kasutatakse vaheesitust. Järelikult on vaheesitus ka *middle-end*'i sõlmede suhtlusprotokoll. Selline disain aitab töötlussõlmi omavahel lahus hoida. Järgmise sõlme töö ei sõltu süntaktiliselt eelmise sõlme töö tulemusest. See võimaldab arendajal muuta sõlmede järjekorda lihtsalt.

See aga ei tähenda, et *middle-end*'i sõlmede käivitamise järjekord ei ole oluline. Kuigi sõlmed ei kutsu üksteist otse välja, võivad need sõltuda eelnevate sõlmede poolt vaheesitusse lisatud informatsioonist. Näiteks signaalide paigutuse kontrollimiseks peab enne olema teada, mis bitte iga signaal sõnumis kasutab.

See tähendab, et sõlmede vahel on olemas semantiline järjestussõltuvus. See on kompilaatori faasijärjestuse probleemi erijuhtum [23], kus sõlmede järjekord mõjutab väljundi korrektsust, mitte väljundi optimaalsust. Lahenduses ei proovita sõlmede järjekorda automaatselt tuletada ning optimeerida. Selle asemel on konveieri järjekord määratud käsitsi vastavalt sõlmede teadaolevatele sõltuvustele.

Selline lahendus on projekti jaoks piisav, sest koodigeneraatori eesmärk ei ole vaheesituse optimeerimine. Oluline on, et andmemudel läbiks kontrollid ja sisaldaks enne koodi genereerimist vajalikku informatsiooni. Fikseeritud järjekord muudab töövoo lihtsaks, deterministlikuks ja testitavaks.

5.1.4 *Backend*

Backend vastutab ettevalmistatud vaheesituse teisendamise eest sihtkeele koodiks. Sisendi semantilist parandamist selles etapis enam ei tehta. Selle asemel lähtutakse eeldusest, et *middle-end* on vaheesituse valideerinud ning õigele kujule teisendanud. Nii jääb koodigeneraatori roll kitsalt piiritletuks.

Praegune lahendus toetab kahte väljundkeelt: Rusti ja C++-i. Mõlemad generaatorid kasutavad sama vaheesitust, kuid koodi koostamise tehnika erineb sihtkeele võimaluste ja olemasolevate abiteekide tõttu.

Rusti generaatoris kasutatakse teeke quote, syn ja prettyplease. Generaator ei koosta Rusti lähtekoodi tekstina, vaid loob esmalt Rusti süntaksile vastava sümbolijada. Seejärel parsitakse need syn abil failiks ning vormindatakse prettyplease abil loetavaks koodiks. Selline lahendus vähendab sõnede ühendamisest tekkivate süntaktiliste vigade riski ning lihtsustab keerukamate konstruktsioonide genereerimist.

C++ jaoks ei ole samaväärset süntaksipuu põhist teeki. Seetõttu realiseeriti C++ koodi genereerimiseks eraldi abistruktuur, mis hoiab genereeritavat teksti puhvris ning pakub meetodeid ridade ja plokkide väljastamiseks. Lisaks vastutab see abistruktuur treppimise eest. Nii ei pea generaator jälgima, mitu tühikut rea ette tuleb lisada.

Bittide lugemise ja kirjutamise lahendus erineb samuti sihtkeeliti. Rusti generaator kasutab `bitvec` teeki, mis pakub meetodeid bitivahemike käsitlemiseks ning väärtuste lugemiseks ja

kirjutamiseks eri baidijärjestustes. C++ standardteegis samaväärset kõrgtasemelist vahendit ei ole. Seetõttu genereeritakse C++ väljundisse eraldi abifunktsioonid, mis oskavad sõnumi baitide massiivist vajalikku bitivahemikku välja lugeda, sinna uut väärtust kirjutada ning multipleksitud harude puhul bitivahemikke ühest andmeväljast teise kopeerida. Need abifunktsioonid eristavad väikese ja suure baidijärjestusega signaale ning peidavad bitinihete ja maskide kasutamise ülejäänud genereeritud koodi eest.

Lisaks põhikoodile saab *backend* genereerida ka teste. Rusti puhul lisatakse need *#[cfg(test)]* testimoodulina. C++ puhul genereeritakse päisefaili eraldi testinimeruum koos *run_all* funktsiooniga. Need testid kontrollivad genereeritud konstruktorite, lugemis- ja muutmismeetodite, kodeerimise, dekodeerimise ning veajuhtude käitumist.

Rusti väljundi puhul oli eeskuju olemas. Kuna C++-i jaoks olemasolevat koodigeneraatorit ei olnud, pidid autorid ka genereeritava API struktuuri nullist välja töötama. See kujundati koostöös tellijaga ning eesmärk oli hoida see võimalikult sarnane Rusti väljundiga.

5.2 Genereeritud koodi disain

Genereeritud koodi disaini kujundamisel kaaluti erinevaid viise, kuidas CAN-sõnumit rakenduskoodis esitada. Kõige lihtsam lähenemine on genereerida iga sõnumi jaoks struktuur, mille väljad vastavad signaalide füüsilistele väärtustele. Lugemis- ja muutmismeetodid saavad töötada füüsilise väärtustega ning toorväärtusteks kodeerimine ja andmeväljaks konverteerimine toimub vaid *encode* meetodis. Seda lähenemist demonstreeritakse joonisel 7.

```
struct Message {
    speed: f32; // Signals are stored as physical values
    // Other signals...
}

impl Message {
    pub const LENGTH = 8; // CAN message data field length

    // Get speed signal's physical value
    pub fn speed() -> f32 {
        return self.speed;
    }

    // Set new physical value for speed signal
    pub fn set_speed(speed: f32) -> Result {
        if speed < speed_min or speed > speed_max {
            return Err;
        }
        self.speed = speed;
        return Ok;
    }

    // Convert struct representation to CAN message data field
    pub fn encode() -> [u8; LENGTH] {
        let bytes = [u8; LENGTH];
        let raw = (self.speed - offset) / factor;
        bytes.bits[speed_start..speed_end].write_as_raw_type(raw);
        // Convert other signals to raw values and
        // store them as bits in the data field
        return bytes;
    }

    // Other methods...
}
```

Joonis 7. Lihtne CAN-sõnumi disain pseudokoodina.

Analüüsi käigus ilmnes aga, et selline esitus ei sobi mälupiirangutega keskkondadesse (sardsüsteemid). Füüsilise väärtuse tüüp võib olla oluliselt suurem kui selle toorväärtuse esitus CAN-sõnumis. Näiteks 8-bitise toorväärtusega signaali füüsiline väärtus võib olla 32-bitine ujukomaarv. Lisaks võivad struktuuride joondamine (*alignment*) ja täidistus (*padding*) suurendada objekti tegelikku mälumahtu [24].

Seetõttu valiti lahendus, kus iga genereeritud sõnumi objekt hoiab sisemiselt DBC-failis

määratud pikkusega baitide massiivi. Lugemismeetodid dekodeerivad signaali väärtuse baitide jadast ning muutmismeetodid kodeerivad kasutaja antud füüsilise väärtuse tagasi samasse jadasse. Selline lahendus vastab paremini CAN-sõnumi tegelikule esitusviisile, vähendab mälukasutust ning lihtsustab sõnumi saatmist, sest andmeväli on objektis alati olemas. Seda lähenemist demonstreerib joonis 8.

```
struct Message {
    data: [u8; LENGTH]; // Message stores the data field
}

impl Message {
    pub const LENGTH = 8; // CAN message data field length

    // Get speed signal's physical value
    pub fn speed() -> f32 {
        let raw = self.data.bits[speed_start..speed_end].read_as_raw_type();
        let physical = raw * factor + offset;
        return physical;
    }

    // Set new physical value
    pub fn set_speed(speed: f32) -> Result {
        if speed < speed_min or speed > speed_max {
            return Err;
        }
        let raw = (speed - offset) / factor;
        bytes.bits[speed_start..speed_end].write_as_raw_type(raw);
        return Ok;
    }

    // Convert struct representation to CAN message data field
    pub fn encode() -> [u8; LENGTH] {
        return self.data;
    }

    // Other methods...
}
```

Joonis 8. Kasutusel olev CAN-sõnumi disain pseudokoodina.

Selle lähenemise puudus on see, et signaali väärtuste lugemisel ja muutmisel tuleb teha teisendusi iga väljakutse korral. Projekti kontekstis peetakse seda kompromissi põhjendatuks, kuna sihtkeskkonnas on mälukasutus oluline ning CAN-sõnumi baitkuju

hoidmine muudab objekti esituse kompaktseks.

Multipleksitud sõnumite puhul esitatakse erinevad multipleksitud alamobjektid eraldi tüüpidena. Igal alamobjektil on oma meetodid konstrueerimiseks ning signaalide väärtuste lugemiseks ja muutmiseks. Ülemobjekt sisaldab alati ühte kehtivat alamobjekti ning ei ole teadlik selle signaalidest. Selline disain väljendab tüübisüsteemi tasemel, et korraga saab aktiivne olla ainult üks multipleksitud haru [25]. Seda on näha joonisel 9.

```
struct Multiplexed0 { data: [u8; LENGTH]; }
struct Multiplexed1 { data: [u8; LENGTH]; }
// Methods are not shown in this code snippet

enum Multiplexed {
    V0(Multiplexed0),
    V1(Multiplexed1),
} // Algebraic sum type

struct MainMessage { data: [u8; LENGTH]; }
impl MainMessage {
    pub fn set_mux0(variant: Multiplexed0) {
        // Write variant signals' raw values to wrapper message data field
        // Change multiplexor value to 0.
    }
    pub fn set_mux1(variant: Multiplexed1) {
        // Write variant signals' raw values to wrapper message data field
        // Change multiplexor value to 1.
    }
    pub fn mux() -> Multiplexed {
        // Get multiplexor value
        // Depending on the multiplexor value construct
        // either Multiplexed0 or Multiplexed1
        // Return it wrapped in the Multiplexed enum
    }
    // Other methods...
}
```

Joonis 9. Multipleksitud CAN-sõnumi disain pseudokoodina.

5.3 Testidega väljundi korrektsuse tagamine

Lahenduse korrektset tööd kontrollitakse testidega. Selles projektis kasutatakse kahte tüüpi teste: hetktõmmisteste ja integratsiooniteste. Eraldi üksusteste projektis ei kasutata. Kuna lahenduse struktuur muutus arenduse käigus sageli, looksid üksustestid suure hoolduskulu

ega kontrolliks otseselt kasutajale olulist väljundit. Peamine rõhk on generaatori väljundi testimisel.

5.3.1 Hetktõmmistestid

Hetktõmmistest salvestab programmi väljundi ning võrdleb järgmisel käivitamisel uut väljundit viimase salvestatud väljundiga [12]. Kui väljund muutub, test ebaõnnestub ning arendaja peab otsustama, kas uus väljund on korrektne ja see tuleb salvestada uue oodatud väljundina või on tegemist ootamatu veaga.

Koodigeneraatorite puhul on hetktõmmistest eriti kasulik järgnevatel põhjustel:

1. See aitab tuvastada tahtmatuid regressioone. Näiteks võib ühe koodigeneraatori osa parandamine kogemata muuta mõne teise osa väljundit. Kui see muudab tööriista väljundit, näitab test kohe erinevust.
2. Suure väljundi puhul võib mõni muudatus käsitsi kontrollides märkamata jääda. Kui väljundit võrreldakse hetktõmmistestiga automaatselt, ei ole võimalust muudatust mitte tähele panna.
3. See muudab genereeritud koodi API evolutsiooni nähtavaks. `git diff`-i abil on lihtne näha, milline muudatus koodigeneraatoris põhjustab muudatuse genereeritud koodis.

Hetktõmmistestid tagavad, et generaator annab stabiilset väljundit, kuid ei garanteeri, et väljund on semantiliselt õige ja töötab korrektselt.

5.3.2 Integratsioonitestid

Integratsioonitestid kontrollivad, kas mitu süsteemi osa töötavad koos õigesti [10]. Selle projekti puhul kontrollivad need tervet koodigenereerimise ahelat.

Integratsiooniteste saab liigitada mitmesse klassi:

1. Parsimise testid. Test õnnestub, kui süntaktiliste vigadega failis tuvastatakse vead.
2. Semantilise analüüsi testid. Test õnnestub, kui semantiliste probleemidega failis tuvastatakse vead.
3. Koodi kompileerimise testid. Test õnnestub, kui vigadeta DBC-failist genereeritud

kood kompileerub.

4. Genereeritud koodi testid. Test õnnestub, kui genereeritud kood läbib selle jaoks loodud testid.

Parsimise ja semantilise analüüsi testid puudutavad vigaseid sisendfaile. Kui DBC-failis esineb viga, tuvastab tööriist vea, lõpetab töö ja ei genereeri koodi.

Kompileerimise testid kontrollivad olukorda, kus sisendfail on edukalt parsitud ja semantiliselt valideeritud. Sellisel juhul genereeritud Rusti või C++-i kood kindlasti kompileerub.

Genereeritud koodi testid kontrollivad genereeritud koodi käitumist. Kui genereeritud kood kompileerub, läbib see genereeritud testid.

Integratsioonitestide jaoks kasutatakse kokku 86 DBC-faili. Testfailid pärinevad OxiBUSi avatud lähtekoodiga koodihoidlast ning nende kasutamine on lubatud [26].

Nende testidega teevad autorid kindlaks, et tööriist töötab kogu ahela ulatuses ootuspäraselt ning valideerivad genereeritud koodi käitumise korrektsust.

5.4 Projekti seis

Lõputöö tulemusena valmis avatud lähtekoodiga DBC-põhine koodigeneraator. Tööriista nimeks on valitud dbc-codegen2, et rõhutada, et olemasolevast dbc-codegen projektist on tehtud samm edasi. Lahendus on kõigile kättesaadav GitHubis ja arhiveeritud kujul Software Heritage Archive'is seisuga 15.05.2026.

Projekti töökindlust toetab GitHub Actionsi töövoog, mis käivitub iga kord, kui muudatus lükatakse *main*-harusse. Töövoog koosneb integratsioonitestidest ja hetktõmmistestidest. Selline automaatne kontroll aitab vähendada riski, et vead jäävad arenduse käigus märkamata.

Valminud generaator vastab kõigile varem püstitatud nõuetele. See tuvastab sisendis vigu ning genereerib koodi, millel on korrektsuse garantii. Lisaks varem sõnastatud nõuetele lisandusid arendustöö käigus tellija tagasiside põhjal mõned lisanõuded:

1. Korraga saab genereerida koodi mitme DBC-faili põhjal.

2. Genereeritud koodis ei esine dubleeritud loendeid. Kui eri signaalidega on seotud sisuliselt samad loendid, genereeritakse mitme loendi asemel üks loend, mida taaskasutavad kõik sellega seotud signaalid.
3. Loenditele genereeritakse abimeetodid, mis lihtsustavad arvuliste väärtuste ja loendivariantide vahelist teisendamist. Sõltuvalt generaatorile antud konfiguratsioonist võivad need meetodid käituda erinevalt. Näiteks võib DBC-failis olla loend, mis kirjeldab seost 0 OFF ja 1 ON. Ühe konfiguratsiooni korral annab abimeetod vea, kui sellele antakse väärtus 2 ja palutakse see loendivariandiks teisendada. Teise konfiguratsiooni korral töötab sama kood veata ning tagastab vaikimisi `Other` variandi.
4. Signaalide puhul, mille minimaalseks ja maksimaalseks füüsiliseks väärtuseks on määratud 0, saab muutmismeetoditest eemaldada vahemikukontrolli *if*-laused.
5. Kui lahendust kasutatakse teegina, saab genereeritud koodi lisada kasutaja määratud koodi. Näiteks Rusti puhul võimaldab see lisada sõnumi tüübi ette *derive*-atribuudi, mis implementeerib sellele sõnumile automaatselt teatud käitumise.
6. Lisaks DBC-failis olevatele kommentaaridele kirjutab generaator sõnumitele ja signaalide meetoditele kommentaare nende definitsioonide põhjal. Nii on genereeritud koodis näha objektide originaalnimed ning muu info, mis on kasulik koodi mõistmiseks ja visuaalseks kontrolliks.

Ka need nõuded on edukalt realiseeritud. Projekti koodiridade arv seisuga 11.05.2026 on ligikaudu 6600.

6 Tulemuste analüüs

Loodud tööriista praktilist väärtust hinnatakse võrdluses dbc-codegeniga ning valdkonna spetsialistidelt kogutud tagasiside põhjal.

6.1 Võrdlus dbc-codegeniga

Kuna tellija kasutab hetkel dbc-codegeni, siis võrreldakse loodud lahendust just sellega.

Võrreldakse järgmisi aspekte:

- tööriista kasutusmugavus ja olemasolev funktsionaalsus;
- ohutuse garantii, mida tööriist annab koodi genereerimise ajal;
- genereeritud koodi mugavus ja ohutus;
- tööriista ja genereeritud koodi jõudlus.

6.1.1 Kasutamine ja funktsionaalsus

Mõlemat tööriista saab kasutada nii käsurea tööriistana kui ka teegina. Neid on lihtne alla laadida Cargo paketihalduri kaudu ning mõlemad pakuvad põhjalikku dokumentatsiooni koos koodinäidetega. Genereeritud kood toetab *no_std* keskkonda.

dbc-codegenil on piiranguid. See ei toeta korraga mitme DBC-faili põhjal koodi genereerimist, mistõttu tuleb kasutajal genereerida mitu koodifaili eraldi ning hiljem käsitsi korduvad osad eemaldada. Kui seda mitte teha, põhjustavad korduvad osad nimekonflikte.

dbc-codegen võimaldab lisada genereeritavasse koodi kasutaja määratud koodilõike, kuid see võimalus on piiratud. Kasutaja ei saa vabalt määrata, millist koodi ja millistesse kohtadesse lisada.

Lisaks ei oska dbc-codegen tuvastada, et kaks loendit on sisuliselt võrdsed, mistõttu genereerib see palju korduvat koodi.

Autorite projekt lahendab need probleemid ning pakub lisaks muid varem kirjeldatud võimalusi.

6.1.2 Ohutus genereerimise ajal

Selles kontekstis tähendab ohutus, et DBC-failis semantilise vea tuvastamisel ei genereerita vigast koodi.

Võrdluseks viiakse läbi katse. Kasutatakse 27 DBC-faili, milles esineb erinevaid semantilisi probleeme. Mõlemale tööriistale antakse need failid ükshaaval sisendina ning vaadatakse, kuidas tööriistad käituvad.

Kahel korral 27-st annab dbc-codegen veateate, et midagi läks valesti ning koodi ei ole võimalik genereerida. Veateade ei selgita probleemi põhjust ega viita sellele, et DBC-failis on sisuline viga. 12 korral genereerib dbc-codegen koodi, mida ei saa hiljem kompileerida. Ülejäänud 13 korral genereerib dbc-codegen koodi, mis kompileerub, kuid selle käitumine käitlusajal ei vasta kasutaja ootustele.

Autorite tööriist ei genereeri koodi kordagi. Selle asemel annab see kasutajale teada, et DBC-failis on probleem, milles probleem seisneb ning millise sõnumi või signaaliga on see seotud. Kui DBC-failis on mitu viga, tuvastab tööriist kõik vead ega lõpeta tööd esimese vea leidmisel. Lisaks veateadetele annab tööriist ka hoiatusi. Näiteks hoiatab see olukorras, kus sõnumi definitsioonist ilmneb, et andmeväljas on bitte, mida ei kasuta ükski signaal.

Seisuga 11.05.2026 tuvastab dbc-codegen2 20 erinevat veatüüpi ning hoiatab kolme potentsiaalselt ebasoovitava olukorra eest.

6.1.3 Genereeritud kood

Struktuuri mõttes genereerivad mõlemad tööriistad sarnast koodi. Genereeritud kood sisaldab sõnumitele vastavaid objekte, nende meetodite implementatsioone, signaalide väärtuste loendeid, loendite abimeetodeid, veatüüpide loendit ning ülemise taseme loendit, mille variantideks on kõik sõnumid. Ülemise taseme loendit kasutatakse geneerilise sõnumitüübina.

dbc-codegen genereerib iga sõnumi signaali jaoks konstandid, mis näitavad signaali

füüsilise väärtuse miinimumi ja maksimumi. Iga objekti definitsiooni ja implementatsiooni ette lisab dbc-codegen konfiguratsiooniridu, mis summutavad koodi stiilihoiatusi. Lisaks implementeerib tööriist iga sõnumi jaoks *embedded_can::Frame* liidese.

Pärast arutelu tellijaga selgub, et konstantide genereerimine ei ole selle projekti jaoks vajalik. Ka stiilihoiatuste summutamist ei pea tegema genereeritud koodi sees, vaid kasutaja saab seda teha üks kord enne genereeritud koodi lisamist oma projekti. Tellija sõnul *embedded_can::Frame* liidese implementatsiooni praktiline väärtus on väike.

Tööriistade väljundi mahu võrdlemiseks tehakse katse. Selleks kasutatakse sünteetiliselt koostatud DBC-faili, milles on 80 erinevat sõnumit ning mille pikkus on ligikaudu 1200 rida. dbc-codegeni väljundi pikkus on ligikaudu 43 tuhat koodirida. Autorite lahendus genereerib ilma testideta ligikaudu 32 tuhat koodirida. See tähendab, et dbc-codegen2 genereerib ligikaudu 25% vähem koodi. Mõlemad tööriistad kasutavad koodi vormindamiseks sama teeki, seega ei tulene erinevus vormindamisest.

API ja implementatsiooni vaatenurgast genereerivad mõlemad tööriistad sarnast koodi, kuid on olemas olulised erinevused:

1. Uuel tööriistal on genereeritud objektide nimed loetavamad. Nimekonfliktide lahendamise algoritmi tõttu on genereeritud nimed üldjuhul lühemad kui vanal tööriistal. Koodi kirjutamisel peab kasutaja sisestama ka vähem sümboleid.
2. Kuigi dbc-codegen genereerib signaalide väärtuste jaoks loendeid, ei genereeri see sõnumite konstruktoreid, mis võtaksid argumentidena loendivariante. Genereeritud konstruktorid töötavad ainult arvuliste väärtustega. See ei ole kasutajale mugav, sest sunnib kasutajat tähendusliku nimetuse asemel meeles pidama arvulist väärtust. Seevastu genereerib dbc-codegen2 konstruktoreid, mis kasutavad loenditüüpe.
3. Vana lahendus genereerib signaalide jaoks täiendavaid lugemismeetodeid, mille nimi lõpeb *"_raw"* sufiksiga. Sufiks jätab mulje, et meetod tagastab signaali toorväärtuse. Tegelikult tagastab *"_raw"* variant sõnumi andmeväljast dekodeeritud signaali füüsilise väärtuse ning tavaline signaali lugemine kutsub selle variandi omakorda välja. Sellise vahesammu eesmärk ei ole selge. Autorite lahendus seda liigset koodi ei genereeri.
4. Multipleksitud sõnumite alamobjektide konstruktorid ei võta dbc-codegenis ar-

gumente. Teisisõnu loob konstruktor tühja alamobjekti. Hiljem saab signaalide väärtusi salvestada muutmismeetoditega. See teeb kehvemaks kasutuskogemust, sest ühe valiidse objekti loomiseks peab kasutaja välja kutsuma mitu meetodit. Lisaks on selline lahendus potentsiaalselt ohtlik. Tühi alamobjekt ei pruugi esitada õiget sõnumi andmevälja olekut. Kui kasutaja unustab muutmismeetoditega õiged väärtused salvestada, võib see põhjustada vea rakenduse loogikas. Nendel põhjustel genereerib autorite tööriist konstruktoreid, mis nõuavad alati vajalikke argumente. Nii jõustatakse koodi korrektsust API tasemel.

5. Multipleksitud sõnumite puhul genereerib dbc-codegen koodi, mis lubab kasutajal muuta multiplekseri väärtust. See on ohtlik operatsioon, sest eri multiplekseri väärtuste puhul aktiivsed signaalid võivad jagada samu sõnumi andmevälja bitte. Seega võib valiidse sõnumi loomine ja sellele järgnev multiplekseri väärtuse muutmine viia sõnumi vastuolulisse olekusse. Seetõttu ei lase uue tööriistaga genereeritud kood multiplekseri väärtust otse muuta. Kui varianti on vaja muuta, tuleb selleks kasutada muutmismeetodeid, mis võtavad argumendina terve alamobjekti, salvestavad selle ning muudavad ka multiplekseri väärtust.
6. Multipleksitud sõnumite alamobjektide muutmismeetodid on dbc-codegenis ohtlikud. Uue alamobjekti andmeväljale, mida tahetakse salvestada, rakendatakse loogiline VÕI-tehe sõnumis juba oleva andmevälja suhtes. See operatsioon ei asenda vana väärtust uue sisestatud väärtusega, vaid moodustab vana ja uue väärtuse põhjal kolmanda väärtuse. Selline käitumine ei vasta kasutaja ootustele. dbc-codegen2 genereerib koodi, mis kirjutab iga signaali vana väärtuse uue väärtusega üle.

6.1.4 Jõudlus

Tööriistade jõudluse mõõtmiseks kasutatakse sama ligikaudu 1200-realist DBC-faili, mida kasutati väljundkoodi pikkuse mõõtmiseks. Mõlemat tööriista käivitatakse 50 korda ning sisendiks antakse sama fail. Tööaja mõõtmiseks kasutatakse *time* käsurea programmi. Keskmiselt lõpetab dbc-codegen töö 0,19 sekundiga. dbc-codegen2 töötab keskmiselt 0,17 sekundit, kui koodi jaoks teste ei genereerita, ning 0,31 sekundit, kui testid genereeritakse. Kui testid genereeritakse, on väljundi koodiridade arv ligikaudu kaks korda suurem kui ilma testideta. Need tulemused näitavad, et tööriistadel on sarnane jõudlus, mis on piiratud mitte nende loogika, vaid faili kirjutamise operatsiooni kiirusega.

Genereeritud koodi jõudluse võrdlemiseks viiakse läbi teine katse. Mõlema tööriistaga genereeritakse sama sisendi põhjal kood, luuakse sama sõnum ning mõõdetakse, kui kaua võtavad aega signaali väärtuse salvestamine ja tagastamine ning multipleksitud alamobjekti salvestamine ja tagastamine. Iga operatsiooni mõõdetakse tuhat korda ning tulemuste põhjal arvutatakse keskmine operatsiooni kestus. Sellest piisab praktilise kasutatavuse kontrollimiseks.

Katset tehakse MacBook Pro arvutiga, millel on M2 Pro kiip ja 16 gigabaiti muutmälu. Katse tulemused on esitatud tabelis 1.

Tabel 1. Genereeritud koodi operatsioonide keskmine täitmisaeg.

Operatsioon	**dbc-codegen**	**dbc-codegen2**
Signaali muutmismeetod	4,79 ns	4,83 ns
Signaali lugemismeetod	2,00 ns	1,96 ns
Multipleksitud alamobjekti muutmismeetod	43,88 ns	109,37 ns
Multipleksitud alamobjekti lugemismeetod	29,67 ns	26,79 ns

Tabelist on näha, et kolmes kategoorias töötab mõlema tööriista genereeritud kood sisuliselt sama kiirusega. Multipleksitud alamobjekti salvestamine on aga originaalsel lahendusel ligikaudu 2,5 korda kiirem. See tulemus on ootuspärane, sest nagu eespool mainitud, ei taga dbc-codegen selle operatsiooni puhul koodi käitumise korrektsust. Selle operatsiooni implementatsioon on dbc-codegenis lihtsam ning see põhjustabki jõudluse erinevust.

Mõõtmised tehti ühel arvutil, mistõttu võivad tulemused erineda teistsuguse riistvara või operatsioonisüsteemi korral. Samuti võivad lühikese kestusega operatsioonide mõõtmist mõjutada süsteemi taustaprotsessid.

6.2 Valdkonna spetsialistide tagasiside

Lisaks autorite tehtud valideerimisele koguti tööriista kohta tagasisidet nii tellijalt kui ka laiemalt arendajakogukonnalt.

Tellija andis arenduse jooksul regulaarselt tagasisidet. Tagasiside oli valdavalt positiivne ning tellija hinnangul muutis tööriist DBC-failidega töötamise oluliselt mugavamaks. Eriti tõsteti esile tööriista võimet tuvastada sisendfailides vigu enne koodi genereerimist. See

aitas leida probleeme tellija DBC-failides ja koodis.

Laiema kogukonna tagasiside kogumiseks koostati küsimustik ning saadeti olemasolevate tööriistade aktiivsetele arendajatele ning jagati Redditi kanalites *r/embedded*, *r/rust* ja *r/cpp*. Küsimustik sisaldas praktilisi ülesandeid, kus vastajal paluti genereerida DBC-faili põhjal koodi, kasutada genereeritud API-t, töötada multipleksitud sõnumitega ning proovida tööriista vigaste sisendfailidega. Küsiti hinnanguid tööriista kasutusmugavuse, genereeritud koodi loetavuse ja API praktilisuse kohta.

Kavandatud küsimustike kaudu tagasisidet ei laekunud, mistõttu ei saa nende põhjal teha üldistatavaid järeldusi tööriista kasutusmugavuse kohta. Projekti väline valideerimine tugines tellija jooksvale tagasisidele.

6.3 Järeldused

Esitatud tulemuste põhjal võib järeldada, et arendatud tööriist täidab sellele seatud ootusi ning parandab mitut dbc-codegeni olulist puudust. Kõige olulisem edasiminek on sisendi valideerimine.

Samuti genereerib uus lahendus koodi, mis on mitmes olukorras ohutum ja kasutajale selgem. Eriti oluline on see juhtudel, kus vale kasutus võib tekitada sisuliselt vigase sõnumi. Autorite tööriist jõustab genereeritud koodi korrektset kasutamist API tasemel.

Katsete põhjal ei kaasne korrektsusgarantiidega olulist jõudluskaotust. Erandiks on multipleksitud alamobjekti salvestamine, kus suurem korrektsus tähendab keerukamat implementatsiooni ja seetõttu aeglasemat tööd. See on põhjendatud kompromiss.

Lisaks tehnilistele tulemustele kinnitab töö praktilist väärtust ka tellija tagasiside. Töö käigus on tellija andnud järjepidevalt positiivset tagasisidet ning plaanib autorite arendatud tööriista kasutusele võtta.

7 Tagasivaade ja tulevik

Arendusprotsessi käigus tehtud valikud ei pruugi projekti lõpuks enam olla kõige sobivamad. Seetõttu sisaldab lahendus osi, mida saab edasises arenduses lihtsustada või paremini eraldada.

7.1 Märkused arhitektuurile

Vaheesituse ülesehitust on võimalik lihtsustada. Töö käigus lisandusid tellijalt nõuded, millest hiljem loobuti. Nende nõuete toetamiseks tehti vaheesituses muudatusi, kuid arenduse hilisemas etapis ei jäänud piisavalt aega nende eemaldamiseks. Seetõttu sisaldab vaheesitus osi, mis ei ole lõpliku lahenduse seisukohalt vajalikud ja muudavad selle keerukamaks.

Tööriistal on kindel arhitektuur, mille järgi generaatorid vastutavad ainult koodi genereerimise eest. Lahenduses leidub siiski üks koht, kus seda põhimõtet ei järgita täielikult. Praegu käivad generaatorid läbi iga sõnumi signaalid, et tuvastada, kas sõnum on multipleksitud. Selline loogika ei kuulu generaatori vastutusalasse, vaid peaks paiknema mõnes *middle-end*'i transformatsioonisõlmes. Sõnumite klassifitseerimine toimub küll ühes valideerimissõlmes, kuid selle tulemust ei salvestata vaheesitusse. Seetõttu kordavad generaatorid sama loogikat. Selle parandamiseks tuleb muuta vaheesitust ning eemaldada generaatoritest sinna mittekuuluv loogika.

7.2 Tehniline võlg

Rakenduse sisemises loogikas esineb koodi dubleerimist. Arendusaja piirangute tõttu sisaldavad *middle-end*'i sõlmed kopeeritud koodi, mille saaks asendada taaskasutatavate abifunktsioonidega. Nende osade ümberkorraldamine vähendaks koodibaasi ligikaudu 300 rea võrra.

Rusti koodigeneraator kasutab testide jaoks pseudojuhuslike väärtuste genereerimisel

arbitrary teeki [27]. See teek võimaldab genereerida juhuslikke täisarve etteantud vahemikus, kuid ei paku sama funktsionaalsust ujukomaarvude jaoks [28]. Seetõttu kasutatakse praegu lahendust, kus genereeritakse juhuslik ujukomaarv ning piiratakse see signaali minimaalse ja maksimaalse väärtuse vahele. See töötab, kuid ei pruugi tagada piisavat väärtuste varieeruvust. Selle probleemi lahendamiseks tuleks kasutada mõnda teist teeki.

Töö käigus ilmus uus versioon can_dbc teegist, mida kasutatakse DBC-failide parsimiseks. Uus versioon sisaldab tagasiühilduvust mittetoetavaid muudatusi: uued andmetüübid ning muudetud API [29]. Autoritel ei olnud piisavalt aega, et võtta kasutusele parseri uut versiooni.

Lisaks tuvastati töö käigus uusi probleeme, mis võivad DBC failis esineda, kuid mille jaoks ei jõutud valideerimissõlmi implementeerida.

7.3 Uus funktsionaalsus

Uue funktsionaalsusena oleks kasulik lisada laiendatud multipleksimise (*extended multiplexing*) tugi [9]. See võimaldaks kirjeldada sõnumeid, kus on mitu multiplekser-signaali ning üks signaal sõltub mitmest multiplekserist. Praegu toetab tööriist ühe multiplekseri olemasolu. Selle funktsionaalsuse realiseerimist raskendab täpse dokumentatsiooni puudumine. Ei ole selge, kuidas laiendatud multipleksimist DBC-failides kirjeldatakse ja kuidas see täpselt toimima peaks. Lähtepunktina võiks uurida, millise sisendi põhjal ja millist koodi genereerib cantools, sest see tööriist väidetavalt toetab laiendatud multipleksimist [30].

Tellija avaldas soovi, et oleks võimalik genereerida C++-i koodi ka nii, et iga sõnum paikneb eraldi päisefailis. Seda funktsionaalsust võiks lisada valikulise käsurea lipuna.

Samuti oleks kasulik toetada lisaks DBC-le ka mõnda muud sisendformaati, näiteks KCD-d [30], [31]. Rakenduse arhitektuuri tõttu on selle formaadi toe lisamiseks vaja realiseerida ainult uus *frontend*.

7.4 Kasutuselevõtt ja tulevik

Tellijal on plaanis võtta autorite tööriist kasutusele järgmisel kvartalil.

Autorite soov on saada loodud projekt OxiBUSi GitHubi organisatsiooni. Autorid on võtnud ühendust OxiBUSi liikmetega, et arutada seda võimalust, kuid otsust pole veel langetatud. Kui õnnestub liituda OxiBUSiga, võib loodud tööriist tulevikus täielikult asendada dbc-codegeni.

8 Kokkuvõte

Töö eesmärgiks oli kirjutada tellija soovidele ja nõuetele vastav tööriist, mis suudab genereerida DBC-failide põhjal Rusti ja C++-i koodi.

Töö tulemusena valmis avatud lähtekoodiga tööriist dbc-codegen2. Tööriist on disainitud sarnaselt kompilaatorile: see parsib DBC-faili, muudab selle vaheesituseks, valideerib ning teisendab vaheesitust ja seejärel genereerib koodi.

Valideerimiseks võrreldi loodud lahendust olemasoleva dbc-codegen tööriistaga. Tulemused näitasid, et uuel lahendusel on mitu eelist: see tuvastab probleeme DBC-failides ja genereerib selgemat koodi, mis pakub paremaid korrektsusgarantiisid peaaegu ilma jõudluskaota.

Loodud tööriist ei lahenda ainult tellija konkreetseid probleeme, vaid on kasutatav ka üldotstarbelise DBC-põhise koodigeneraatorina. Seetõttu pakub lahendus väärtust ka teistele arendajatele, kes töötavad CAN-võrkude ning DBC-failidega.

Valminud tööriist vastab tellija vajadustele ning see on plaanis võtta kasutusele järgmisel kvartalil. Lõputöö eesmärk on saavutatud.

Kasutatud kirjandus

[1] „Controller Area Network (CAN) Overview.“ National Instruments. Vaadatud: 1. oktoober 2025. [Võrgumaterjal.] Kättesaadav: https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/controller-area-network--can--overview.html

[2] „CAN Specification Version 2.0.“ University of California, Riverside. Vaadatud: 1. oktoober 2025. [Võrgumaterjal.] Kättesaadav: http://esd.cs.ucr.edu/webres/can20.pdf

[3] „CAN DBC File Database Introduction.“ CSS Electronics. Vaadatud: 1. oktoober 2025. [Võrgumaterjal.] Kättesaadav: https://www.csselectronics.com/pages/can-dbc-file-database-intro

[4] „DBC Files - University of Waterloo Robotics Design Team.“ University of Waterloo Robotics Design Team. Vaadatud: 10. mai 2026. [Võrgumaterjal.] Kättesaadav: https://uwaterloo.atlassian.net/wiki/spaces/UWRT/pages/33955647206/DBC+Files

[5] „An Introduction to J1939 and DBC files.“ Kvaser. Vaadatud: 10. mai 2026. [Võrgumaterjal.] Kättesaadav: https://kvaser.com/developer-blog/an-introduction-j1939-and-dbc-files/

[6] K. Czarnecki ja U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Reading, MA, USA: Addison-Wesley, 2000.

[7] „CANdb++ Admin Fact Sheet.“ Vector Informatik GmbH. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://cdn.vector.com/cms/content/products/candb/Docs/CANdb_Admin_FactSheet_EN.pdf

[8] D. Murzinov, „awesome-canbus.“ GitHub. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://github.com/iDoka/awesome-canbus

[9] „Extended Signal Multiplexing in DBC Databases.“ Vector Informatik GmbH. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://cdn.vector.com/cms/content/know-how/_application-notes/AN-ION-1-0521_Extended_Signal_Multiplexing.pdf

[10] I. Sommerville, *Software Engineering*, 10. väljaanne. Boston, MA, USA: Pearson, 2016.

[11] „The Rust CLI Book.“ rust-cli. Vaadatud: 1. oktoober 2025. [Võrgumaterjal.] Kättesaadav: https://rust-cli.github.io/book/index.html

[12] „insta Documentation.“ docs.rs. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://docs.rs/insta

[13] A. V. Aho, M. S. Lam, R. Sethi ja J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2. väljaanne. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[14] E. Moqvist, „cantools.“ GitHub. Vaadatud: 1. oktoober 2025. [Võrgumaterjal.] Kättesaadav: https://github.com/cantools/cantools

[15] J. Weiß, „dbcppp.“ GitHub. Vaadatud: 1. oktoober 2025. [Võrgumaterjal.] Kättesaadav: https://github.com/xR3b0rn/dbcppp

[16] J. Whitley, „can_dbc_loader.“ GitHub. Vaadatud: 1. oktoober 2025. [Võrgumaterjal.] Kättesaadav: https://github.com/astuff/can_dbc_loader

[17] Y. Astrakhan, „OxiBUS.“ GitHub. Vaadatud: 13. mai 2026. [Võrgumaterjal.] Kättesaadav: https://github.com/oxibus

[18] Y. Astrakhan, „dbc-codegen.“ GitHub. Vaadatud: 10. mai 2026. [Võrgumaterjal.] Kättesaadav: https://github.com/oxibus/dbc-codegen

[19] M. Fairman, „dbc-data.“ docs.rs. Vaadatud: 1. oktoober 2025. [Võrgumaterjal.] Kättesaadav: https://docs.rs/dbc-data/latest/dbc_data/

[20] „The Embedded Rust Book: A no_std Rust Environment.“ rust-lang.org. Vaadatud: 10. mai 2026. [Võrgumaterjal.] Kättesaadav: https://doc.rust-lang.org/stable/embedded-book/intro/no-std.html

[21] „Working Draft, Standard for Programming Language C++: Implementation Compliance.“ ISO/IEC JTC 1/SC 22/WG 21. Vaadatud: 10. mai 2026. [Võrgumaterjal.] Kättesaadav: https://eel.is/c++draft/intro.compliance

[22] „N4950: Working Draft, Standard for Programming Language C++: C++ Headers for Freestanding Implementations.“ ISO/IEC JTC 1/SC 22/WG 21. Vaadatud: 10. mai 2026. [Võrgumaterjal.] Kättesaadav: https://timsong-cpp.github.io/cppwp/n4950/organization#compliance

[23] P. A. Kulkarni, D. B. Whalley, G. S. Tyson ja J. W. Davidson, „Exhaustive Optimization Phase Order Space Exploration“, teoses *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006)*, IEEE Computer Society, 2006, lk. 306–318. DOI: 10.1109/CGO.2006.15.

[24] „GNU C Manual.“ Free Software Foundation. Vaadatud: 18. mai 2026. [Võrgumaterjal.] Kättesaadav: https://www.gnu.org/software/c-intro-and-ref/manual/html_node/index.html

[25] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[26] Y. Astrakhan, „OxiBUS Shared Test Files.“ GitHub. Vaadatud: 18. mai 2026. [Võrgumaterjal.] Kättesaadav: https://github.com/oxibus/shared-test-files

[27] „arbitrary Documentation.“ docs.rs. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://docs.rs/arbitrary

[28] P. Hertleif, „Best Way to Generate Float Values in Range.“ GitHub. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://github.com/rust-fuzz/arbitrary/issues/77

[29] „can-dbc Documentation.“ docs.rs. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://docs.rs/can-dbc

[30] „cantools Documentation.“ Read the Docs. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://cantools.readthedocs.io/

[31] „Supported File Formats.“ Read the Docs. Vaadatud: 14. mai 2026. [Võrgumaterjal.] Kättesaadav: https://canmatrix.readthedocs.io/en/latest/formats.html

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Meie, Pavel Pavliv ja Rasmus Raasuke

1. Anname Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "DBC-põhise Rusti ja C++-i koodigeneraatori arendamine", mille juhendaja on Tavo Annus
    1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
    1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Oleme teadlikud, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autoritele.
3. Kinnitame, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

01.06.2026

[1] Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – DBC-faili näidis

```
VERSION ""

NS_ :
    BA_
    BA_DEF_
    BA_DEF_DEF_
    BA_DEF_DEF_REL_
    BA_DEF_REL_
    BA_DEF_SGTYPE_
    BA_REL_
    BA_SGTYPE_
    BO_TX_BU_
    BU_BO_REL_
    BU_EV_REL_
    BU_SG_REL_
    CAT_
    CAT_DEF_
    CM_
    ENVVAR_DATA_
    EV_DATA_
    FILTER
    NS_DESC_
    SGTYPE_
    SGTYPE_VAL_
    SG_MUL_VAL_
    SIGTYPE_VALTYPE_
    SIG_GROUP_
    SIG_TYPE_REF_
    SIG_VALTYPE_
    VAL_
    VAL_TABLE_

BS_:

BU_: DBG DRIVER IO MOTOR SENSOR

BO_ 100 DRIVER_HEARTBEAT: 1 DRIVER
 SG_ DRIVER_HEARTBEAT_cmd : 0|8@1+ (1,0) [0|255] "" SENSOR,MOTOR

BO_ 500 IO_DEBUG: 4 IO
```

```
 SG_ IO_DEBUG_test_unsigned : 0|8@1+ (1,0) [0|255] "" DBG
 SG_ IO_DEBUG_test_enum : 8|8@1+ (1,0) [0|255] "" DBG
 SG_ IO_DEBUG_test_signed : 16|8@1- (1,0) [-128|127] "" DBG
 SG_ IO_DEBUG_test_float : 24|8@1+ (0.5,0) [0|127.5] "" DBG

BO_ 101 MOTOR_CMD: 1 DRIVER
 SG_ MOTOR_CMD_steer : 0|4@1+ (1,-5) [-5|5] "" MOTOR
 SG_ MOTOR_CMD_drive : 4|4@1+ (1,0) [0|9] "" MOTOR

BO_ 400 MOTOR_STATUS: 3 MOTOR
 SG_ MOTOR_STATUS_wheel_error : 0|1@1+ (1,0) [0|1] "" DRIVER,IO
 SG_ MOTOR_STATUS_speed_kph : 8|16@1+ (0.001,0) [0|65] "kph" DRIVER,IO

BO_ 200 SENSOR_SONARS: 8 SENSOR
 SG_ SENSOR_SONARS_mux M : 0|4@1+ (1,0) [0|15] "" DRIVER,IO
 SG_ SENSOR_SONARS_err_count : 4|12@1+ (1,0) [0|4095] "" DRIVER,IO
 SG_ SENSOR_SONARS_left m0 : 16|12@1+ (0.1,0) [0|409.5] "" DRIVER,IO
 SG_ SENSOR_SONARS_middle m0 : 28|12@1+ (0.1,0) [0|409.5] "" DRIVER,IO
 SG_ SENSOR_SONARS_right m0 : 40|12@1+ (0.1,0) [0|409.5] "" DRIVER,IO
 SG_ SENSOR_SONARS_rear m0 : 52|12@1+ (0.1,0) [0|409.5] "" DRIVER,IO
 SG_ SENSOR_SONARS_no_filt_left m1 : 16|12@1+ (0.1,0) [0|409.5] "" DBG
 SG_ SENSOR_SONARS_no_filt_middle m1 : 28|12@1+ (0.1,0) [0|409.5] "" DBG
 SG_ SENSOR_SONARS_no_filt_right m1 : 40|12@1+ (0.1,0) [0|409.5] "" DBG
 SG_ SENSOR_SONARS_no_filt_rear m1 : 52|12@1+ (0.1,0) [0|409.5] "" DBG

CM_ BU_ DRIVER "The driver controller driving the car";
CM_ BU_ MOTOR "The motor controller of the car";
CM_ BU_ SENSOR "The sensor controller of the car";
CM_ BO_ 100 "Sync message used to synchronize the controllers";

BA_DEF_ "BusType" STRING ;
BA_DEF_ BO_ "GenMsgCycleTime" INT 0 0;
BA_DEF_ SG_ "FieldType" STRING ;

BA_DEF_DEF_ "BusType" "CAN";
BA_DEF_DEF_ "FieldType" "";
BA_DEF_DEF_ "GenMsgCycleTime" 0;

BA_ "GenMsgCycleTime" BO_ 100 1000;
BA_ "GenMsgCycleTime" BO_ 500 100;
BA_ "GenMsgCycleTime" BO_ 101 100;
BA_ "GenMsgCycleTime" BO_ 400 100;
BA_ "GenMsgCycleTime" BO_ 200 100;
BA_ "FieldType" SG_ 100 DRIVER_HEARTBEAT_cmd "DRIVER_HEARTBEAT_cmd";
BA_ "FieldType" SG_ 500 IO_DEBUG_test_enum "IO_DEBUG_test_enum";
```

```
VAL_ 100 DRIVER_HEARTBEAT_cmd 2 "DRIVER_HEARTBEAT_cmd_REBOOT" 1
    "DRIVER_HEARTBEAT_cmd_SYNC" 0 "DRIVER_HEARTBEAT_cmd_NOOP" ;
VAL_ 500 IO_DEBUG_test_enum 2 "IO_DEBUG_test_enum_two" 1
    "IO_DEBUG_test_enum_one" ;
```

Joonis 10. DBC-faili näidis.