Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra kybernetiky

DIPLOMOVÁ PRÁCE

Metody optimalizace generovaného kódu pro software automatické převodovky

Plzeň, 2019 Jan Raděj

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: **2018/2019**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan RADĚJ**
Osobní číslo:	**A16N0146P**
Studijní program:	**N3918 Aplikované vědy a informatika**
Studijní obor:	**Kybernetika a řídicí technika**
Název tématu:	**Metody optimalizace generovaného kódu pro software automatické převodovky**
Zadávající katedra:	**Katedra kybernetiky**

Zásady pro vypracování:

1. Popis vhodných metod měření systémových prostředků (RAM,ROM,CPU runtime) řídící jednotky převodovky (TCU).
2. Analýza generování C/C++ kódu [1][2] ze Simulink [3] modelu pomocí programu dSpace TargetLink.
3. Měření náročnosti jednotlivých částí aplikace pro řízení automatické převodovky.
4. Návrh pravidel za účelem generování efektivního kódu pro snížení nároků na systémové prostředky.
5. Úprava modelu v Simulinku na základě navržených pravidel a vygenerování nové verze softwaru.
6. Srovnání využití systémových prostředků původního SW a SW po úpravách.

Rozsah grafických prací: **dle potřeby**

Rozsah kvalifikační práce: **40-50 stránek A4**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

[1] HEROUT, Pavel. Učebnice jazyka C 1. díl. České Budějovice: KOPP, 2016. ISBN 978-80-7232-383-8.
[2] DUFOUR, Emmanuel. MISRA C++:2008 Guidelines for the use of the C++ language in critical systems. Warwickshire, UK: Hobbs the Printers, 2008. ISBN 978-1-906400-04-0.
[3] COLGREN, Richard. Basic MATLAB, Simulink and Stateflow. Lawrence, Kansas: American Institute of Aeronautic and Astronautics, 2007. ISBN 978-1-56347-838-3.

Vedoucí diplomové práce: **Ing. Pavel Balda, Ph.D.**
Výzkumný program 1

Konzultant diplomové práce: **Ing. Martin Fajfr**
ZF Engineering Plzeň s.r.o.

Datum zadání diplomové práce: **1. října 2018**

Termín odevzdání diplomové práce: **25. května 2019**

Doc. Dr. Ing. Vlasta Radová
děkanka

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
L.S.

Prof. Ing. Josef Psutka, CSc.
vedoucí katedry

V Plzni dne 1. října 2018

PROHLÁŠENÍ

Předkládám tímto k posouzení diplomovou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejíž úplný seznam je součástí.

V Plzni dne .....

.....
vlastnoruční podpis

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat panu Ing. Pavlu Baldovi, Ph.D., svému vedoucímu diplomové práce, za vstřícnost při konzultacích a cenné rady při psaní práce. Dále bych chtěl poděkovat společnosti ZF Engineering Plzeň za možnost zde zpracovávat diplomovou práci, poskytnutí veškerých materiálů a nástrojů potřebných k vypracování a vstřícný přístup. Jmenovitě bych chtěl zde poděkovat Ing. Martinu Fajfrovi za osobní přístup, cenné rady a pravidelné konzultace při vypracovávání práce a také Ing. Ondřeji Váchalovi za podporu a konzultace při psaní dokumentace k práci. Na závěr bych rád také poděkoval své přítelkyni za podporu při sepisování práce.

Abstrakt

Tato práce se zabývá návrhem pravidel modelování za účelem generování efektivnějšího kódu pro řídicí jednotku automatické převodovky do nákladních vozidel. První část práce obsahuje stručný popis použitých softwarových nástrojů a pravidel programování používaných v automobilovém průmyslu. V další části je provedena analýza hardwarových prostředků řídicí jednotky a je popsán způsob generování zdrojového kódu. Následující kapitola práce je věnována způsobům měření zátěže procesoru, zajištění validace měření a jeho následnému zpracování. V závěrečné části jsou navrženy optimalizační metody modelování, ve smyslu nejnižší zátěže procesoru, dále je provedena jejich analýza a zhodnocení.

Klíčová slova: TargetLink, Simulink, MATLAB, auto–kód, optimalizace, mikro kontrolér, automatická převodovka, datafield parametr.

Abstract

The aim of this thesis is designing modeling rules for generating more effective code for electronic control unit for a truck automatic transmission. The first part of the thesis contains a short description of used software tools and there is also description of programming rules for automotive. In the next part an analysis of hardware resources of electronic control unit was made and there is a description of how to generate code. The next chapters of thesis describes ways how to measure runtime load on the processor, how to ensure measurement validity and how to process measurements. In the final part optimize methods are designed for modeling in the terms of the lowest ECU load and analyzis of the methods is performed.

Key words: TargetLink, Simulink, MATLAB, auto–code, optimization, micro controller, automatic transmission, datafield parameter.

Obsah

Kapitola 1

Úvod

1.1 Hlavní myšlenka

Cílem práce je navrhnout metody modelování v Simulinku, které by, po automatickém vygenerování zdrojového kódu, vedly k nižším nárokům na výkon použitého hardwaru. Automatické generování zdrojového kódu přímo z prostředí modelu je technika, která znatelně urychluje vývoj softwaru. Umožňuje funkčnímu vývojáři produkovat rovnou zdrojový kód a tím obejít nutnost ručního psaní softwaru. Tímto způsobem lze zkrátit vývojový cyklus průmyslové aplikace. Problém tohoto řešení je, že odebírá možnost přímého zásahu do zdrojového kódu a funkční vývojář je plně odkázán na způsob, jakým nástroj pro automatické generování přetvoří jeho myšlenky v modelu ve zdrojový kód. Přestože algoritmy těchto generátorů jsou na velmi dobré úrovni, ne vždy kód vygenerovaný s jejich pomocí je plně efektivní.

Část diplomové práce je věnována analýze vybraných případů generování kódu a tomu, jak návrh modelu ovlivní výsledný vygenerovaný kód. S pomocí této analýzy jsou pak navrhnuty metody modelování pro optimalizaci vygenerovaného kódu.

1.2 Motivace

Navrhovaná řešení jsou prováděna a testována na softwaru elektronické řídicí jednotky automatických převodovek, které se používají v nákladní dopravě. V automobilovém průmyslu a ve velkosériové průmyslové výrobě obecně je kladen velký důraz na cenu jednotlivých použitých součástek, protože i relativně malé zdražení jednoho dílu se v absolutním měřítku velkosériové produkce může velice prodražit. Vzhledem k tomu, že cena hardwarových komponent se také dílem promítá do výrobní ceny produktu, je tomu podřízen i návrh řídicí aplikace. Při softwarovém vývoji v automobilovém průmyslu je otázka spotřeby systémových prostředků významným tématem a jakékoli navýšení nároků na systémové prostředky musí být zdůvodněno. Hledání metod modelování, které by optimalizovaly běh výsledné aplikace, je tedy velmi žádoucí a každé prokázané zlepšení se velice cenění.

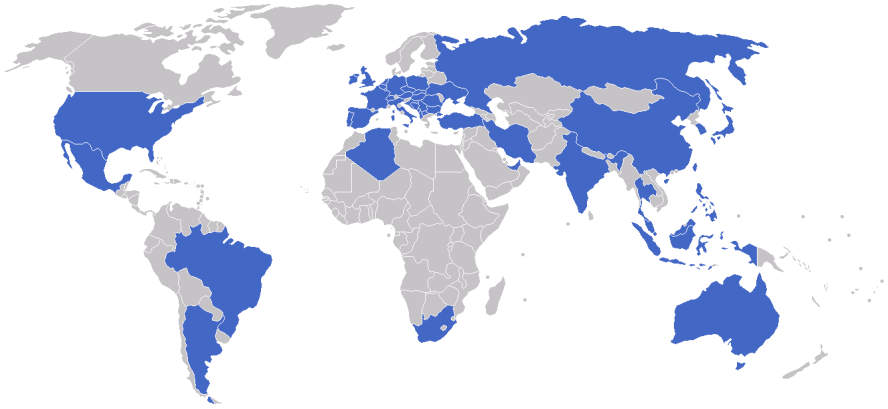
1.3 Vybraná společnost

Tato diplomová práce byla vypracována ve spolupráci se společností ZF Engineering Plzeň, ze strany společnosti byly pro práci poskytnuty softwarové a hardwarové nástroje i potřebné know–how. V následující části bude firma představena.

1.3.1 O společnosti

Společnost ZF Friedrichshafen AG, označována jako ZF Group, je jedním z předních dodavatelů technologií a systémů pro osobní i komerční dopravu a průmyslových technologií. Mezi nejznámější produkty patří automatické planetové převodovky pro osobní automobily, či automatické převodovky pro kamionovou dopravu. Dále však vyvíjí systémy pohonného ústrojí, e–mobility a s tím spojené technologie pro novou generaci dopravy.

Firma ZF[1] byla založena roku 1915 hrabětem von Zeppelinem v německém Friedrichshafenu na břehu Bodamského jezera. Během minulého století se společnost rozrostla a získala ve svém odvětví významnou pozici. V roce 2018 měla kolem 150 000 zaměstnanců v 230 pobočkách a ve 40 zemích po celém světě.



Obrázek 1.1: Ilustrační mapa rozmístění ZF Group ve světě.

1.3.2 ZF v České republice

Společnost ZF má v České republice momentálně (začátek roku 2019) 10 poboček, jedná se o 6 výrobních závodů a 4 vývojová centra.

Výrobní závody	Vývojová centra
ZF Frýdlant	ZF Openmatics Plzeň
ZF Jablonec nad Nisou	ZF Engineering Plzeň
ZF Klášterec nad Ohří	ZF Jablonec nad Nisou
ZF Staňkov	ZF Zlín
ZF Stará Boleslav	
ZF Žatec	

Tabulka 1.1: Přehled poboček ZF Group v České republice

1.3.3 ZF Engineering Plzeň

Vývojové centrum ZF Engineering Plzeň bylo založena již roku 2002. Zdejší pobočka v dnešní době poskytuje zákazníkům služby v oblasti elektroniky, vývoje a testování softwaru, či výpočtů a mechatroniky.

V současné době[2] je zde zaměstnáno přibližně 500 pracovníků.

[1] z německého Zahnradfabrik Friedrichshafen – firma na ozubená kola

[2] začátek roku 2019

Kapitola 2

Použité standardy a SW nástroje

2.1 Standardy v programování

Standardy v programování nebo–li programovací konvence jsou pokyny, které jsou doporučeny při psaní kódu dodržovat. Jedná se především o styl a metody programování, používání (nebo nepoužívání) určitých funkcionalit programovacího jazyka, architektonicky osvědčených postupů, apod.. Tyto konvence mohou mít různé cíle, na jednu stranu mohou přispívat k lepší čitelnosti kódu, k jeho kvalitě[1] nebo bezpečnosti[2].

Právě na kvalitu a bezpečnost kódu je při vývoji softwaru pro automobilový průmysl kladen největší důraz, následuje několik nejužívanějších standardů v automobilovém průmyslu.

2.1.1 MISRA, MISRA C a MISRA C++

2.1.1.1 Historie

Název pochází z anglického spojení ***M**otor **I**dustry **S**oftware **R**eliability **A**ssociation.* Jedná se o projekt, který začal v 90. letech ve Spojeném království jako součást programu *SafeIT*, který se zabýval financováním projektů zaměřených na bezpečnost elektronických systémů.

Projekt *MISRA* měl sestavit pokyny a doporučení pro tvorbu softwaru používaného v automobilovém průmyslu. Po skončení financování programem *SafeIT* se členové *MISRA* rozhodli pro pokračování neformální spolupráce. Současnými členy projektu *MISRA* jsou společnosti

- Bentley Motors
- Delphi Diesel Systems
- Ford Motor Company Ltd
- HORIBA MIRA Ltd
- Jaguar Land Rover
- Protean Electric Ltd

[1] Kvalitou psaného kódu se může rozumět například konzistentní formátování, které přispívá k lepší čitelnosti kódu tak, aby byl snadno srozumitelný i pro další vývojáře. Konzistentní software umožňuje lepší orientaci v kódu i pro samotného autora.

[2] U bezpečnostních konvencí jsou stanoveny pravidla vylučující používání potenciálně nebezpečných funkcionalit, ve kterých se může snadno udělat chyba, či dojít k zacyklení programu

- Ricardo plc
- The University of Leeds
- Visteon Engineering Services Ltd
- ZF TRW

[5]

2.1.1.2 MISRA C

MISRA C se stala prvním z produktů této neoficiální spolupráce. Společnosti Ford a Rover vytvořily podmnožinu jazyka C, jejíž používání vedlo ke zlepšení kvality softwaru a jeho bezpečnosti. Vytvoření podmnožiny programovacího jazyka mělo přímou návaznost na původní dokument *MISRA* z roku 1994 tak, aby splňoval požadavky na danou úroveň bezpečnosti. První vydání *MISRA C* bylo vydáno roku 1998.

MISRA C získala během krátké doby poměrně hodně uživatelů a následně i zpětné vazby na zpracování pravidel. Z toho důvodu byla v roce 2004 vydána přepracovaná verze *MISRA C:2004* nebo také jinak označována jako *MISRA C2*.

V dnešní době existuje už třetí vydání *MISRA C* vydané v roce 2012, ve kterém je zapracována další zpětná vazba od uživatelů a byla zde také vytvořena křížová reference pro normu *ISO 26262* [2],[5].

2.1.1.3 MISRA C++

S ohledem na rostoucí popularitu programovacího jazyka *C++*, který se ocitl podobné pozici jako kdysi před tím jazyk *C*, kterému byla vyčítána nevhodnost použití v kritických systémech.

Z toho důvodu vydala společnost *MISRA* v roce 2008 soubor pokynů a pravidel pro používání jazyka *C++* v kritických systémech. Dokument byl publikován 5. června 2008 pod názvem *MISRA C++* [2],[5].

2.1.2 AUTOSAR

Jedná se o partnerství výrobců a dodavatelů pro automobilový průmysl. Jedná se především o výrobce automobilů a dodavatele elektroniky a softwaru, který se v nich používá. Cílem této aliance je vytvořit otevřený průmyslový standard pro architekturu v automobilovém softwaru. Jedná o části jako je základní infrastruktura pro správu funkcí nebo standardní softwarové moduly.

Společenství bylo založeno roku 2002 firmami BMW, Bosch, Continental, Daimler-Chrysler a Volkswagen, později se připojil i Siemens VDO. Z počátku se jednalo především o rozvržení prvotního plánu a strategie *AUTOSAR*u. O rok později byly představeny hlavní cíle a spuštěna první fáze projektu, kdy bylo zapotřebí vytvoření a schválení plánu vývoje.

Dnes má *AUTOSAR* za sebou již třetí fázi vývoje a ke konci roku 2016 se společenství rozrostlo celkem na 191 partnerů, ze kterých se 112 podílí na vývoji softwarových produktů.

Mezi hlavní produkty patří:

- **Aplikační rozhraní**
  Jedná se o rozsáhlou standardizovanou sadu aplikačních rozhraní z pohledu syntaxe a sémantiky programovacího jazyka pro následující části vozidel

  - motor a hnací ústrojí,
  - převodovka,
  - řízení podvozku,
  - bezpečnost a bezpečnost chodců.

- **Klasická platforma**
  Jedná se o typ abstraktní architektury, která je rozdělená do tří vrstev
  - aplikace, kde tato vrstva je nezávislá na hardwaru a komunikaci s hardwarem zprostředkovává vrstva basic software dále jen *BSW*.
  - Základní software (*BSW*), jak už název napovídá, jedná se o základní software, který zprostředkovává komunikaci mezi aplikací a hardwarovými ovládacími prvky.
  - Runtime prostředí (anglicky Runtime Environment dále jen *RTE*), představuje rozhraní, kde spolu komunikuje aplikační vrstva a vrstva základních ovladačů.

- **Adaptivní platforma**
  Tato platforma je orientovaná pro vysoce výkonné jednotky, které jsou použity pro náročné operace jako je například autonomní řízení. Ve srovnání s klasickou platformou nabízí vrstva *RTE* možnost propojení služeb a klientů během běhu [6].

2.1.3 ISO 26262

Cílem tohoto standardu je snížení rizik spojených s používáním elektronických systémů v automobilech. Potřeba této normy začala koncem 80. let, kdy se v moderních autech začalo objevovat větší množství elektronických systémů. Tyto systémy mezi sebou komunikují a mohou mít zásadní vliv na řízení automobilu, pokud selže jeden systém, jeho porucha se může promítnout i do dalších závažnějších oblastí. Dnešní moderní automobily disponují funkcemi jako adaptivní tempomat, systém nouzového brzdění, automatická převodovka, adaptivní světlomety a podobně.

Pokud by došlo k nějaké poruše, může se stát, že v noci přestanou světlomety na nepřehledné cestě svítit, nebo vozidlo začne v plné rychlosti na dálnici nouzově brzdit.

Norma ISO 26262 byla vytvořena, aby bylo eliminováno riziko, že nastanou případy tohoto druhu. Orientuje se na na splnění cílů funkční bezpečnosti. Řešením prostřednictvím eliminace všech nepřijatelných rizik a hrozeb, přičemž norma využívá rizikový přístup v průběhu celého životního cyklu bezpečnosti výrobku: od základního konceptu, přes návrh, vývoj až po výrobu a provoz.

Základní koncepce říká, že pokud selže jakákoli komponenta, nesmí způsobit ohrožení pro cestující ani osobám mimo vozidlo [7].

2.2 Použité softwarové nástroje

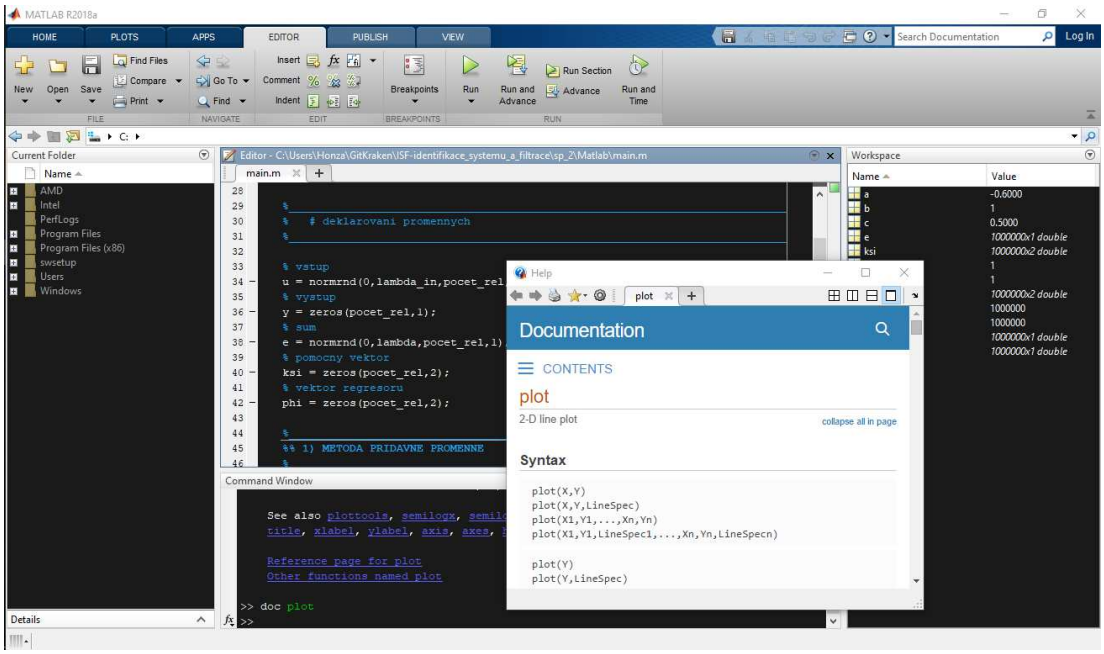
K vypracování práce bylo zapotřebí specializovaných softwarových nástrojů, které umožňovali simulovat hardware ve smyčce (anglicky hardware in the loop dále bude použita zkratka HIL simulace), modelové simulace, generování kódu nebo měřit zprávy, které přicházejí za běhu, z elektronické řídicí jednotky. V této části budou všechny použité nástroje v krátkosti popsány.

2.2.1 Mathworks

Společnost Mathworks se zaměřuje především na vývoj matematického a výpočetního softwaru zaměřeného především na vědce a inženýry. Ke svým nástrojům MATLAB, Simulink nebo Polyspace poskytuje společnost podporu v podobě rozsáhlé knihovny funkcí a nápovědy doplněné o množství příkladů na svých stránkách [10].

2.2.1.1 MATLAB

Jedná se o programovací jazyk 4. generace, který má své vlastní uživatelské rozhraní, na které je navázána řada dalších funkcionalit. Primárně byl vyvinut pro počítání složitých numerických výpočtů a snadnému programování algoritmů zpracovávajících velké množství dat, ale dnes se jedná o plný programovací jazyk podporující objektově orientované programování. Postupem času byly vyvinuty nové knihovny a rozšíření specializující se na různá inženýrská odvětví jako jsou řídicí systémy, zpracování signálu či obrazu, statistika či finanční matematika.



Obrázek 2.1: Ukázka programovacího jazyka MATLAB a jeho grafického rozhraní s nápovědou.

Kromě zpracování dat umožňuje i jejich snadnou vizualizaci, nastavení zobrazení a jejich snadný export. Od verze 2016b umožňuje tvorbu takzvaných živých skriptů. Jedná se o dokumenty s formátovaným textem proložené funkčním kódem s výpočty a případně vizualizovanými daty. U těchto dokumentů je pak umožněn export do standardního formátu pdf [10].

2.2.1.2 Simulink

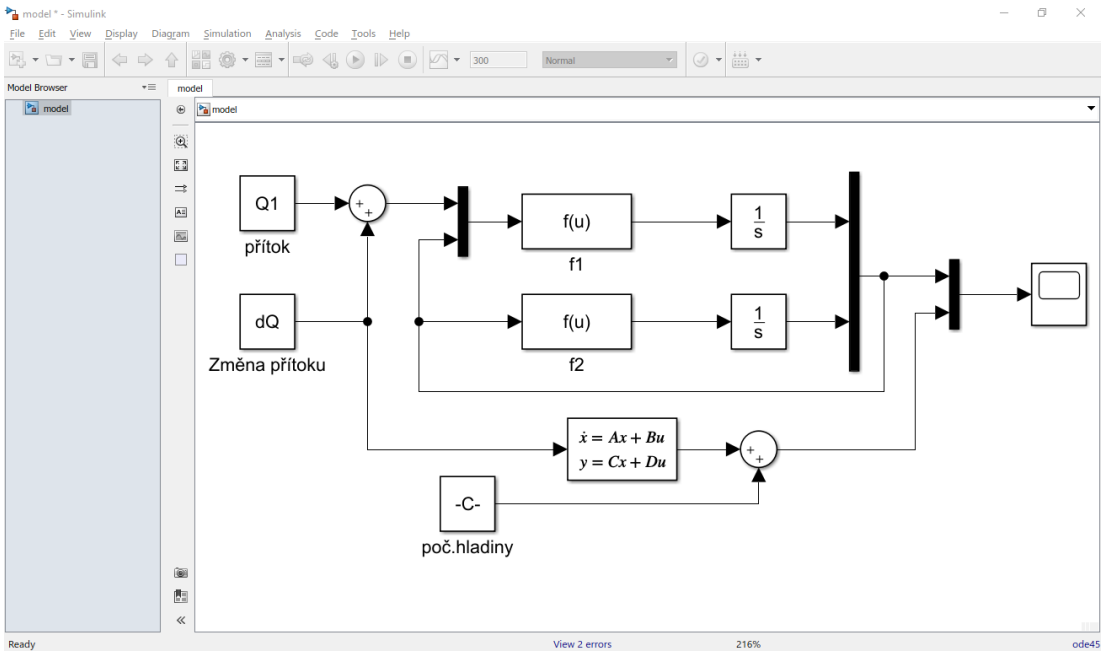
Jedná se o prostředí určené ke grafickému návrhu matematických modelů a jejich simulaci. V dnešní době se velmi využívá i pro Model–Based Design vývoj[3]. Nástroj

[3]Model–Based Design zkráceně MBD je metoda pro rychlý a efektivní návrh dynamického systému. Při vývojovém procesu je navržen matematický model požadovaného systému, který je průběžně testován pomocí simulací a následně zpřesňován [8].

podporuje návrh na úrovni systému, simulaci, průběžné testování i ověřování vestavěných systémů.

Vzhledem k tomu, že se jedná o nadstavbu výpočetního nástroje *MATLAB*, umožňuje kromě vlastní knihovny bloků, používat i plné provázání s *MATLAB*em. Tato provázanost spočívá především v možnosti začlenit algoritmy napsané přímo v jazyce *MATLAB* do simulace, či načítat a exportovat výsledky simulace zpět do *MATLAB*u, pro jejich další zpracování.

Nástroj obsahuje vlastní knihovnu bloků, pomocí kterých je možno modelovat spojité i diskrétní systémy a navrhovat regulační smyčky, dále je možné si psát vlastní „S–funkce“ a „M-funkce“ v jazyce *MATLAB* [10]. Na obrázku 2.2 je znázorněn příklad návrhu modelu v prostředí *Simulink*.



Obrázek 2.2: Ukázka modelování v nástroji *Simulink*.

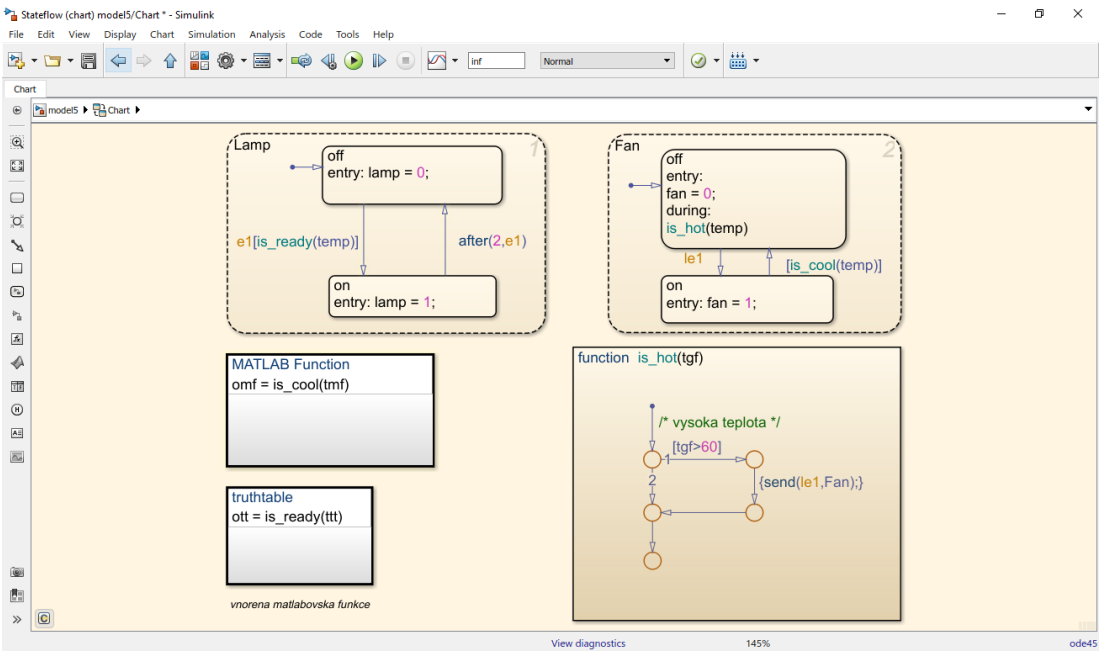
2.2.1.3 Stateflow

Nástroj *Simulink* dále obsahuje nadstavbu *Stateflow*, což je grafický jazyk, který umožňuje modelování pomocí stavových diagramů, diagramů přechodu a stavových a pravdivostních tabulek. Pomocí toho nástroje lze modelovat sekvenční či kombinatorickou logiku. Animace přechodu mezi diagramy pomáhá ke snazšímu ladění modelu.

Stateflow umožňuje navrhovat a modelovat dohledové řízení, plánování úloh a komunikační rozhraní. Hlavními nástroji jazyka jsou stavové bločky, logické uzly a směrové šipky mezi nimi.

Úlohu lze modelovat pomocí stavových bloků, které si drží aktivitu a vykonávají funkcionalitu v nich napsanou do doby, než není splněna výstupní podmínka přechodu. Další způsobem práce se *Stateflow* diagramy je pomocí podmínek přechodu a logických uzlů nebo kombinací obou způsobů [10]. Příklad možného diagramu je na následujícím obrázku 2.3.

Je možné si povšimnout, že stavy jsou v pravém horním rohu očíslované, tyto čísla určují pořadí, ve kterém se budou stavy vykonávat v případě paralelní simulace. Podobně jsou očíslované i šipky vedoucí z jednoho uzlu ve funkci *is_hot(tgf)*, tato čísla opět určují pořadí, ve kterém se budou přechody vykonávat.



Obrázek 2.3: Ukázka diagramu *Stateflow* s několika stavy, podmínkami přechodu mezi nimi a funkcemi přímo *Statflow*, tak i napsané v prostředí *MATLAB*.

2.2.2 dSPACE

Společnost dSpace je německá firma se sídlem ve městě Paderborn, která se zabývá vytvářením nástrojů zaměřených na vývoj a testování nových technologií převážně v automobilovém průmyslu už od roku 1988.

Hlavním portfóliem této firmy jsou produkty, které umožňují vyvíjet a testovat nová softwarová řešení pro elektronické řídicí jednotky a mechatronické řízení v automobilové i letecké dopravě.

Firma vyvíjí mnoho hardwarových i softwarových nástrojů, které cílí především na podporu vývoje v automobilovém a leteckém průmyslu, jejich řešení usnadňuje proces vývoje a testování správné funkčnosti zařízení.

Mezi jejich nejznámější produkty patří HIL simulátory. Jedná se o zařízení, na kterém je možné testovat chování řídicí jednotky, například převodovky, bez nutnosti mít fyzicky zařízení k dispozici, či zbytek auta. HIL je schopný zpracovat výstupní signály z elektronické řídící jednotky (v angličtině se používá výraz Electronic Control Unit dále jen ECU) a zároveň generovat všechny potřebné vstupy tak, jako by se jednalo o skutečný automobil na silnici.

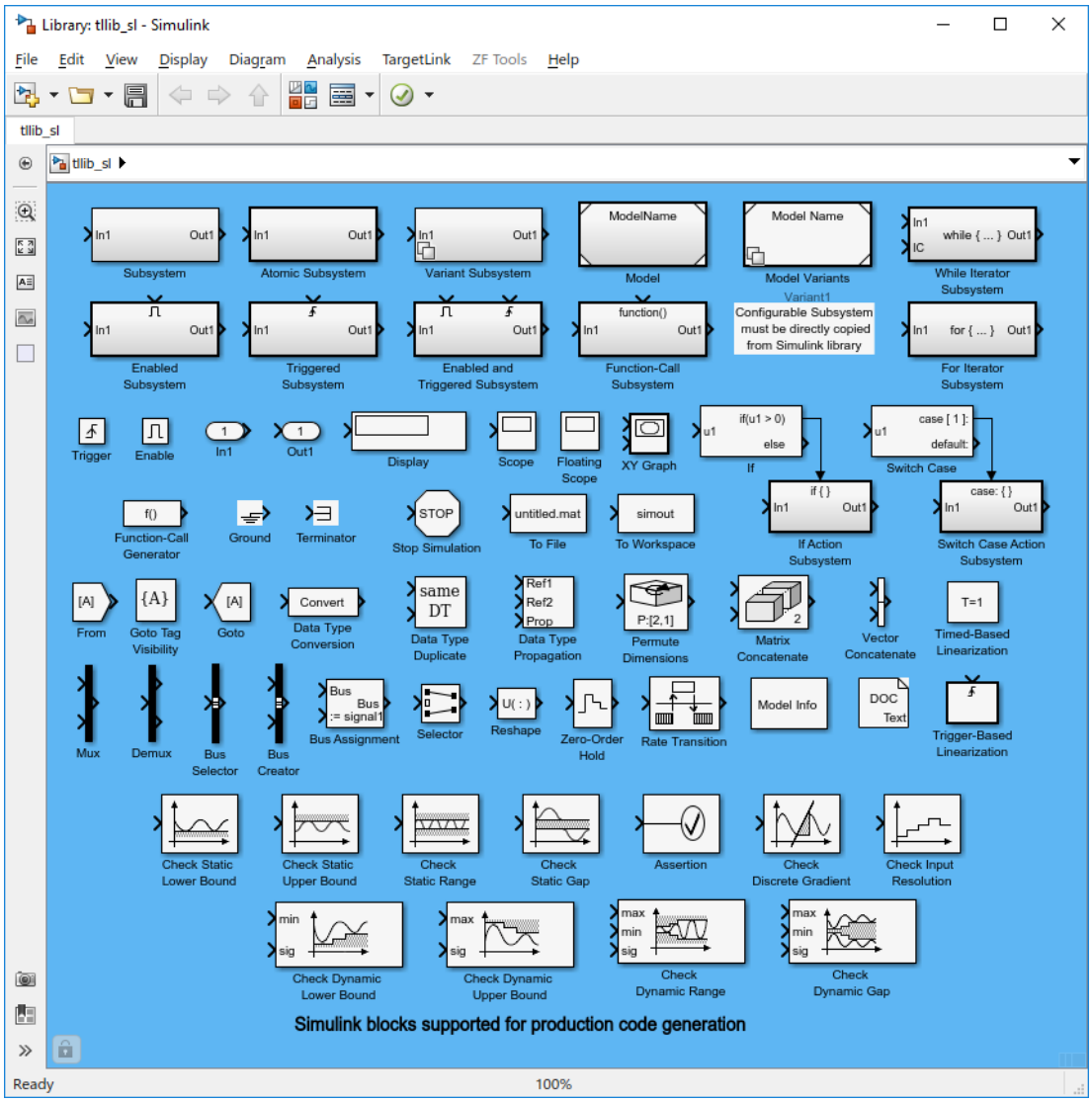
Dalším nástrojem vyvíjeným firmou v Paderbornu, který je možno zmínit, může být například ControlDesk, software poskytující uživatelské rozhraní pro ovládání řídicí jednotky připojené k HIL simulátoru.

V neposlední řadě právě TargetLink, který slouží k automatickému generování zdrojového kódu z modelu v Simulinku nebo Stateflow.

2.2.2.1 TargetLink

Jedná se o nástroj, který slouží ke generování produkčního zdrojového kódu přímo z prostředí *Simulink*. Cílem je snížení času potřebného k vývoji aplikace. Tento software je dlouhodobě vyvíjen se zaměřením na vývoj v automobilovém průmyslu, díky tomu generovaný kód splňuje normy ISO 26262, ISO 25119 a IEC 61508, MISRA C a zároveň obsahuje nativní podporu *AUTOSAR*u.

Hlavní myšlenka produktu spočívá v urychlení procesu vytváření softwarových aplikací. Doposud tento proces spočíval v práci funkčního vývojáře, který zpracovával informace od zákazníka. Následně ve vývojovém prostředí *Simulink*u modeloval a vytvářel algoritmy, do kterých zapracovával požadavky na funkcionalitu softwaru. Model–in–the–loop simulací pak tuto funkčnost ověřil. Následně bylo nutné k těmto algoritmům napsat specifikace a dokumentaci.



Obrázek 2.4: Knihovna standardních Simulink–bloků, které TargetLink nativně podporuje.

Tyto specifikace byly dále předány softwarovému inženýrovi, který se následně snažil navrženou funkcionalitu převést ze specifikací do zdrojového kódu. K tomu byla potřeba znalost programování a časová náročnost spojená s implementací návrhů ve specifikacích do produkčního kódu.

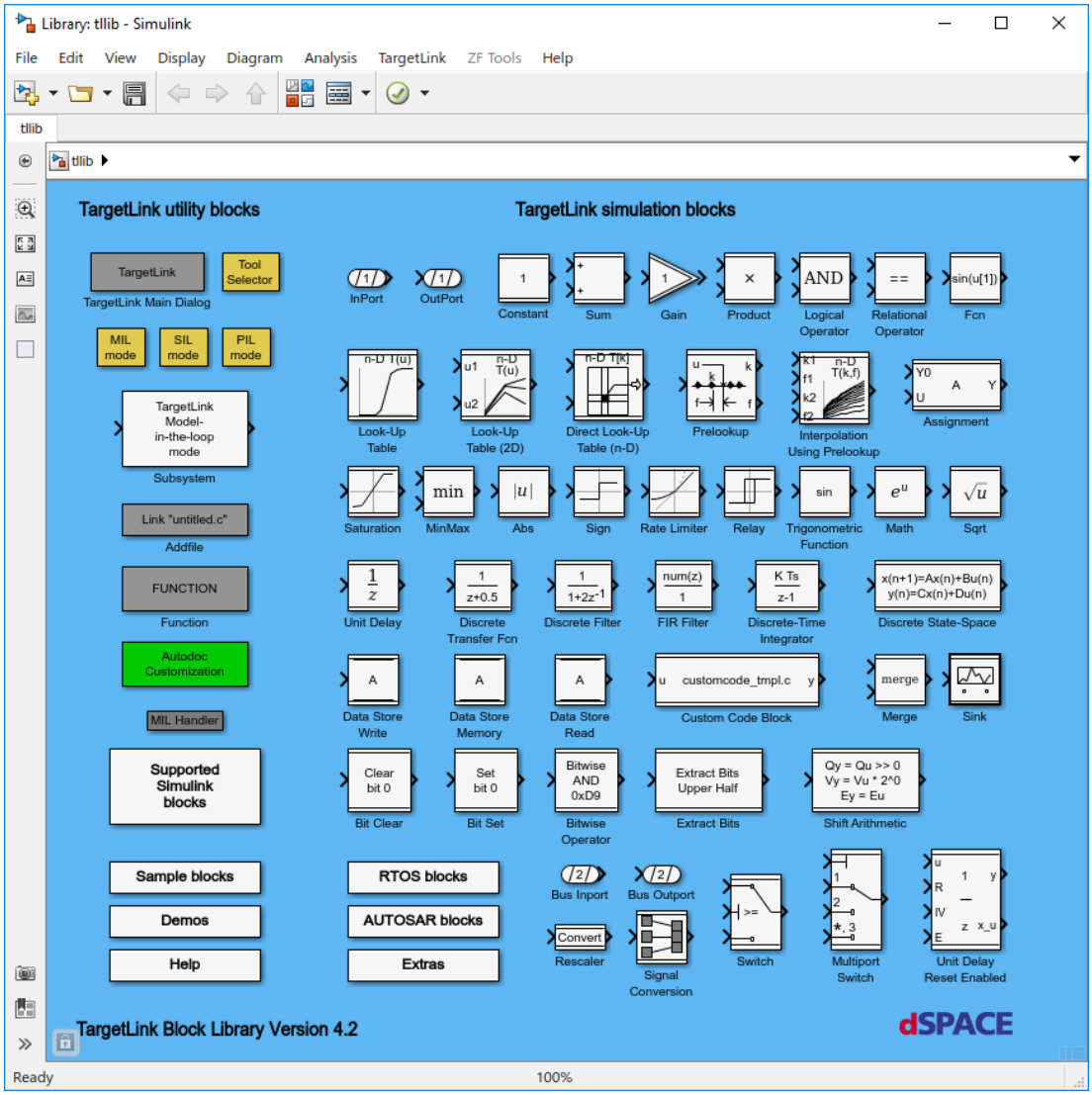
Kód mohl být následně otestován pomocí PIL[4] simulace. Pokud je během ní zpozorován funkční problém, je pečlivě zdokumentován. Dokumentace je následně poslána zpět funkčnímu vývojáři a celý cyklus se opakuje [9].

[4]PIL z anglického **P**rocessor–**i**n–the–**l**oop je typ simulace, kdy je matematický model simulován v reálném čase, řídicí systém je provozován na cílové hardwarové platformě. Při simulaci nejsou použity žádné I/O karty ani aktuátory. [13]

Možnost generování produkčního kódu a jeho testování přímo v rozhraní Simulink, by tedy podstatně zkrátila celý koloběh vývoje aplikace. Práci softwarového inženýra by na sebe mohl částečně převést funkční vývojář a program TargetLink.

TargetLink využívá mnoho bloků ze standardní knihovny Simulinku, jak je možné vidět na obrázku 2.4, zároveň plně podporuje Stateflow chart bloky a modelování pomocí stavových bloků.

S přímou podporou knihovny Simulinku nabízí TargetLink vlastní knihovnu (obrázek 2.5), kterou částečně nahrazuje většinu běžných simulinkovských bloků, pro které disponuje možnostmi dalšího nastavení s ohledem na generovaný kód. Pro generované proměnné je zde možné nastavit parametry jako:

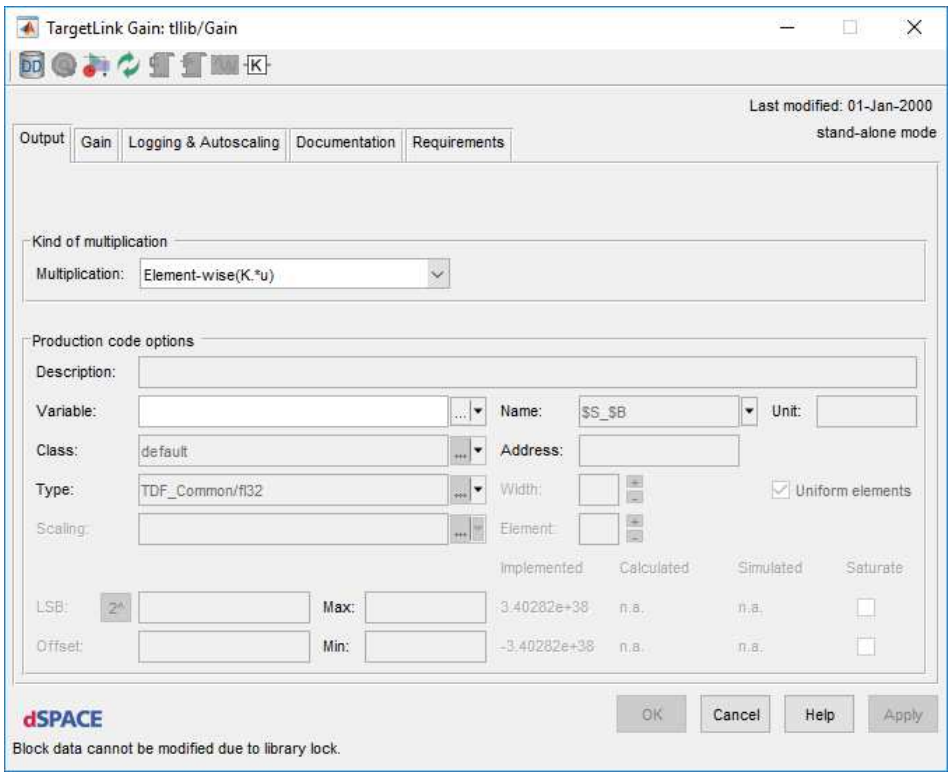


Obrázek 2.5: Knihovna simulačních bloků TargetLink.

- Jméno proměnné
- Jméno proměnné v kódu
- Datový typ
- Popis do komentáře, který se pak s kódem vygeneruje.

- Jednotky proměnné
- Škálování
- Offset

a mnoho dalších parametrů, které je možné specifikovat vlastním způsobem. Interface s možnostmi nastavení bloku „Gain“ je znázorněno na obrázku č. 2.6.



Obrázek 2.6: Možnosti nastavení pro generování C kódu, které nabízí targetlink–blok gain. Jednotlivé bloky, mají nabídku možností nastavení velmi podobnou

Data dictionary

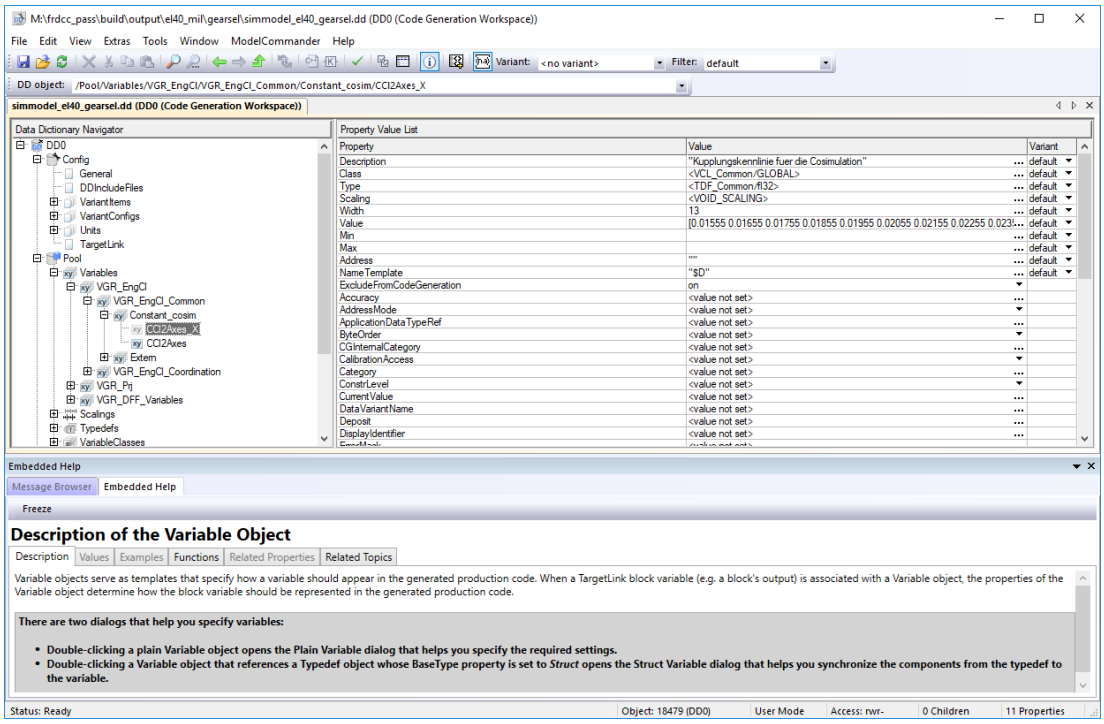
Targetlink obsahuje i nástroj *Data Dictionary*. Jedná se o databázi, kam je možné si definovat všechny používané proměnné se všemi podrobnostmi a nastaveními, jako je např.:

- jméno
- počáteční hodnota
- datový typ
- škálování
- jednotky proměnné
- popis proměnné
- adresa

- třída
- modul
- maximum/minimum

a ještě některá další. Takto nakonfigurovanou proměnou lze pak jednoduše použít v blocích nástroje *TargetLink* (viz. obrázek 2.6), kde po zadání jména proměnné se zbytek nastavení předvyplní.

Na následujícím obrázku 2.7 je možní si povšimnout, že nástroj nabízí v dolní části rozhraní i popis pro každou vybranou položku.



Obrázek 2.7: Ukázka možností nastavení parametrů proměnné v nástroji *Data Dictionary*, který je součástí programu *TargetLink*.

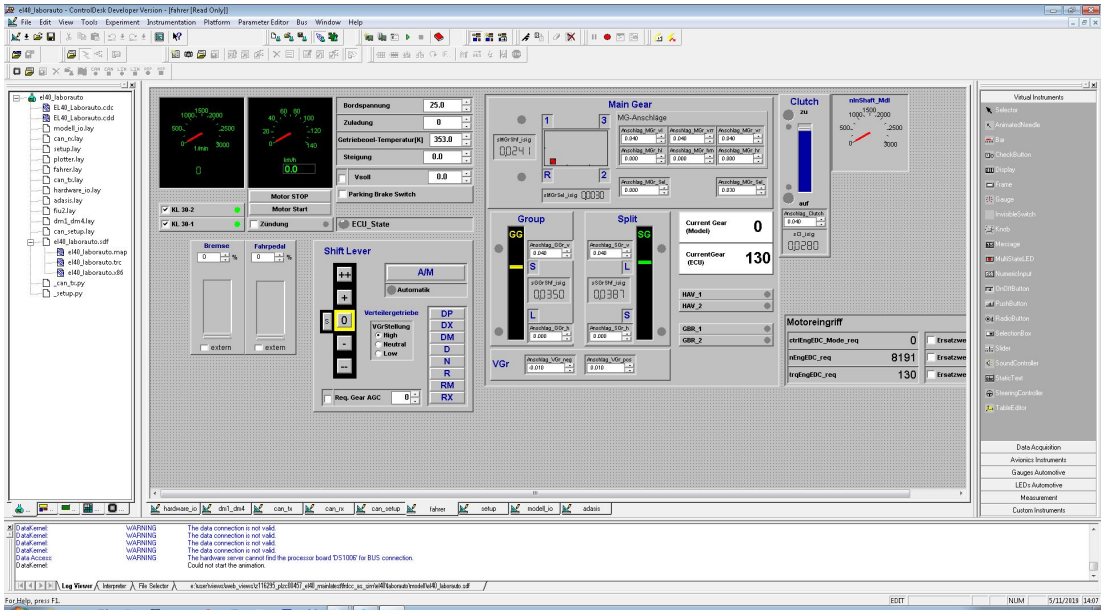
2.2.2.2 ControlDesk

Jedná se o univerzální modulární software, který slouží k vývoji elektronických řídicích jednotek (ECU). Současný ControlDesk verze 6.4 spojuje funkce, které často vyžadují více specializovaných nástrojů. Poskytuje přístup k simulačním platformám, je možné s jeho pomocí provádět měření, diagnostiku a kalibraci na ECU. Flexibilní modulární struktura programu poskytuje vysokou škálovatelnost pro splnění všech požadavků specifických aplikací [9].

Tento nástroj je cílen na úlohy jako jsou

- Rapid control prototyping
- Hardware–in–the–loop simulace (HIL)
- Měření a kalibrace ECU
- Přístup do sběrnicového systému vozidla (CAN, CAN FD, LIN, Ethernet)

V Plzni v oddělení vývoje převodovek do kamionů je používána pouze starší verze 3.7, která nenabízí tolik možností a je využívána především pro Hardware–in–the–loop (HIL) simulace, pro diagnostiku a měření jsou využívány interní nástroje nebo programy firmy Vector. Příklad použitého grafického rozhraní programu *ControlDesk* je možné vidět na obrázku číslo 2.8.



Obrázek 2.8: Ukázka grafického uživatelského rozhraní v této práci použitého programu ControlDesk.

2.2.3 Vector CANape

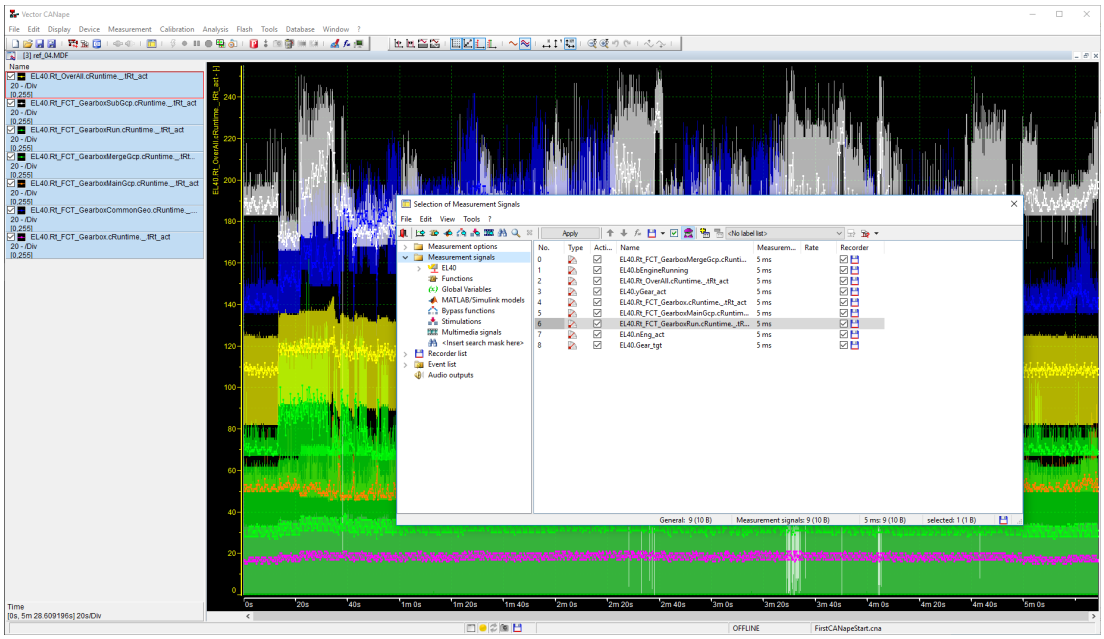
Vector je firma, která vyvíjí software i elektroniku pro automobilový průmysl a příbuzná odvětví. Svojí platformou nástrojů a softwarových komponent se snaží podporovat nejen výrobce automobilů, ale i jejich dodavatele. Svými produkty usnadňují inženýrům v automobilové průmyslu jejich náročné úkoly. Ukázka uživatelského rozhraní programu *CANape* je na obrázku číslo 2.9.

Jedním z takových nástrojů je i program CANape, který by měl najít primární využití především v oblasti optimalizace parametrizace elektronických řídicích jednotek. Během procesu měření můžou být současně kalibrovány parametry a být zaznamenávány signály. Komunikace mezi programem CANape a řídicí jednotkou probíhá přes protokoly jako XCP nebo přes rozhraní mikrokontroléru s měřícím a kalibračním hardwarem VX1000[5] Program podporuje také ADAS senzory[6], jako je například radar, LIDAR[7] a video. Díky možnosti správy kalibračních dat a snadnému zobrazení a vyhodnocení naměřených dat je program CANape komplexní nástroj pro kalibraci řídicí jednotky [11].

[5] Jedná se také o produkt firmy Vector.

[6] z anglického spojení **A**dvanced **D**river **A**ssistance **S**ystems - jedná se o asistenční systémy pro řidiče, které mají za úkol zvyšovat komfort a bezpečnost motorových vozidel.

[7] anglického spojení **Li**ght **D**etection **A**nd **R**anging je metoda měření vzdálenosti na principu laserového paprsku. K měřené překážce je vyslán laserový impuls, který se odrazí zpět ke snímači, z doby trvání odrazu je pak vypočtena vzdálenost.



Obrázek 2.9: Ukázka grafického uživatelského rozhraní nástroje CANape, s jehož pomocí bylo prováděno měření náročnosti aplikace v této práci.

2.2.4 EXAM

EXAM je nástroj pro automatizaci testů, který společně vyvinuly společnosti Volkswagen AG a MicroNova. Název EXAM vznikl spojením slov EXtended Automation Method. Pro návrh testů je v programu primárně používán vlastní grafický programovací jazyk, který je ale ve skutečnosti nadstavbou jazyka *Python*. Díky tomu může být programování velice variabilní, ve chvíli, kdy nějaká funkcionalita v grafickém jazyce chybí, může být snadno dopsána. Grafická metoda psaní testů poskytuje navíc výhodu, že pro její zvládnutí nejsou potřeba žádné důkladné znalosti programovacích technik. Centrální knihovna EXAMu funguje na principu sdílení celou komunitou uživatelů a v podstatě každý uživatel do ní může s jistými omezeními přispívat. Testy navržené v tomto nástroji slouží primárně pro automatické hardware in the loop simulace (HIL). Nově je však možné EXAM používat i pro softwarové software in the loop simulace (SIL) [12].

Kapitola 3

Hardwarové zdroje

Většina elektrických zařízení je v dnešní době ovládána softwarovými aplikacemi, které umožňují těmto zařízením reagovat na vstupy z okolí. K běhu těchto aplikací je zapotřebí obvykle mikrokontrolér s dostatečným množstvím paměti a periferií, aby byl schopen vykonávat zadané úkony dostatečně rychle. Účel použití daného zařízení je většinou hlavním kritériem při výběru hardwaru, který ho bude řídit. Pokud by aplikace běžící na dané jednotce byla příliš složitá a nebo by zpracovávala příliš mnoho informací použitý mikrokontrolér by informace nestíhal zpracovávat, běh aplikace by byl pomalý a zařízení by nefungovalo správně. Na druhou stranu, pokud by hardware byl předimenzovaný, zařízení by sice fungovalo správně, ale mohlo by mít větší spotřebu elektrického proudu, což sebou nese další obtíže, se kterými je nutné se vypořádat. Nejspíše hlavní důvodem pak ale je, že výkonnější hardware obvykle znamená dražší hardware a ve velkosériové produkci je i minimální zdražení součástky znát.

3.1 Použitý hardware

V této části práce jsou popsány systémové prostředky řídicí jednotky, které jsou dostupné pro běh aplikace. Informace o nich jsou dostupné ze specifikací k použité řídicí jednotce.

Obrázek 3.1: řídicí jednotka převodovky

Veškeré testy a měření bylo prováděno na řídicí jednotce k převodovce od firmy ZF. Jedná se o automatickou převodovku, známou pod názvem TraXon, používanou především ve velkých kamionech pro dálkovou přepravu. Jde o plně robotizovanou převodovku ovládanou pneumatickým systém. Tento typ převodovky je možné dodávat v několika variantách a to buď s dvanácti a nebo se šestnácti dopřednými rychlostmi

a čtyřmi zpátečními. Dále se vyskytují varianty, které je možné napájet napětím o velikosti 12 $[V]$ a 24 $[V]$. Mezi přednosti této převodovky patří i možnost modularity, kdy je možné zvolit připojení různého typu spojky, hydrodynamického měniče točivého momentu, elektromotoru či jiných zařízení.

Obrázek 3.2: Ukázka modulární převodovky, pro jejíž software byly prováděny optimalizace.

3.1.1 Paměť ROM

Paměť *ROM*[1] je pevná paměť řídicí jednotky, kde je uložena řídicí aplikace. Tento typ paměti je nezávislý na napájení a data v něm uložená se po odpojení zdroje elektrického napětí nevymažou.

Celá paměť není bohužel pro aplikaci zcela dostupná, část je vyhrazena pro firmware paměti, část pro bootloader[2] a další část pro hodnoty parametrů, které se buď mění během jízdy a je nutné si je pamatovat po dalším spuštění aplikace, nebo se jedná o hodnoty nastavení pro konkrétní značku a model vozidla.

celková *ROM*	3,75 MiB
dostupná *ROM*	2,88 MiB
využitá *ROM*	1,45 MiB

Tabulka 3.1: Přehled velikosti dostupné a využité *ROM* paměti.

[1] z anglického *R*ead–*O*nly *M*emory

[2] Zjednodušeně se jedná o program, který pomáhá spustit a zavést hlavní aplikaci po startu řídicí jednotky.

3.1.2 Paměť RAM

Paměť *RAM*[3] je polovodičový typ paměti s přímým přístupem, umožňuje jak čtení, tak i zápis. Tento typ paměti slouží výhradně pro běh aplikace a všechna data, pokud se nepřepíší do *ROM*, se po odpojení napájení smažou.

Stejně jako v předchozím případě není pro běh funkcionality aplikace uvolněna celá paměť. Část je opět vyhrazena pro bootloader, část pro konstanty a část paměti je prázdná a odděluje od sebe jednotlivé sektory.

celková *RAM*	256 KiB
dostupná *RAM*	153 KiB
využitá *RAM*	116 KiB

Tabulka 3.2: Přehled velikosti dostupné a využité *RAM* paměti.

3.1.3 Mikroprocesor

Pro řízení jednotky je použit procesor od firmy Renesas, který je taktován na $200MHz$. Hlavní aplikace je navržena, aby běžela v pěti milisekundových cyklech. Pokud smyčka proběhne rychleji, program do konce intervalu počká. Pokud se úloha nestihne v uvedeném čase spočítat, nechá se úloha doběhnout. Jednou za 5 cyklů se spouští WatchDog, který hlídá jestli cyklus neběží déle než 7 ms, pokud ano, jednotka se zresetuje.

Vzhledem ke zjištěným dostupným softwarovým prostředkům a skutečnosti, že je práce vypracovávána v soukromé firmě, pro kterou je v tuto chvíli kritická doba běhu hlavní aplikace, je práce především orientována na hledání metod optimalizující běh softwaru.

[3] z anglického *R*andom–*A*ccess–*M*emory

Kapitola 4

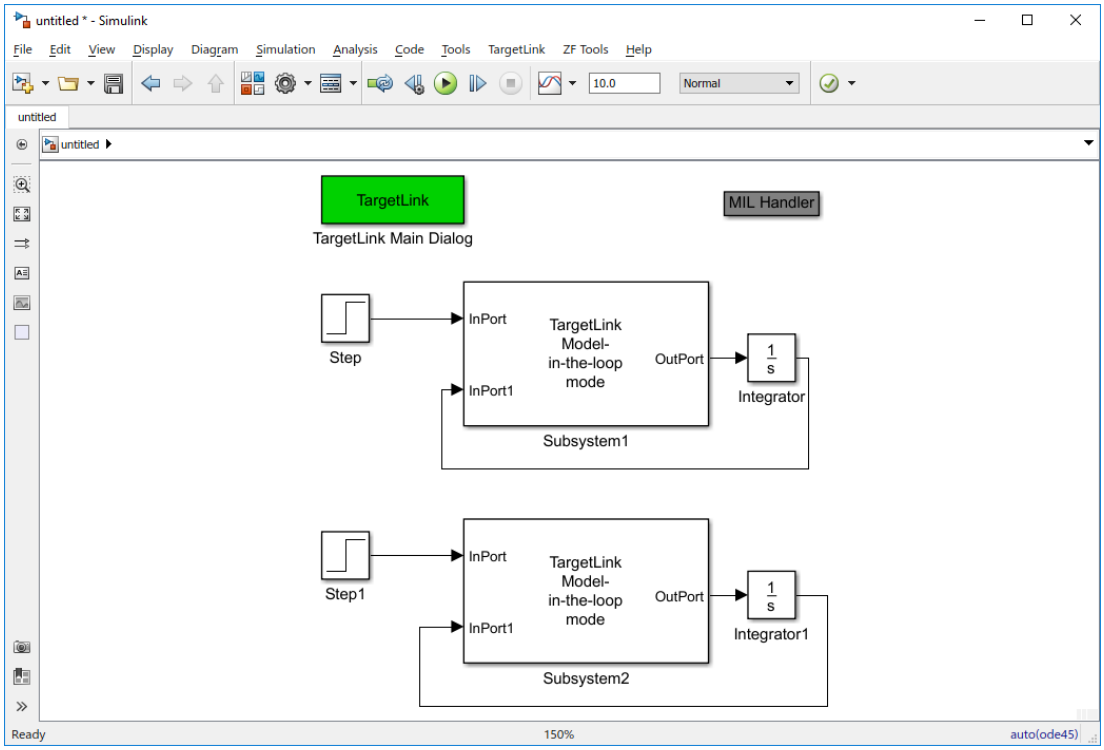
Automatické generování kódu

4.1 dSpace–Targetlink

Jako nástroj pro generování kódu byl v této práci použit program od firmy *dSpace* *TargetLink*, který nabízí alternativu k integrovanému generátoru zdrojového kódu *Embedded Coder*u od společnosti *MathWorks*.

4.2 Generování kódu

Pro generování zdrojového kódu je nutné do vrchní struktury modelu přidat z knihovny *TargetLink*u blok *TargetLink Main Dialog* a blok *MIL Handler*, celý model musí být navíc zavřen ve speciálním Targetlink–Subsystému, který také najdeme v knihovně programu. Ukázku modelu s těmito bloky je možné vidět na obrázku číslo 4.1. Právě

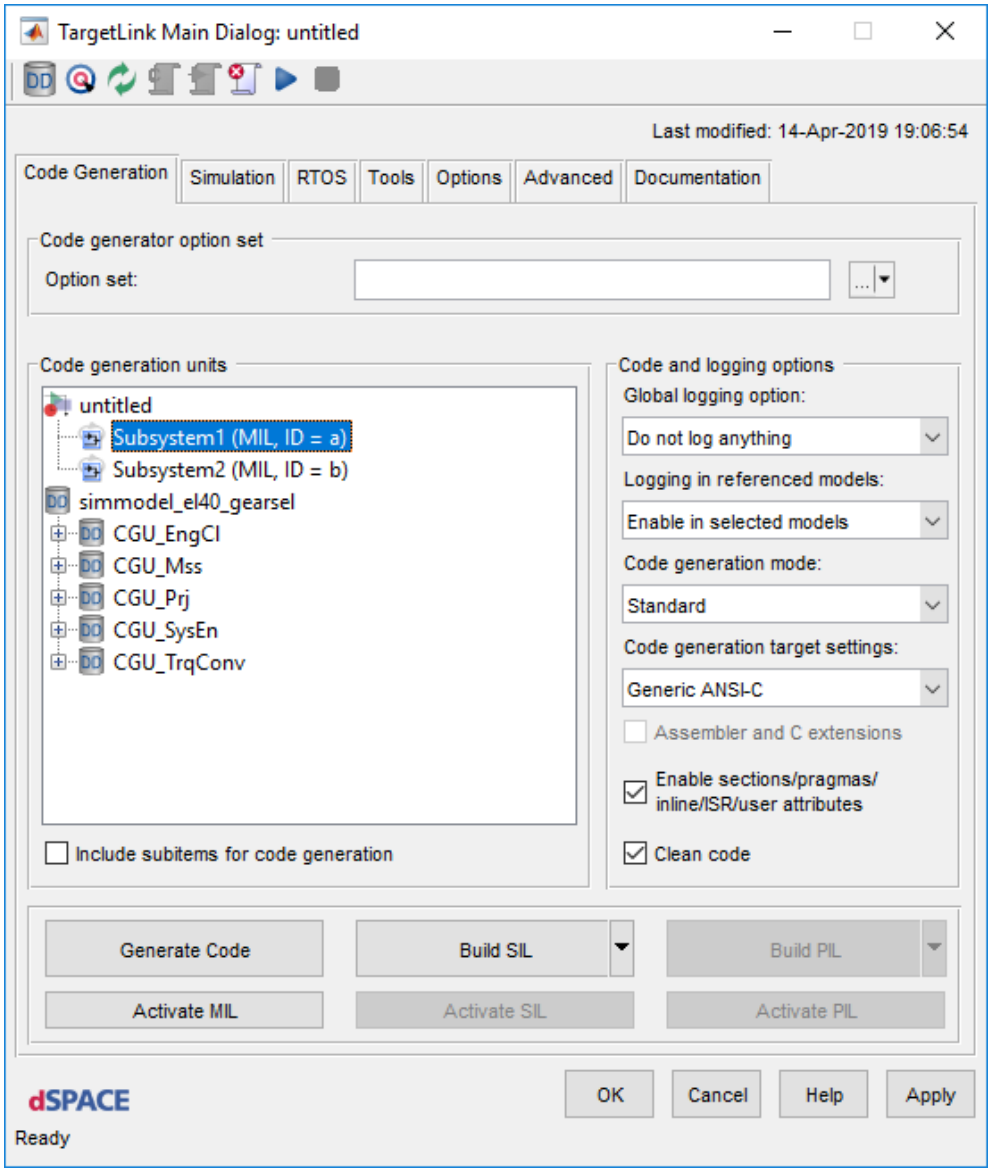


Obrázek 4.1: Ukázka vrchní vrstvy modelu, možné rozložení subsystémů a Targetlink–bloků nutných ke generování kódu.

z těchto subsystémů se bude pracovní kód generovat. Tento systém dává uživateli na výběr zda chce vygenerovat kód pro celý model, nebo třeba jen pro jeden subsystém. Pro vytvoření modelu, ze kterého může být pomocí programu *TargetLink* generován zdrojový kód, je zapotřebí použít výhradně modelové bloky z knihovny *TargetLinku* 2.5 nebo z knihovny bloků *Simulinku*, které mají přímou podporu programu 2.4.

4.2.1 Výchozí rozhraní

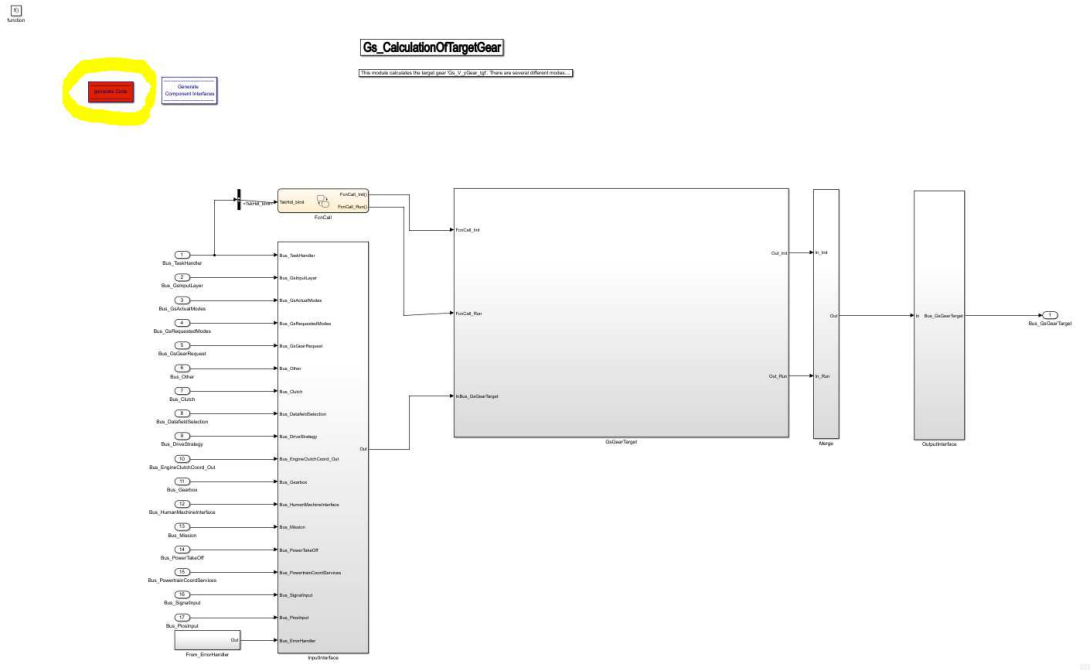
Dialog pro generování kódu se spouští poklepáním na blok *TargetLink Main Dialog*, následně se otevře hlavní rozhraní, které nabízí poměrně rozsáhlé možnosti nastavení. Neslouží pouze ke spouštění generování c–kódu, ale například i pro generování dokumentace. Dále se zde nalézá mnoho dalších nastavení pro samotné generování. Všechna tato nastavení je možné uložit do konfiguračního souboru a ten pak jednoduše otevřít v kolonce *Option set:*. Obrázek číslo 4.2 nabízí ukázku tohoto rozhraní.



Obrázek 4.2: Ukázka uživatelského rozhraní pro generování kódu.

4.2.2 Generování v ZF

Generování kódu ve společnosti ZF je podobné, ale postup je o něco zjednodušený. Stále se používá konstrukce zobrazená na obrázku 4.1, pro generování jsou ale použity firemní konfigurace, bylo by tedy zbytečné proklikávat se všemi možnostmi, které standardní uživatelské rozhraní nabízí. Z toho důvodu bylo vytvořeno tlačítko pro generování kódu (obrázek 4.3), které je umístěno na úrovni modelu, odkud má být kód generován. Uživatel je pak pouze vyzván, jestli chce aktualizovat parametry proměnných uložených v *Data Dictionary* a může přistoupit přímo ke generování kódu, všechna ostatní nastavení jsou nakonfigurována tak, aby se vykonávala automaticky na pozadí.



Obrázek 4.3: Ilustrační příklad použití v praxi. Ve žlutém kroužku je znázorněno tlačítko pro generování.

Kapitola 5

Měření

5.1 Metody měření

V následující sekci budou popsány metody, které byly pro měření náročnosti aplikace použity. Nejdříve bude popsáno měření paměti, v následující části bude rozebrán mechanizmus měření doby běhu aplikace.

5.1.1 Měření velikosti paměti

Velikost obsazení paměti je možné vyčíst z takzvaného *.map*–souboru, což je textový soubor, kde jsou obsaženy informace, na jaké místo v paměti se při flashování softwaru do jednotky která část dat uloží. Pomocí interní *Java*–aplikace je možné z adres *.map*–souboru vygenerovat *.xls*–soubor, kde je přehledně vypsáno obsazení jednotlivých částí *RAM* a *ROM* paměti.

5.1.2 Měření CPU runtimu

Pojem „runtime“, který je dále použit, označuje dobu běhu programu, k měření té doby nebo doby běhu jen části programu v rámci jedné smyčky je v aplikaci implementována interní funkce, která má dva argumenty. Prvním je ID instance a druhým je rozlišení měření v mikrosekundách. Dále obsahuje funkce *begin()* a *end()*.

Při zavolání funkce *begin()* se načte čas z procesorového čítače do proměnné instance třídy. Tento čas je brán jako výchozí.

Funkce *end()* po svém zavolání načte znovu čas z procesorového čítače a odečte ho od původní hodnoty. Tato funkce zároveň počítá a ukládá do atributu třídy maximální naměřenou hodnotu a průměrnou hodnotu doby běhu měřeného funkcionality.

Listing 5.1: Ukázka fungování funkcí *begin()* a *end()*.

```
void begin()
{
        start_value = Get.timer_value();
}

void end()
{
        runtime_value = Get.timer_value() - start_value;
        if(runtime_value > runtime_value_max)
        {
                runtime_value_max = runtime_value;
        }

        ...
```

```
}
```

Samotné měření runtimu bylo prováděno se softwarem, který byl nahrán na skutečné řídicí jednotce převodovky. Testy byly prováděny během simulace, kdy byla jednotka připojena k HiL simulátoru, který umožňuje provádět hardware in the loop simulace. Hodnoty runtimu byly vyčítány za běhu programu do externího počítače přes rozhraní CAN pomocí nástroje Vector CANape.

5.2 Příprava dat pro měření

Jak bylo zmíněno v předchozí kapitole, měření doby běhu funkcí v jednom cyklu aplikace nelze provést tak jednoduše a přímo, jako měření paměti.

Vybrané funkce či úseky kódu, které mají být změřeny, musejí být obaleny do obálky funkcí pro měření. Vzhledem ke skutečnosti, že automaticky generována je pouze část softwaru, bylo nutné nejprve generované úseky vyhledat a analyzovat. Následně byly rozděleny do funkčních celků, které byly měřeny zvlášť. Pro každý tento celek byla vytvořena instance třídy *cRuntimeEx* a bylo nutné navrhnout a nastavit rozlišení v mikrosekundách, ve kterém se daný úsek měřil. Možné před definované varianty jsou $1, 2, 4, 8, \ldots, 4194304$ [μs], jedná se o mocniny čísla dvě.

Při návrhu je třeba brát ohled na složitost dané části kódu a na očekávané době trvání, protože datový typ parametru, do kterého je hodnota doby trvání zapisována je osmibitový unsigned integer. To znamená, že maximální hodnota parametru je omezena číslem 255[1], pak by se mohlo stát, že datový typ "přeteče"a naměřené hodnoty nebudou správné, na druhou stranu, pokud se zvolí rozlišení příliš nízké, výsledné hodnoty nebudou dostatečně přesné. Před prvním měřením byly zvoleny hodnoty rozlišení 4[μs], které pak byly podle naměřených výsledků individuálně upraveny. Ukázku deklarování těchto instancí je možné vidět v následujícím výpisu 5.2

Listing 5.2: Ukázka definování instancí třídy *cRuntimeEx* a jejího rozlišení. První argument třídy je index dané instance a druhý argument je pak zvolené rozlišení.

```
cRuntimeEx Rt_Gearbox_A           ( 0x40, C_Resolution_4us);
cRuntimeEx Rt_Gearbox_B           ( 0x41, C_Resolution_4us);
cRuntimeEx Rt_Gearbox_C           ( 0x42, C_Resolution_4us);
cRuntimeEx Rt_Gearbox_D           ( 0x43, C_Resolution_4us);
cRuntimeEx Rt_Gearbox_E           ( 0x43, C_Resolution_4us);
cRuntimeEx Rt_Powertrain_A        ( 0x44, C_Resolution_4us);
cRuntimeEx Rt_Powertrain_B        ( 0x45, C_Resolution_4us);
cRuntimeEx Rt_Powertrain_C        ( 0x46, C_Resolution_4us);
cRuntimeEx Rt_Powertrain_D        ( 0x46, C_Resolution_4us);
...
```

Pro implementace jsou této měřící funkcionality jsou v softwaru vytvořeny funkce *M_RT_BEGIN()* a *M_RT_END()*, které ohraničují měřený úsek. Jako argument do těchto funkcí vstupuje název instance třídy *cRuntimeEx* (např. *Rt_Fct_Applic*, viz. výpis 5.3) a uvnitř je pouze volána funkce *begin* nebo *end* příslušné instance.

Listing 5.3: Ukázka použití funkcí pro měření runtimu implementovaných v kódu kde funkce "M_RT_BEGIN"a "M_RT_END"jsou funkce které ohraničují měřený úsek a parametr "Rt_FCT_Applic"je parametr do kterého se ukládá naměřený čas běhu všech funkcí které se spouští mezi M_RT_BEGIN a M_RT_END je zaznamenávána do použitého parametru.

```
{
```

[1] Nejvyšší hodnota je 255 z důvodu, že se jedná o osmi bitový integer a to znamená 2^8 hodnot

```
    M_RT_BEGIN(Rt_Powertrain_A);

    PwrTrain_A.TaskSchedule();

    M_RT_END(Rt_Powertrain_A);
}
```

Takovým způsobem byly postupně bylo obaleno celkem 15 modulů, které jsou automaticky generovány. Jejich přehled je v následující tabulce 5.2.

Komponenta Gearbox A	Komponenta Gearbox se stará o řízení mechaniky převodového ústrojí uvnitř převodovky.
Komponenta Gearbox B	
Komponenta Gearbox C	
Komponenta Gearbox D	
Komponenta Gearbox E	
Komponenta Powertrain A	Tato komponenta se stará převod točivého momentu ve směru do i ze převodovky při specifických jízdních režimech (rozjíždění se při uvolnění brzdy, jízda v malé rychlosti nebo při zapnutém adaptivním tempomatu).
Komponenta Powertrain B	
Komponenta Powertrain C	
Komponenta Powertrain D	
Komponenta DriverDemant	Tato komponta zpracovává příkazy přicházející od řidiče.
Komponenta Mission	Komponenta Mission se stará o obsluhu různých režimů převodovky, ve kterých je možné využít výkon motoru na jiných zařízeních vozidla, například u jeřábu, domýchávačky nebo popelářského auta.
Komponenta SystemEnergy	Komponenta SystemEnergy je zodpovědná za řízení tlaku v pneumatickém systému, tento system slouží k fyzickému ovládání mechanických částí převodovky.
Komponenta TorqueConverter	Tato komponenta se stará o obsluhu hydrodynamického měniče točivého momentu. Ten může být ve vozidle použit dvojím způsobem. V prvním případě může být připojen mezi motor a převodovku, kde slouží místo spojky, nebo může být mezi převodovkou a výstupem ke kolům. Tam funguje jako hydrodynamická brzda.
Komponenta PreSelection	Tato komponenta se stará o připravení řazení rychlosti, která bude zařazena, ale čeká na splnění všech potřebných podmínek, aby výsledné řazení mohlo proběhnout rychleji.

Tabulka 5.1: Přehled všech autkódových komponent. Jsou zde zobrazeny symbolicky jednotlivé komponenty a stručný popis chování jednotlivých komponent. Skutečná jména částí softwaru nejsou uvedena z důvodu, že to není pro práci nezbytně nutné a bylo by to v rozporu s chráněným know–how společnosti.

5.3 Zajištění validace měření

Aby bylo možné software testovat a výsledná data z jednotlivých verzí mezi sebou porovnávat, bylo nutné vytvořit metodu testování tak, aby bylo možné daný test libovolně opakovat. Z dostupných zdrojů, které byly ve společnosti k dispozici, se nabízely dvě varianty.

První možností bylo navrhnout makro v nástroji ControlDesk. Jednalo se o posloupnost příkazů, které program zaznamenával tak, jak byly zadávány uživatelem. Tato sekvence byla následně uložena jako skript v programovacím jazyce Python a mohla být znovu spuštěna.

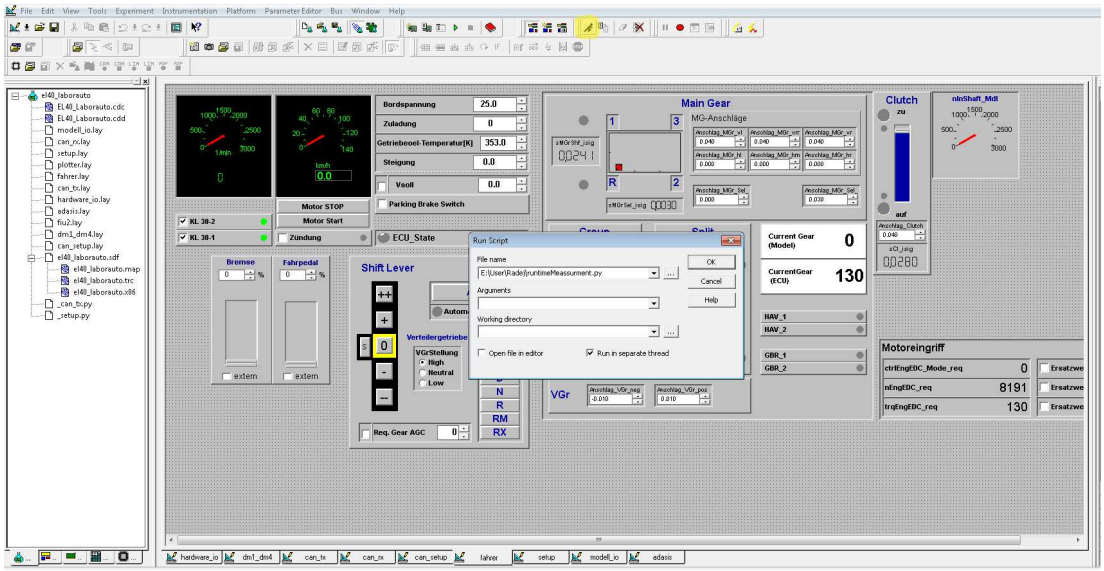
Druhá možnost byla vytvoření plnohodnotného testu v programu Exam a ten potom spouštět.

5.3.1 Makro v nástroji ControlDesk

ControlDesk je nástroj firmy dSPACE, který je popsán výše v sekci 2.2.2.2. Slouží jako vstupní rozhraní pro ovládání hardware in the loop simulace. Je zde možnost nastavení podobných rozhraní, které se nalézají ve skutečném vozidle a jedná se o relevantní vstupy, které mohou ovlivnit chod automatické převodovky, na příklad zapalování, nastavení brzdy a plynu, volič automatické převodovky, dále jsou zde možné nastavit vnější vlivy jako například stoupání či klesání vozovky.

Program nabízí i možnost, posloupnost těchto navolených příkazů nahrát, uložit a pak znovu spustit. Tuto sekvenci ukládá jako jednoduchý skript v Pythonu.

Po opětovném spuštění nahraného skriptu, začal program vykonávat jednotlivé příkazy tak, jak byly nahrány. Implementace tohoto mechanismu je velmi jednoduchá a po spuštění už neobsahuje žádné další uživatelské rozhraní, jen se vykonávají jednotlivé úkony. Není zde ani žádná notifikace, zda sekvence běží, nebo zda již proběhla, nelze ani její vykonávání přerušit, uživatel musí počkat, dokud celá neproběhne. Pokud před spuštěním makra nejsou počáteční podmínky nastaveny do výchozího stavu, program to není schopen detekovat. Začne vykonávat spuštěnou sekvenci příkazů nezávisle na tom, v jakém je právě stavu[2].



Obrázek 5.1: Ukázka startu testovací sekvence v nástroji dSpace ControlDesk. Žlutě je vyznačeno tlačítko pro spuštění startovacího dialogu.

Měření hodnot spotřebovaného času probíhá pomocí externího programu CANape. Hodnoty se z jednoty vyčítají za běhu přes rozhraní CAN a program je v reálném čase

[2]např.: Pokud se v testovací sekvenci počítá s tím, že vozidlo se rozjíždí z klidu, ale automobil ve chvíli spuštění již jede, sekvence příkazů sice proběhne, ale chování vozidla bude jiné, než se předpokládalo. Měření tím pádem není použitelné.

zobrazuje. Nástroj CANape není bohužel s ControlDeskem nijak provázaný, proto se měření musí spouštět manuálně a jeho ukončení také.

Z výše zmíněného je patrné, že tato metoda má poměrně mnoho nevýhod jako jsou

- každé spuštění makra je nutné provést manuálně,
- měření hodnot není provázáno se sekvencí, jeho spuštění a ukončení musí být provedeno odděleně,
- testovací sekvenci nelze ukončit a musí proběhnout vždy celá,
- uživatel musí dávat pozor na správné nastavení počátečních podmínek před spuštěním testovací sekvence.

Na druhou stranu tento přístup poskytuje výhody ve formě

- snadného ovládání a velké univerzálnosti,
- je velmi snadné navržený test změnit,
- nabízí snadnou implementaci nových změn v softwaru.

5.3.2 Test v programu EXAM

Jedná se o program určený k testování softwaru při hardware in the loop simulacích. Program slouží hlavně ke psaní testů, je ale také propojen s ControlDeskem, do kterého posílá příkazy, které se mají vykonat. V rámci testů je možné volat program CANape a spouštět v něm měření, takže měření se zapíná a vypíná automatiky s testovací sekvencí. Zároveň se po skončení testu výsledky uloží.

Testy jsou v EXAMu rozděleny do tří částí

1. část inicializační
   V této části se nastaví počáteční podmínky testu, vozidlo se z výchozího stavu uvede do situace, která bezprostředně předchází samotnému testu.
2. část testovací
   V této části se provede samotný test a změří se všechny žádané signály.
3. část ukončovací
   Tato část testu slouží k uvedení vozidla zpět do výchozího stavu tak, aby test, který následuje, měl zajištěny stejné počáteční podmínky.

Díky tomuto způsobu testování je možné spouštět několik testů automaticky za sebou, uživatel pouze nastaví, jaké testy se mají spustit a kolikrát, a program se již postará o spouštění testů, měření i ukládání výsledků.

Způsob měření náročnosti běhu aplikace pomocí nástroje EXAM nabízí mnoho výhod oproti předchozí metodě. Technicky se sice od předchozího způsobu měření příliš neliší, provádí se v podstatě stejné operace, ale EXAM je dokáže provádět automaticky bez nutnosti zásahu uživatele.

Naproti tomu vyskytují se zde nevýhody v podobě složitější obsluhy programu.

Výhody

- možnost automatického spouštění testů,
- možnost spouštění stejného testu (stejné sekvence) vícekrát po sobě,
- spouštění a vypínání měření je prováděno automaticky během testu.

Nevýhody

- náročná a složitá příprava samotného testu, k napsání kvalitního testu je potřeba širšího know–how,
- těžkopádné přidávání nových signálů, které mají být změřeny,
- oproti předchozí metodě je zde složitější implementace nových změn v softwaru, který má být změřen,

5.3.3 Použitá metoda pro zajištění validace měření

Po analýze kladů a záporů jednotlivých metod, bylo jako výsledné řešení vybráno měření pomocí makra přímo v programu ControlDesk. Volbě tohoto způsobu nahrála snazší implementace nových změn v kódu a nenáročné sestavení testů, dalším důvodem je, že díky odbourání jedné softwarové mezivrstvy, odpadá systémová komunikace navíc a při měření je možné dosáhnout lepší realtime synchronizace s HIL simulátorem. Pro úplnou synchronizaci by však program ControlDesk musel být vybaven pluginem, který však nebyl k dispozici.

Vzhledem k tomu, že při použití tohoto způsobu měření se spouští měření signálů z jednotky nezávisle na testovací sekvenci, bylo nutné jasně definovat začátek a konec testu, který by byl v naměřených datech jasně viditelný. Z toho důvodu byla mezi měřené signály přidána hodnota indikující běh motoru. Podle spuštění a vypnutí motoru simulovaného vozidla se určoval začátek a konec testu.

Program CANape umožňuje nastavit frekvenci vyčítání dat z jednotky. Jak bylo řečeno v sekci výše (3.1.3), hlavní aplikace běží ve smyčce, která trvá 5 ms, proto bude stejná perioda nastavena pro vyčítání dat z jednotky tak, aby byla hodnota vyčtena v každém cyklu.

Pro vlastní měření byla navržena jednoduchá sekvence, kdy je testována dopředná jízda vozidla nejprve v automatickém módu, následně v manuálním módu a na závěr jízda vzad. Celá testovací sekvence je kompletně popsána níže.

1. Motor je nastartován.
2. Na voliči převodovky je vybrán automatický mód.
3. Na voliči převodovky je vybrána poloha *D*, jako "drive".
4. Pedál plynu je stlačen na hodnotu 100%.
5. Vyčká se, dokud není zařazena nejvyšší rychlostní stupeň (12), to je přibližně kolem rychlosti $105\frac{km}{h}$.
6. Po dosažení nejvyššího rychlostního stupně je pedál plynu uvolněn, nastaven na hodnotu 0%, a pedál brzdy je stlačen na hodnotu 100%.
7. Je vyčkáno dokud vozidlo úplně nezastaví.

8. Následně je na voliči automatické převodovky vybrán manuální mód.
9. Pedál brzdy je úplně uvolněn a pedál plynu je opět stlačen na 100%.
10. Postupně je manuálně řazeno až do 6. rychlostního stupně.
11. Následně je znovu zabrzděno.
12. Na voliči převodovky je zařazena zpátečka *R*.
13. Poté je nastaven plný plyn.
14. Postupně se zařadí všechny 4 zpětné rychlosti.
15. Po zařazení všech rychlostí se vozidlo opět zastaví.
16. Sekvence končí vypnutím motoru.

5.4 Získání referenčních dat

Aby mohly být zjišťovány změny v náročnosti aplikace ve vztahu k hardwaru, je nalezení validní metody měření prvním krokem. Dalším krokem, který musí logicky následovat, je, pomocí této metody (nalezené v bodě 5.3.3) získat referenční data. Tato měření budou moci být následně porovnána s daty získanými ze změněného softwaru.

Při prvním měření byly naměřeny časové hodnoty všech signálů z tabulky 5.2 a navíc signál *bEngineRunning* indikující běh motoru, který byl použit pro zjištění začátku a konce měření.

Následně bylo provedeno první měření referenčních dat. Hodnoty signálů se samozřejmě lišily v čase, pro lepší názornost byla z každého signálu vypočítána průměrná hodnota v průběhu celé sekvence.

Signál	průměrná hodnota [μs]
Komponenta Gearbox A	182.64
Komponenta Gearbox B	50.77
Komponenta Gearbox C	236.93
Komponenta Gearbox D	86.30
Komponenta Gearbox E	0.10
Komponenta Powertrain A	0.19
Komponenta Powertrain B	6.60
Komponenta Powertrain C	0.21
Komponenta Powertrain D	8.88
Komponenta DriverDemant	0.09
Komponenta Mission	118.15
Komponenta SystemEnergy	45.67
Komponenta TorqueConverter	0.08
Komponenta PreSelection	0.26

Tabulka 5.2: Průměrné hodnoty v mikrosekundách všech měřených modulů z jednoho měření.

Vzhledem ke skutečnosti, že nejjemnější možné rozlišení, se kterým je možné měřit, je 1 mikrosekunda (viz. 5.2), je z uvedené tabulky na první pohled jasné, že mnoho modulů softwaru je tak malých, že jejich dopad na běh CPU, je mimo rozsah měřitelnosti. Proto je při následném vybírání kandidátů pro optimalizaci není třeba uvažovat.

5.4.1 Analýza naměřených dat

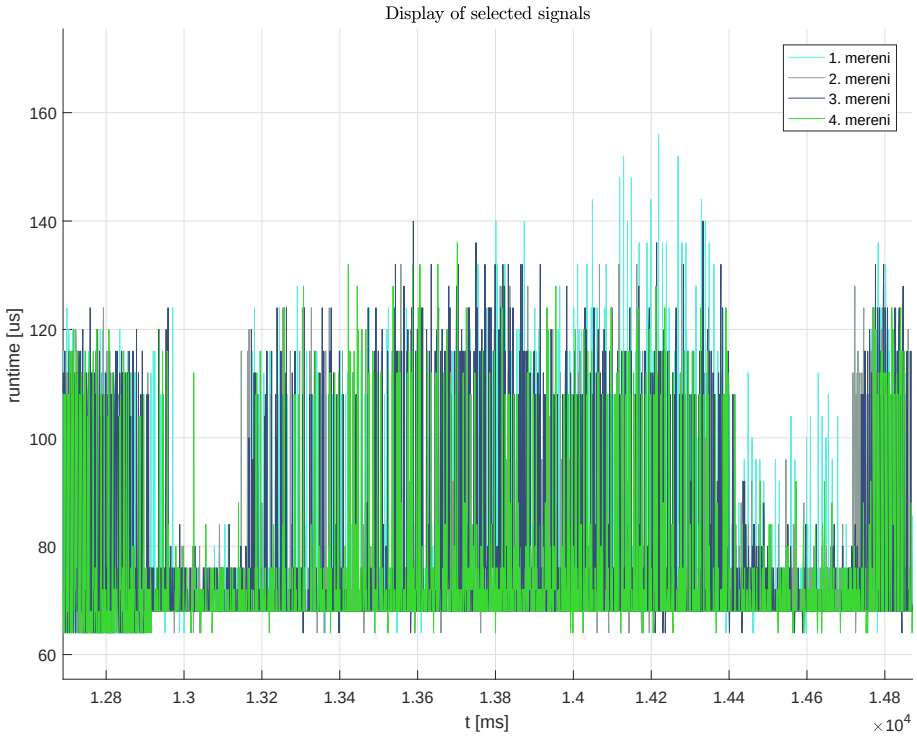
Následně byla naměřena další 4 měření a provedena hlubší analýza. Jednotlivá měření byla mezi sebou porovnána a bylo zjištěno, že se signály z různých měření od sebe liší nejen průběhem, ale i střední hodnotou.

Tato různorodost je dána především kombinací těchto dvou faktorů

- složitostí aplikace,
- systémem simulace HIL simulátoru, která není real–time

Simulace na HIL simulátoru pomocí programu ControlDesk neprobíhá v reálném čase, existuje plug–in do ControlDesku, který real–time simulaci podporuje a je schopný běh svého prostředí s během simulátoru a testované jednotky synchronizovat, ale ve firmě nebyl bohužel k dispozici. HIL simulátor tak vkládá do testu jistý prvek neurčitosti, může dojít ke zpoždění zadání příkazu v řádu milisekund. V kombinaci s komplexností aplikace převodovky, je každá simulace trochu jiná. Počáteční podmínky nejsou vždy u HIL simulátoru úplně identické, v softwaru naopak existuje mnoho cest, kterými se může běh aplikace každý cyklus ubírat. Z tohoto důvodu se jednotlivá měření lišila i v délce trvání v řádu sekund. Tuto skutečnost je možné vysvětlit tak, že v některých situacích program mohl zvolit trochu jiné cesty, tím pádem se některé výpočty mohli o několik cyklů zpozdit. Výsledkem mohlo být trochu jiné řazení a z toho vyplývající odlišná akcelerace vozidla.

Při porovnávání signálů z jednotlivých měření je možné si všimnout podobnosti časového průběhu, ale zároveň je na první pohled zřejmé, že hodnoty nejsou identické. Příklad takového porovnání je znázorněn na následujícím obrázku 5.2, kde je znázorněn časový průběh CPU náročnosti autokodové komponenty Gearbox_D ze čtyř měření.



Obrázek 5.2: Ukázka krátké části měření náročnosti jedné z autokódových částí softwaru (*Gearbox_D*), pro porovnání je zde zobrazen průběh ze čtyř měření.

Z uvedené skutečnosti plyne nemožnost přesného porovnání dvou měření, proto pro lepší představu o hodnotách náročnosti programu byl ze všech měření vypočítán aritmetický průměr a zároveň ke každému signálu vypočítána odchylka.

Odchylka je vypočítána nejprve pro každý signál zvlášť, jako druhá mocnina rozdílu průměrné hodnoty a střední hodnoty konkrétního signálu.

$$o_i = \underbrace{\frac{1}{N}\sum_{j=1}^{N}\frac{1}{T}\sum_{t=1}^{T}s_j^2(t)}_{\substack{\text{průměrná hodnota}\\ \text{signálu ze všech měření}}} - \underbrace{\frac{1}{T}\sum_{t=1}^{T}s_i^2(t)}_{\substack{\text{průměrná hodnota}\\ \text{signálu z konkrétního}\\ \text{měření i}}}, \tag{5.1}$$

kde

$N \ldots$ je počet měření,
$T \ldots$ je celkový počet časových okamžiků měření,
$i \ldots$ je číslo konkrétního měření,
$j \ldots$ jsou indexy jednotlivých měření
$s \ldots$ je měřený signál,
$o \ldots$ je odchylka signálu z konkrétního měření. i

Výsledná odchylka, která je zapsaná v tabulce je pak aritmetický průměr odchylek signálu přes všechna měření.

$$o = \frac{1}{N}\sum_{i=1}^{N}o_i, \tag{5.2}$$

kde je opět

$N \ldots$ počet měření,
$i \ldots$ číslo konkrétního měření,
$o \ldots$ průměrná odchylka daného signálu.

Vypočítané hodnoty jsou zaznamenány v následující tabulce.

Signál	prům. hod. [μs]	odchyl. sig. [μs]	rel. odchyl. sig. [%]
Komponenta Gearbox A	184.64	0.15	0.08
Komponenta Gearbox B	49.77	0.14	0.28
Komponenta Gearbox C	240.3	0.083	0.15
Komponenta Gearbox D	87.28	0.13	0.15
Komponenta Gearbox E	0.10	0.001	1.05
Komponenta Powertrain A	0.19	0.00093	0.49
Komponenta Powertrain B	6.43	0.0024	0.04
Komponenta Powertrain C	0.21	0.059	2.81
Komponenta Powertrain D	8.70	0.012	0.14
Komponenta DriverDemant	0.10	0.00025	0.26
Komponenta Mission	120.15	0.041	0.03
Komponenta SystemEnergy	45.55	0.065	0.14
Komponenta TorqueConverter	0.08	0.00058	0.69
Komponenta PreSelection	0.25	0.00045	0.18

Tabulka 5.3: Průměrné hodnoty časové náročnosti autokódových částí softwaru vypočítané přes všechna měření a odchylka těchto hodnot.

Výše v této sekci bylo vysvětleno, že v daných podmínkách nelze dosáhnout zcela deterministických výsledků měření spotřeby času. Pro zpřesnění výsledných hodnot bylo provedeno více měření, ze kterých byla vypočítána střední hodnota. Tyto hodnoty a jejich průměrné odchylky je možné vidět v předchozí tabulce 5.3. Je možné si povšimnout, že relativní odchylka je poměrně malá a to především u vyšších naměřených hodnot.

Ze zmíněného vyplývá, že stačí pro dostatečně přesnou hodnotu udělat více měření, v tomto případě byl počet stanoven na dvacet pět měření pro referenční signál, byl to výsledek kompromisu mezi přesností odhadu a časovou náročností měření. Hodnota N použitého ve vztazích 5.1 a 5.2 je tedy

$$N = 25. \tag{5.3}$$

5.5 Metody zpracování měření

Pro dosažení ještě lepších výsledků bylo rozhodnuto, že místo jedné průměrné hodnoty, přes celý testovací interval, by mohli být vybrány čtyři kratší úseky, kde by rozdíly mezi simulacemi mohlo být ještě menší a tím pádem by dopad navržených změn mohl být viditelnější.

Pro měření byla použita stále stejná testovací sekvence popsána v části 5.3.3. Pro snadnější analýzu byly mezi měřené signály přidány i signál ukazující aktuální zařazenou rychlost ($yGear\_act$) a signál požadovanou rychlost, která by se měla zařadit ($yGear\_tgt$).

Testovací intervaly pro porovnání byly vybrány následovně.

1. úsek, kdy vozidlo jede na plný plyn v dopředném směru a v automatickém módu řazení, mezi rychlostními stupni 8 a 11. Úsek začíná, když požadovaná rychlost se přepne ze nižšího rychlostního stupně na osmý, a končí ve chvíli, kdy přestává být požadována rychlost deset a začíná rychlost jedenáct.

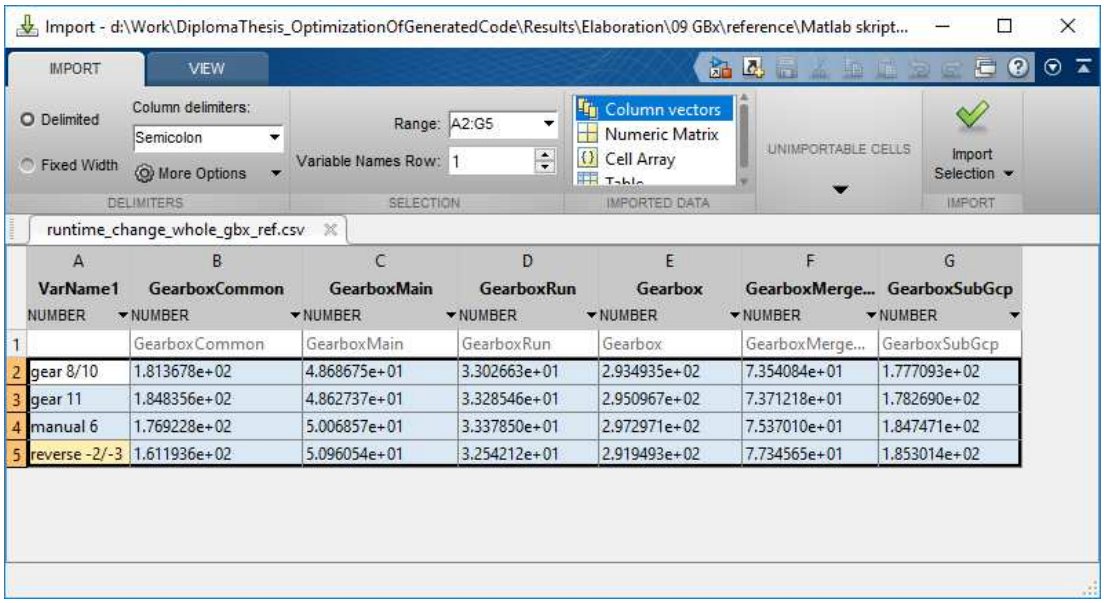
2. je úsek, kdy vozidlo jede stále na plný plyn v před a má zařazený 11. rychlostní stupeň. Úsek začíná ve chvíli, kdy skončí úsek číslo 1 tedy, když se požadovaná

rychlost nastaví na hodnotu 11. Konec intervalu nastává, když se požadovaná rychlost *yGear_tgt* nastaví z jedenáctky hodnotu 12.

3. V tomto intervalu se vozidlo stále pohybuje dopředu, ale již v manuálním módu řazení a pohybuje se na zařazený šestý rychlostní stupeň.

4. Poslední interval k porovnání je jízda ve zpětném směru s řazením ze druhého na třetí rychlostní stupeň. Úsek začíná zařazením druhého rychlostního stupně a končí přeřazením ze třetího rychlostního stupně na stupeň čtvrtý.

Zpracování naměřených signálů bylo prováděno v MATLABu, kde byl vytvořený skript, který zpracovával automaticky signály vždy z jedné sady měření (referenční nebo ze změněné verze softwaru). Nejprve jsou načteny signály postupně ze všech měření. Ty jsou po načtení přenásobeny tak, aby jejich hodnoty odpovídaly mikrosekundám (protože rozlišení měření je u většiny signálů jiné, viz. sekce 5.2). Následně jsou data rozdělena do požadovaných intervalů. Po rozdělení je pro každý signál vypočtena střední a maximální hodnota. Na závěr jsou hodnoty předány funkci, která je uloží do tabulky v **.csv* souboru, který je vidět otevřený v *MATLAB*u na obrázku 5.3. Uvnitř funkce je možné si specifikovat, které signály budou do souboru zapsány.



	A	B	C	D	E	F	G
	VarName1	GearboxCommon	GearboxMain	GearboxRun	Gearbox	GearboxMerge...	GearboxSubGcp
	NUMBER	NUMBER	NUMBER	NUMBER	NUMBER	NUMBER	NUMBER
1		GearboxCommon	GearboxMain	GearboxRun	Gearbox	GearboxMerge...	GearboxSubGcp
2	gear 8/10	1.813678e+02	4.868675e+01	3.302663e+01	2.934935e+02	7.354084e+01	1.777093e+02
3	gear 11	1.848356e+02	4.862737e+01	3.328546e+01	2.950967e+02	7.371218e+01	1.782690e+02
4	manual 6	1.769228e+02	5.006857e+01	3.337850e+01	2.972971e+02	7.537010e+01	1.847471e+02
5	reverse -2/-3	1.611936e+02	5.096054e+01	3.254212e+01	2.919493e+02	7.734565e+01	1.853014e+02

Obrázek 5.3: Ilustrační obrázek vygenerovaného *.csv* souboru otevřeného v MATLABu, byly zde vybrány hodnoty pro moduly z komponenty softwaru Gearbox. Ve sloupcích jsou zobrazeny hodnoty odpovídající jednotlivým modulům softwaru, naopak jednotlivé řádky ukazují hodnoty pro měřené intervaly.

Kapitola 6

Optimalizační metody

V kapitole 3 je zmíněno, že vzhledem k dostupným systémovým zdrojům a prioritám firmy, kde je práce vypracována, budou navržená řešení orientovaná více na optimalizaci rychlosti běhu aplikace, než na snížení nároků na paměť.

6.1 Návrh metod pro zlepšení efektivity generovaného kódu

Po vyřešení předešlých problémů, jako jsou způsoby měření dat či jejich zpracování, je možné přistoupit k návrhům metod modelování, které by mohly vést k optimalizaci vygenerovaného kódu. Po delším rozboru byly navrženy následující čtyři možnosti, které budou v následujících sekcích (6.2, 6.3, 6.4) podrobněji rozebrány.

6.1.1 Nastavení programu Targetlink

Program TargetLink nabízí mnoho nejrůznějších nastavení a optimalizací jak generovat kód, díky kterým by mohlo být dosaženo lepšího a rychlejšího kódu. Tahle varianta se nabízela k analýze jako první.

6.1.2 Přemodelování "if–else"podmínek ve Stateflow diagramech

Výše bylo zmíněno, že v modelu se objevuje hodně situací, kdy je podmínková *if–else* logika simulována pomocí *Stateflow*. V těchto diagramech se dají podmínky modelovat více způsoby, je zde tedy prostor pro analýzu, která z nich vede ve výsledku k „čistějšímu“ kódu a nebo zda *TargetLink* vygeneruje vše stejně.

6.1.3 Nahrazení parametrů pomocnými proměnnými

V modelu a převážně v částech diagramů *Stateflow* jsou často používané takzvané *datafield parametry*. Jedná se o parametry, které mohou jednoduchou změnou hodnoty změnit chování celé převodovky. Mohou mít i různé úrovně zabezpečení.

- Parametry, které může nastavovat pouze výrobce.
  Jedná se o parametry, kterými je možné například nastavit charakteristiky zařízení podle přání a specifikací zákazníků, každý odběratel, má jiné požadavky na chování převodovky.
- Parametry, které může nastavovat zákazník (výrobce kamionů).
  Výrobce tak může například specifikovat chování pro daný typ vozidla či motorizaci.

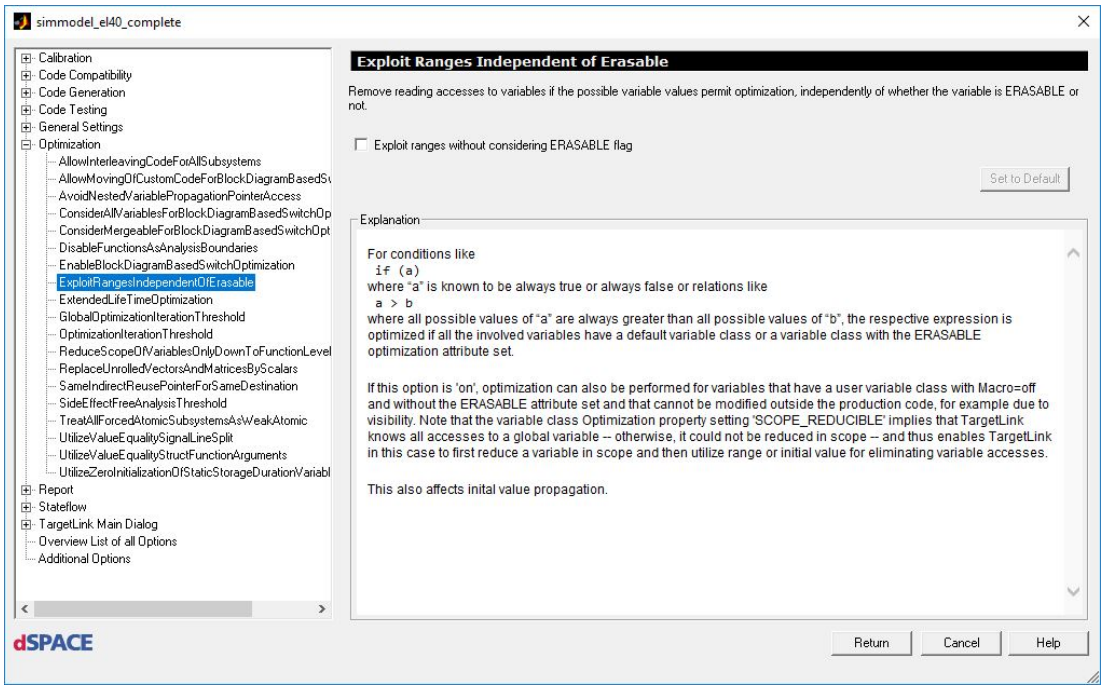
- Parametry, které může nastavovat sám uživatel,
  nejčastěji rovnou při jízdě, jako příklad lze uvést u osobních aut možnost nastavení různých režimů jízdy.

Tyto parametry jsou uloženy ve flash ROM paměti, kam je přístup programu, v porovnání s operační pamětí RAM, pomalý. Princip návrhu optimalizace spočívá v nalezení míst v softwaru, kde dochází k opakovanému čtení takového parametru, a tam ho nahrát do pomocné proměnné, která již bude uložena v rychlejší RAM paměti. Všechna následující čtení budou pak probíhat rychleji.

6.2 Analýza nastavení programu Targetlink

Jedna z prvních myšlenek, jak vylepšit generovaný kód ze Simulinku, byla použít nativní možnosti pro generování kódu, které TargetLink nabízí. Nástroj samotný skýtá mnoho nastavení, které upravují proces generování kódu a přímo ovlivňují, jak bude výsledný zdrojový kód vypadat. Jedněmi z těchto nastavení jsou i možnosti, které nabízejí určitou optimalizace kódu už během generování.

Bylo provedeno mnoho různých variant nastavení, zároveň zkoušeny různé kombinace, kdy byla současně některá nastavení zapnuta a při dalším generování kódu naopak vypnuta. Výsledné vygenerované kódy byly mezi sebou porovnány, ale v žádné z variant vygenerovaného kódu nebyly nalezeny změny, které by měly optimalizace způsobit. Vygenerované soubory byly až na jednu výjimku identické. Ta nastala u nastavení, kdy *TargetLink* umožňuje deklarovat pomocné proměnné co nejblíže místa, kde jsou použity. To může vést v některých případech i k tomu, že daná proměnná deklarována vůbec nebude a tím dojde k šetření paměti i procesorového času. Po zrušení této možnosti se všechny tyto proměnné vygenerovaly na začátek hlavní funkce, k jejich deklaraci docházelo tedy v každém cyklu stejně. Toto nastavení bylo již společně s několika dalšími již



Obrázek 6.1: Ukázka dialogového okna s nabídkou možností nastavení generovaného kódu v programu TargetLink.

v oddělení používáno, nelze je tedy označit za nové pravidlo, které by se mělo používat,

tato analýza však potvrdila potenciál tohoto nastavení. Na obrázku 6.1 je pak vidět grafické rozhraní s popisem jednotlivých nastavení a možností je vypnout či zapnout.

Příčina toho, že nebyly zpozorovány žádné výsledky ostatních nastavení je vysvětlována velkou komplexitou části modelu, na které byla různá nastavení testována, a malým výskytem výpočetních částí, kde by se optimalizace mohly více uplatnit (tato část modelu se zaměřovalo více na podmínkovou logiku spíše než na výpočty).

Byla zvážena i možnost špatného způsobu nastavení a varianta, že *TargetLink* nebral vybrané optimalizace na vědomí. Tato možnost však byla vyloučena po té, co byl v hlavičce vygenerovaného kódu objeven výpis použitých nastavení.

6.3 Re–modelace podmínek v diagramech Stateflow

V sekci s návrhy pro zlepšení modelování 6.1 bylo zmíněno, že velká část rozhodovací „if–else“ logiky je přesunuta z úrovně simulinkovských bloků do diagramů *Stateflow*. Psaní podmínek touto cestou přináší velkou výhodu v přehlednosti. Je zde poměrně dobře vidět, kterou cestou se bude software ubírat, když bude splněna konkrétní podmínka.

Jako při programování standardním způsobem i zde existuje více způsobů, jak danou problematiku namodelovat.

V modelu se ve výchozím stavu nacházelo mnoho soustav podmínek složených z logických *and* a *or*. Mnoho z těchto podmínek mělo jako jednu ze svých součástí stejnou podmínku, která byla použita i v ostatních větvích rozhodování. Příklad je uveden na následujícím obrázku 6.2. Z obrázku 6.2 je dobře zjevné, že podmínka zobrazená ve

Obrázek 6.2: Ilustrační obrázek, kde je uveden příklad modelování *if–else* podmínek, ve kterých je každá jednotlivá podmínka složená z několika dalších. Je možné si povšimnout, že v každé složené podmínce figuruje jedna, která je pro všechny stejná.

výpisu číslo 6.1

Listing 6.1: Podmínka která se objevuje ve více složených podmínkách.

```
phiAccPedVirtual >= Get_Fsh_D_phiAccPedFastShf_min(
    yShigtDynamicDrive_act)
```

by mohla být napsána jako první před všemi ostatními, aby byla provedena pouze jednou. Zároveň i podmínky ve výpisu číslo 6.3

```
yGear_tgt < yGear_tgtold
yGear_tgt > yGear_tgtold,
```

by mohly být zaintegrovány postupně, aby se mohlo předejít jejich případnému vícenásobnému volání. *TargetLink* bohužel sám neumí takové podmínky nalézt a optimalizovat, z toho důvodu by mohlo být pro rychlost softwaru prospěšné takovou soustavu namodelovat jinak. Ve výpisu číslo 6.2 je možné vidět zdrojový kód vygenerovaný z ukázky modelu ve *Stateflow* diagramu na předchozím obrázku číslo 6.2.

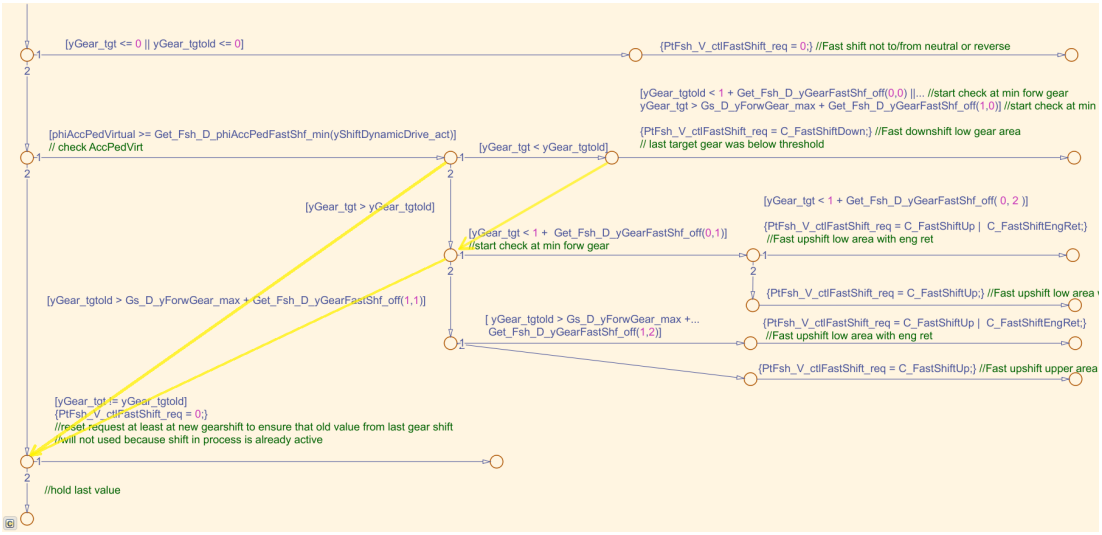
Listing 6.2: Ukázka vygenerovaného kódu pomocí TargetLinku odpovídající části modelu. Jedná se o část od podmínky `yGear_tgt <= 0 || yGear_tgtold <= 0`. Inkriminované podmínky jsou v kódu označeny červenou barvou jako string.

```
if ((ExtIn_PtFsh_Get_yGear_tgt() <= 0) || (X_Sh1_yGear_tgt_old <=
    0)) {
    /* Fast shift not to/from neutral or reverse */
    IF_Set_PtFsh_V_ctlFastShift_req(0);
}
else {
    /* check AccPedVirt */
    if ("(ExtIn_PtFsh_Get_yGear_tgt() < X_Sh1_yGear_tgt_old)" && (
        X_Sh1_yGear_tgt_old
    < (1 + SF_Get_Fsh_D_yGearFastShf_off(0, 0))) &&
    "(ExtIn_PtFsh_Get_phiAccPedVirtual() >=
        SF_Get_Fsh_D_phiAccPedFastShf_min(
        ExtIn_PtFsh_Get_yShiftDynamicDrive_act()))")
    {
    /* Fast downshift low gear area
        last target gear was below threshold */
    IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftDown);
    }
    else {
    /* check AccPedVirt */
    if ("(ExtIn_PtFsh_Get_yGear_tgt() < X_Sh1_yGear_tgt_old)" &&
        (ExtIn_PtFsh_Get_yGear_tgt() > (
            ExtIn_PrjCmn_Get_Gs_D_yForwGear_max() +
        SF_Get_Fsh_D_yGearFastShf_off(1, 0))) &&
        "(ExtIn_PtFsh_Get_phiAccPedVirtual() >=
            SF_Get_Fsh_D_phiAccPedFastShf_min(
            ExtIn_PtFsh_Get_yShiftDynamicDrive_act()))")
        {
        /* Fast downshift upper gear area
            last target gear was below threshold */
        IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftDown);
    }
    else {
        /* check AccPedVirt */
        if ("(ExtIn_PtFsh_Get_yGear_tgt() > X_Sh1_yGear_tgt_old)"
            &&
        (ExtIn_PtFsh_Get_yGear_tgt() < (1 +
            SF_Get_Fsh_D_yGearFastShf_off(0,
        1))) && "(ExtIn_PtFsh_Get_phiAccPedVirtual() >=
            SF_Get_Fsh_D_phiAccPedFastShf_min(
            ExtIn_PtFsh_Get_yShiftDynamicDrive_act()))")
```

```
                    {
                    if (ExtIn_PtFsh_Get_yGear_tgt() < (1 +
                    SF_Get_Fsh_D_yGearFastShf_off(0, 2))) {
                        /* Fast upshift low area with eng ret */
                        IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp |
                            C_FastShiftEngRet);
                    }
                    else {
                        /* Fast upshift low area without eng ret */
                        IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp);
                    }
                }
                else {
                    /* hold last value
                        check AccPedVirt */
                    if ("(ExtIn_PtFsh_Get_yGear_tgt() >
                        X_Sh1_yGear_tgt_old)" &&
                    (X_Sh1_yGear_tgt_old > (
                        ExtIn_PrjCmn_Get_Gs_D_yForwGear_max() +
                    SF_Get_Fsh_D_yGearFastShf_off(1, 1))) &&
                    "(ExtIn_PtFsh_Get_phiAccPedVirtual() >=
                        SF_Get_Fsh_D_phiAccPedFastShf_min(
                        ExtIn_PtFsh_Get_yShiftDynamicDrive_act()))")
                    {
                        if (X_Sh1_yGear_tgt_old > (
                            ExtIn_PrjCmn_Get_Gs_D_yForwGear_max() +
                        SF_Get_Fsh_D_yGearFastShf_off(1, 2))) {
                        /* Fast upshift upper area with eng ret */
                        IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp |
                            C_FastShiftEngRet);
                        }
                        else {
                        /* Fast upshift upper area without eng ret */
                        IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp);
                        }
                    }
                }
            }
        }
    }
```

6.3.1 Podmínka bez větve „else“

V prvním případě přemodelování „if–else“ podmínkové soustavy došlo k postupnému řešení podmínek poznamenaných výše tak, aby jejich postupným řešením zamezilo jejich vícenásobnému volání. Příklad řešení je k nalezení na následujícím obrázku 6.3. Pro správnou funkcionalitu *Stateflow* vyžaduje při modelování podmínek nakreslit vždy i „else“ větev, aby algoritmus při nesplnění podmínky věděl, kam se má vrátit. Na obrázku v příkladu je vidět, že tyto „else“ šipky v diagramu chybí.



Obrázek 6.3: Ilustrační obrázek, kde je uveden příklad modelování *if–else* podmínek. Došlo zde ke sjednocení podmínek. Žlutě jsou zde ale vyznačeny chybějící „else“ šipky.

Při generování *C* kódu vyřešil TargetLink tento problém přidáním pomocné proměnné `Aux_ui8`, které přiřazuje hodnotu podle větve, do které se dostal, a nemůže projít dál. Následně se podle této hodnoty rozhoduje, kde bude ve vykonávání algoritmu pokračovat.

Listing 6.3: Ukázka vygenerovaného kódu pomocí TargetLinku odpovídající části modelu. Cílem je zde ukázat jak TargetLink vyřešil problém s chybějícími „else“ šipkami přidáním pomocné `Aux_ui8`. Jedná se opět o část kódu od podmínky `yGear_tgt <= 0 || yGear_tgtold <= 0`. Pomocná proměnná je ve vygenerovaném kódu označena červeně jako string. (Pro zkrácení výpisu zde střední část kódu chybí.)

```
if ((ExtIn_PtFsh_Get_yGear_tgt() <= 0) || (X_Sh1_yGear_tgt_old <=
    0)) {
   /* Fast shift not to/from neutral or reverse */
   IF_Set_PtFsh_V_ctlFastShift_req(0);
}
else {
   /* SLLocal: Default storage class for local variables | Width: 8
       */
   "ui8 Aux_ui8";

   /* check AccPedVirt
      evaluate function */
   if (ExtIn_PtFsh_Get_phiAccPedVirtual() >=
    Get_Fsh_D_phiAccPedFastShf_min(
        ExtIn_PtFsh_Get_yShiftDynamicDrive_act())) {
      /* start check at min forw gear
         evaluate function
         evaluate function */
      if ((ExtIn_PtFsh_Get_yGear_tgt() < X_Sh1_yGear_tgt_old) &&
       ((X_Sh1_yGear_tgt_old < (1 + Get_Fsh_D_yGearFastShf_off(0,
           0))) ||
       (ExtIn_PtFsh_Get_yGear_tgt() > (
           ExtIn_PrjCmn_Get_Gs_D_yForwGear_max() +
       Get_Fsh_D_yGearFastShf_off(1, 0))))) {
         /* Fast downshift low gear area
            last target gear was below threshold */
         IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftDown);
```

```
                "Aux_ui8 = 2";
            }
            else {
                if (ExtIn_PtFsh_Get_yGear_tgt() > X_Sh1_yGear_tgt_old) {
                    /* start check at min forw gear
                        evaluate function */
                    if (ExtIn_PtFsh_Get_yGear_tgt() < (1 +
                        Get_Fsh_D_yGearFastShf_off(0,
                     1))) {
                        /* evaluate function */
                        if (ExtIn_PtFsh_Get_yGear_tgt() < (1 +
                         Get_Fsh_D_yGearFastShf_off(0, 2))) {
                            /* Fast upshift low area with eng ret */
                            IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp |
                             C_FastShiftEngRet);
                        }
                        else {
                            /* Fast upshift low area without eng ret */
                            IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp);
                        }
                        "Aux_ui8 = 2";
                    }
                    else {

                        ...

                    "Aux_ui8 = 0";
                }
            }
        }
        else {
            "Aux_ui8 = 0";
        }
        if ("Aux_ui8 <= 1") {
            /* hold last value */
            if (ExtIn_PtFsh_Get_yGear_tgt() != X_Sh1_yGear_tgt_old) {
                /* reset request at least at new gearshift to ensure that
                    old value from
                    last gear shift
                    will not used because shift in process is already
                        active */
                IF_Set_PtFsh_V_ctlFastShift_req(0);
            }
        }
    }
```

Z uvedené ukázky vygenerovaného kódu je sice patrné, že došlo k eliminaci nadbytečného řešení cílených podmínek, navzdory tomu výsledný kód nabobtnal o nutnost řešení absence „else“ větví v modelu. Od začátku bylo jasné, že se nebude jednat o ideální řešení tohoto problému, bylo ale zajímavé pozorovat, jak se TargetLink při generování kódu s nastalou situací vypořádal.

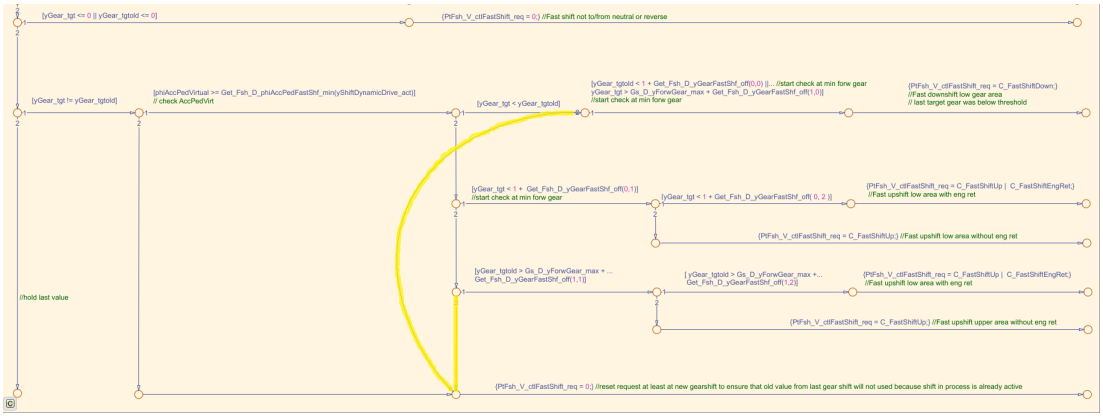
6.3.2 Podmínka s „else“ šipkami do jednoho uzlu

V předchozím případě bylo ukázáno, že TargetLink dokáže vygenerovat strukturu logických „if–else“ podmínek tak, jak je vytvořen jejich model. Problém nastává pokud v modelu nejsou přesně definované „else“ větve podmínek. Ve vygenerovaném kódu se sice zamýšlená funkcionalita objeví, ale zvolené řešení se zcela jistě nedá označit za optimální.

Z toho důvodu bylo rozhodnuto, že budou vytvořeny ještě dvě verze modelu, které

budou obsahovat tyto „else“ šipky, každá bude modelována trochu jinak. Z obou verzí bude vygenerován kód a ten mezi sebou porovnán.

U tohoto prvního přístupu je zvolen postup, kdy je vytvořen pouze jediný „else“ uzel, do kterého budou přivedeny podmínky, pokud nebude možná žádná další varianta. Diagram *Stateflow* se zvoleným řešením je možné vidět na následujícím obrázku 6.4.



Obrázek 6.4: Návrh diagramu *Stateflow*, kdy jednotlivé podmínky jsou za sebou kaskádně seřazeny. Pro varianty, kdy po nesplnění požadavku k průchodu nevede už z uzlu žádná cesta, byly přidány šipky, které nasměrují běh algoritmu. Cesty v tomto případě vedou pouze do jednoho uzlu. Žlutě jsou v modelu vyznačeny navržené „else“ větve.

Po namodelování byl ze *Stateflow* diagramu vygenerován kód, který je k nahlédnutí ve výpisu níže 6.4. V tomto případě byl zdrojový kód vygenerován podle očekávání s do sebe vnořenými podmínkami a bez přídavných pomocných proměnných.

Listing 6.4: Ukázka vygenerovaného zdrojového kódu z modelu kam byly přidány šipky lépe definující směr běhu programu. Červeně jsou zvýrazněny přesunuté podmínky.

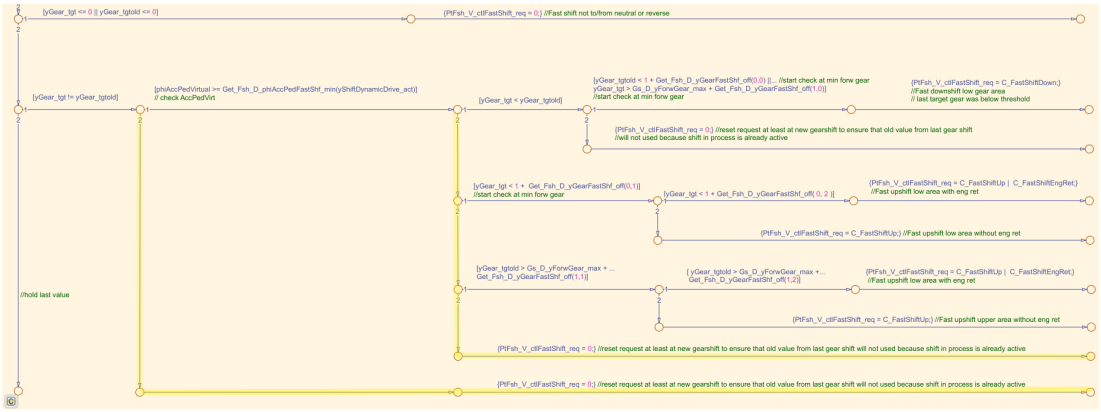
```
if ((ExtIn_PtFsh_Get_yGear_tgt() <= 0) || (X_Sh1_yGear_tgt_old <= 0))
    {
   /* Fast shift not to/from neutral or reverse */
   IF_Set_PtFsh_V_ctlFastShift_req(0);
}
else {
   if (ExtIn_PtFsh_Get_yGear_tgt() != X_Sh1_yGear_tgt_old) {
      /* check AccPedVirt
         evaluate function */
      if ("ExtIn_PtFsh_Get_phiAccPedVirtual() >=
          Get_Fsh_D_phiAccPedFastShf_min(
          ExtIn_PtFsh_Get_yShiftDynamicDrive_act())") {
         if ("ExtIn_PtFsh_Get_yGear_tgt() < X_Sh1_yGear_tgt_old") {
            /* start check at min forw gear
               evaluate function
               evaluate function */
            if ((X_Sh1_yGear_tgt_old < (1 + Get_Fsh_D_yGearFastShf_off
                (0, 0))) ||
             (ExtIn_PtFsh_Get_yGear_tgt() > (
                 ExtIn_PrjCmn_Get_Gs_D_yForwGear_max()
              + Get_Fsh_D_yGearFastShf_off(1, 0)))) {
               /* Fast downshift low gear area
                  last target gear was below threshold */
               IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftDown);
            }
```

```
            else {
               /* reset request at least at new gearshift to ensure
                   that old value
                   from last gear shift will not used because shift in
                       process is
                  already active */
               IF_Set_PtFsh_V_ctlFastShift_req(0);
            }
         }
         else {
            /* start check at min forw gear
               evaluate function */
            if (ExtIn_PtFsh_Get_yGear_tgt() < (1 +
               Get_Fsh_D_yGearFastShf_off(0,
             1))) {
               /* evaluate function */
               if (ExtIn_PtFsh_Get_yGear_tgt() < (1 +
                Get_Fsh_D_yGearFastShf_off(0, 2))) {
                  /* Fast upshift low area with eng ret */
                  IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp |
                   C_FastShiftEngRet);
               }
               else {
                  /* Fast upshift low area without eng ret */
                  IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp);
               }
            }
            else {
               /* evaluate function */
               if (X_Sh1_yGear_tgt_old > (
                   ExtIn_PrjCmn_Get_Gs_D_yForwGear_max() +
                Get_Fsh_D_yGearFastShf_off(1, 1))) {
                  /* evaluate function */
                  if (X_Sh1_yGear_tgt_old > (
                      ExtIn_PrjCmn_Get_Gs_D_yForwGear_max()
                    + Get_Fsh_D_yGearFastShf_off(1, 2))) {
                     /* Fast upshift low area with eng ret */
                     IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp |
                      C_FastShiftEngRet);
                  }
                  else {
                     /* Fast upshift upper area without eng ret */
                     IF_Set_PtFsh_V_ctlFastShift_req(C_FastShiftUp);
                  }
               }
               else {
                  /* reset request at least at new gearshift to ensure
                       that old va
                     lue from last gear shift will not used because
                         shift in proce
                     ss is already active */
                  IF_Set_PtFsh_V_ctlFastShift_req(0);
               }
            }
         }
      }
      else {
         /* reset request at least at new gearshift to ensure that old
              value from
            last gear shift will not used because shift in process is
                already acti
            ve */
         IF_Set_PtFsh_V_ctlFastShift_req(0);
```

```
        }
    }
}
```

6.3.3 Podmínka s „else“ šipkami do více uzlů

V této variantě modelování podmínek bylo postupováno obdobně jako v předchozím řešení. Byla opět vytvořena kaskádní struktura vnořených podmínek tak, aby podmínky uvedené výše ve výpisech 6.1 a 6.3 byly řešeny nejvýše jednou při jakémkoli nastavení okolností. Oproti předchozímu případu, zde bylo využito více uzlů, do kterých ústily šipky „else“ možností. Výsledný způsob realizace je opět zdokumentován na následujícím obrázku, který znázorňuje v pořadí už čtvrtou variantu možného řešení soustavy podmínek ve *Stateflow* diagramu.



Obrázek 6.5: Návrh diagramu *Stateflow*, kdy podobně jako na obrázku 6.4 jsou jednotlivé části podmínek řazeny za sebou a ve všech podmínkových uzlech jsou namodelovány řešení pro možné „else“ varianty. V tomto případě má každá „else“ varianta svojí vlastní cestu s uzlem přes který prochází. Přidané šipky jsou opět zvýrazněny žlutě.

V tomto případě návrhu modelu soustavy podmínek byl vygenerován TargetLinkem zcela identický zdrojový kód jako v předchozím případě, tedy 6.4. Rozdíl byl pouze v hlavičce souboru a to v čase vygenerování.

6.3.4 Shrnutí

V této sekci bylo nastíněno, jakým způsobem je možné pojmout modelování podmínek v diagramech *Stateflow* a zároveň bylo z analyzováno, jak k tomuto problému přistupuje nástroj pro generování zdrojového kódu TargetLink.

Z navrhovaných variant se z hlediska náročnosti běhu programu jeví nejlépe poslední dvě metody modelování 6.3.2 a 6.3.3. Vygenerovaný kód byl u obou způsobů modelování totožný a bylo dosaženo požadovaného rozdělení struktury podmínek.

Z hlediska praktičnosti je pak zřejmě lepší metoda modelování 6.3.3, která nabízí oproti druhé variantě větší přehlednost modelu, obzvlášť u složitějších podmínek.

Na závěr bylo provedeno i měření náročnosti běhu aplikace pro ověření dopadu provedených změn. Naměřené hodnoty však zůstaly přibližně stejné a vlivem míry nepřesnosti měření není možné rozhodnout zda došlo ke zlepší, či nikoli. Tuto skutečnost je možné vysvětlit malým rozsahem provedených změn v rámci této části kódu a také celkově nízkým dopadem celého modulu na běh procesoru, který je se pohybuje na hranici rozlišení měřitelnosti. Bylo nutné přenastavit rozlišení měření na $1\mu s$, ale po

této změně se průměrné hodnoty náročnosti této části kódu se před změnami i po nich pohybovaly kolem hodnoty $0.7\mu s$. U tak nízké hodnoty nemůže být bohužel rozhodnuto zda ke zlepšení došlo. Tabulku s naměřenými hodnotami je možné nalézt zde 6.1.

6.4 Nahrazení parametrů dočasnými proměnnými

V celém softwaru se objevují proměnné, které změnou svojí hodnoty mohou výrazně změnit charakteristiku chování celé převodovky. V textu výše, v sekci 6.1.3, bylo zmíněno že tyto parametry, jsou děleny do více skupin, podle oprávnění přístupu. Je důležité, aby si uchovávaly svoji hodnotu i po vypnutí jednotky a odpojení od napájení, proto jsou tyhle proměnné uloženy v *ROM* paměti řídicí jednotky.

Kromě dělení, které je popsáno v sekci 6.1.3, je možné tyto parametry rozdělit i podle použití.

- Online parametry
  Tento typ datafieldů je po startu aplikace načten do *RAM* paměti, kde může být změněn. Po restartu jednotky má opět výchozí hodnotu.

- EOL parametry
  Tyto parametry jsou velmi podobné chováním „online parametrům“, také se načítají po spuštění aplikace do *RAM* paměti, ale před vypnutím řídicí jednotky se ukládají do *EEPROM* paměti, kde si uchovávají svoji hodnotu i když je jednotka vypnuta.

- Offline parametry
  Tyto parametry jsou stále uloženy v *ROM* paměti, nezapisuje se do nich, jen jsou z nich vyčítány hodnoty.

Používání dat uložených ve *flash ROM* paměti za běhu aplikace má oproti používání zbytku dat, které jsou uloženy v *RAM*, jednu výraznou nevýhodu, kterou je rychlost čtení. Právě doba čtení těchto parametrů, které jsou uloženy v *ROM* paměti, může znamenat potenciální místo, kde by mohl být procesorový čas ušetřen. V modelu se vyskytují části, kde dochází k opakovanému čtení těchto proměnných na jednom místě. Jedná se například o opakované čtení v cyklech, či rozvětvených podmínkách. Možná varianta, jak vyřešit tento problém je, načíst parametr z *ROM* paměti do dočasné pomocné proměnné, která je již uložena v *RAM* paměti řídicí jednotky, kam je přístup rychlejší, a s touto proměnou pak dále pracovat v cyklech nebo jinde, kde je potřeba.

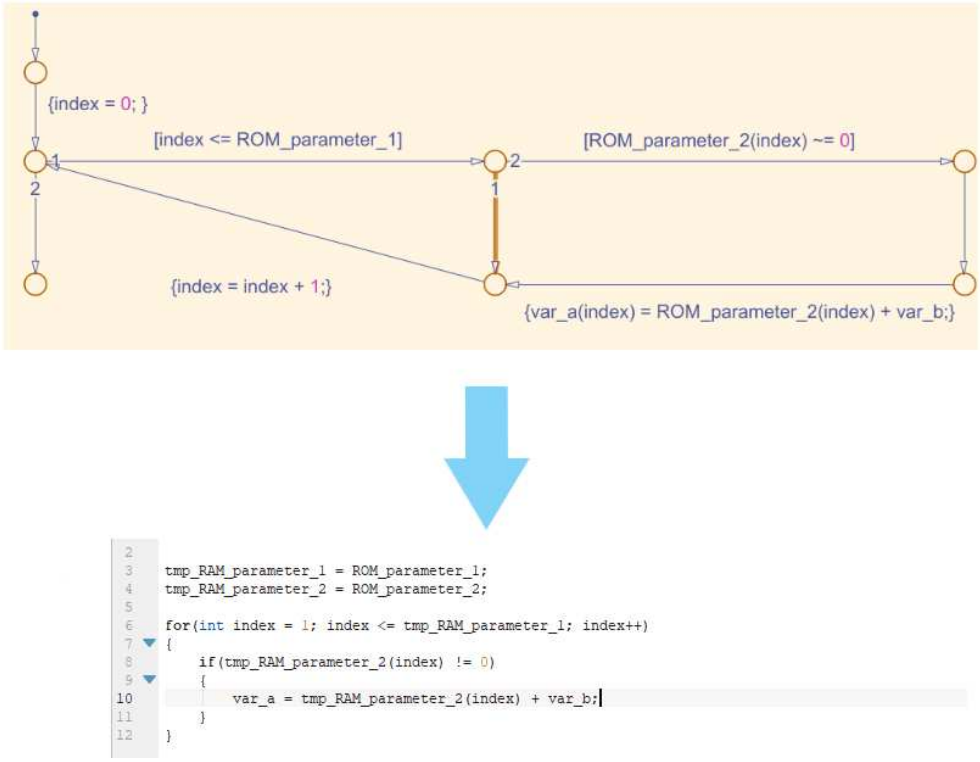
Cílem je, pokud se vyskytuje v modelu vícenásobné volání parametru z *ROM* paměti, získat situaci ilustrovanou na následujícím obrázku 6.6.

6.4.1 Způsoby zavedení dočasné proměnné

Modelovací nástroje *Simulink* a *Stateflow* nabízejí mnohem širší paletu možností, jak vyjádřit požadovanou funkcionalitu, než většina ostatních programovacích jazyků. Pouhé přiřazení jedné proměnné k jinému názvu může být provedeno více způsoby.

6.4.1.1 Přiřazení ve *Stateflow*

Vzhledem k tomu, že většina vícenásobných volání je použita ve *Stateflow*, je nahrazení těchto parametrů přímo uvnitř diagramů první a možná nejednodušší cestou, jak přiřazení dosáhnout. Tato varianta neskýtá mnoho možností volby, probíhá podobně jako v ostatních programovacích jazycích. Na následujícím obrázku 6.7 je možné vidět

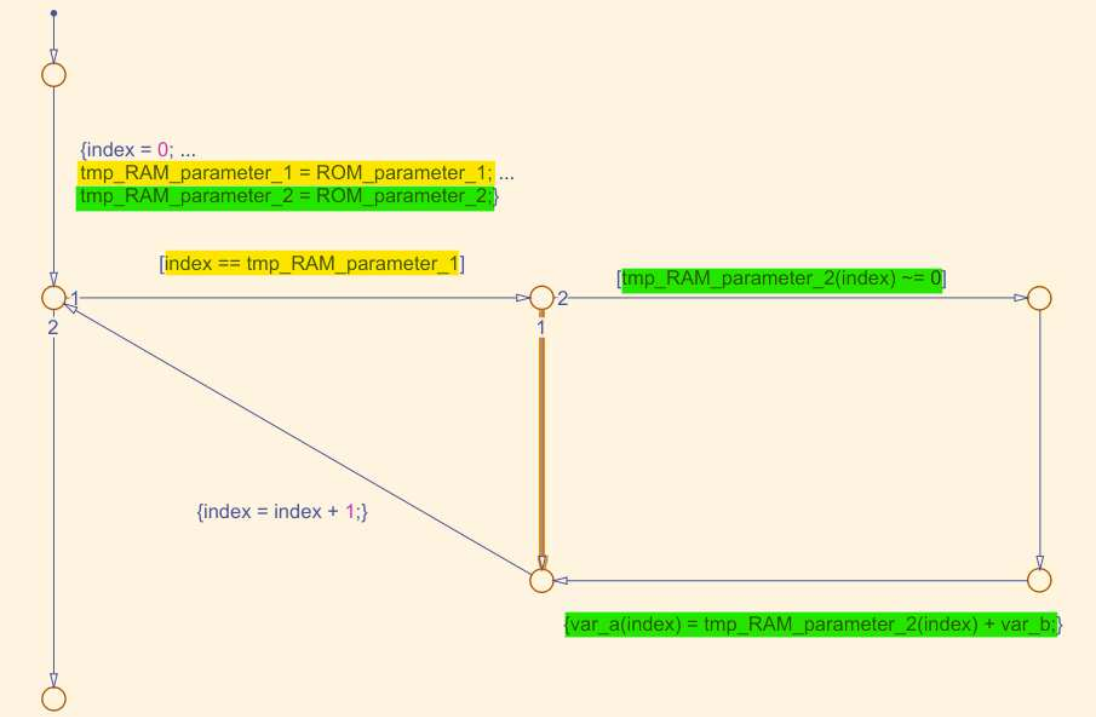


Obrázek 6.6: Ilustrační příklad jak by měl převod vypadat.

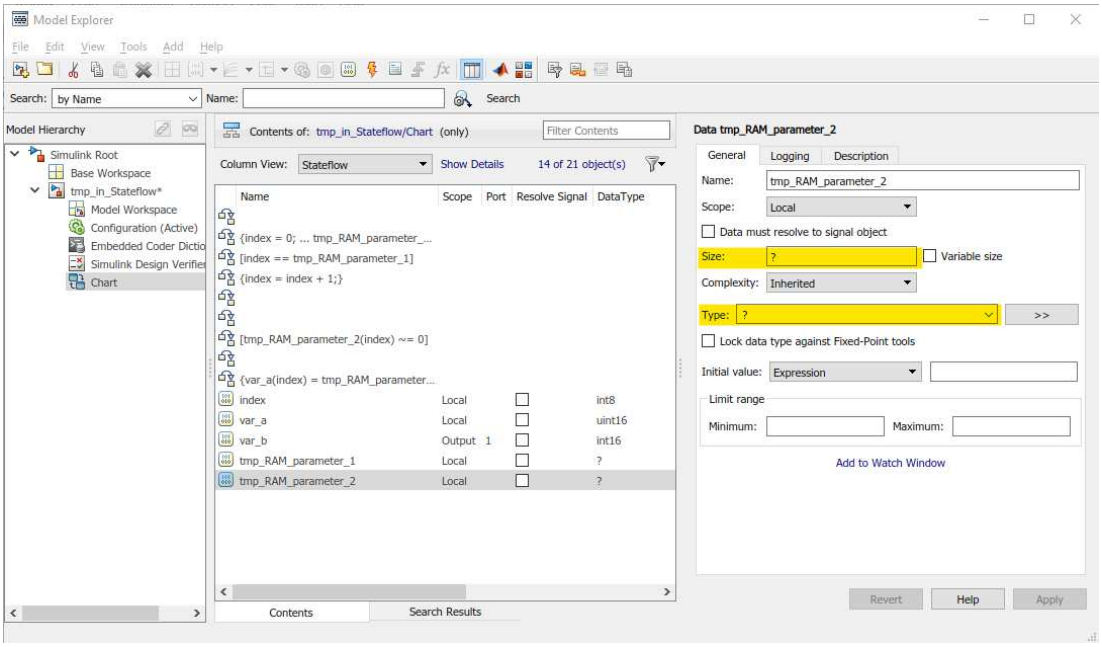
modelový příklad nahrazení parametru čteného z *ROM* paměti dočasnou proměnnou, která již je při použité čtená z rychlejší *RAM* paměti.

Při modelování tohoto způsobu řešení nastává pro modeláře jeden problém při přidávání těchto pomocných proměnných do *Model Explorer*u, kde musí být definovány proměnné použité v diagramu. Zde musí být definovány datové typy zaváděných proměnných a případně i jejich velikost 6.8, které by se měly schodovat s nastavením původních parametrů. Tyto informace se ale vyskytují ve speciálních *.XML* souborech, kde jsou parametry uložené v *ROM* definované. Hledání správných datových typů může být tedy náročné a při modelování více takových situací náročné i časově.

Vygenerovaný kód následně vypadá přesně podle očekávání tak, jak je namodelovaný ve *Stateflow*.



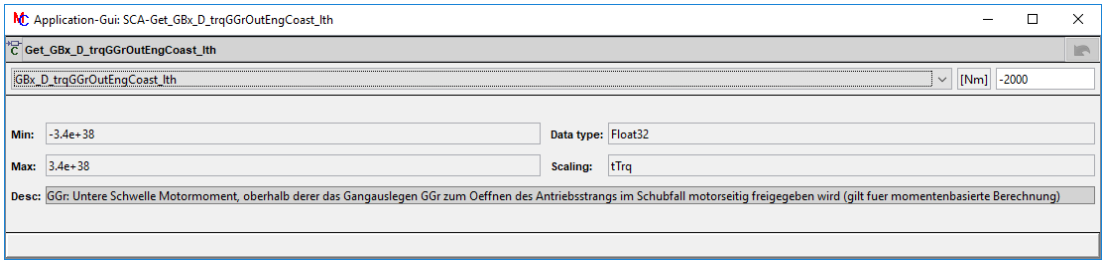
Obrázek 6.7: Modelový příklad nahrazení parametrů čtených z *ROM* proměnnými uloženými v *RAM*. Zeleně a žlutě jsou označeny nahrazené proměnné a jejich následné volání.



Obrázek 6.8: Ukázka prostředí *Model Explorer*u, kde je nutné nastavit všechny informace o použitých proměnných, aby diagram *Stateflow* mohl fungovat správně. Žlutě jsou na obrázku vyznačena nastavení proměnných, které by se musely extra vyhledávat.

6.4.1.2 Přiřazení v Simulinku

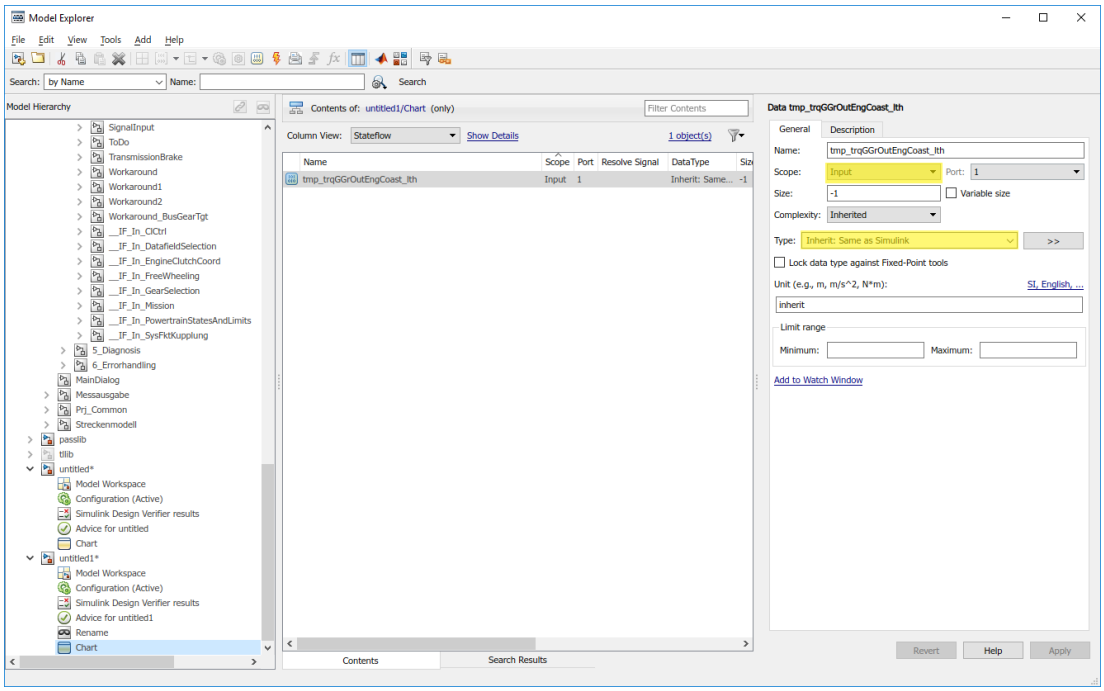
Další variantou, jak je možné přiřazení v modelu provést, je přiřazení před samotným *Stateflow* přímo v prostředí *Simulink*. To může být provedeno tak, že požadovaná proměnná je zavolána speciálním bločkem přímo v *Simulink*u. Jedná se o *constant*–blok, který automaticky generuje hodnoty na základě informace obsažené v *.XML* souboru. Tento blok má v sobě uveden název parametru a jeho specifikaci, jako jsou varianty parametru, jeho datový typ, škálování, maximum a minimum a může být uveden i jeho popis (obrázek 6.9).



Obrázek 6.9: Nastavení bloku, který slouží v prostředí *Simulink* k volání datafield parametru.

Následně je nutné, aby byl signál přerušen jiným blokem, který nezmění hodnotu signálu, může být použit například jednoduchý blok *mux*, blok *gain* s jednotkovým zesílením nebo, pro větší přehlednost, blok u interní knihovny *rename*, jedná se o jednoduchý subsystém, který v sobě má pouze blok *mux*, který přerušuje signál. Jde v podstatě o stejné řešení jako u použití samotného bloku *mux*, ale modelování pomocí tohoto typu bloků má výhodu, že uživatel ví, proč tam blok je a přispívá k lepší čitelnosti modelu.

Výhoda tohoto řešení spočívá při nastavování proměnné v *Model Exlorer*u diagramu *Stateflow*. Zde jde totiž u datového typu proměnné zvolit možnost *Inherit: Same as Simulink*. Zvolením tohoto nastavení diagram převezme datový typ, který má parametr specifikován v bloku, ve kterém se volá. Odpadá tedy nutnost vyhledávat datový typ externě pro každý použitý datafield parametr. Pokud je z nějakého důvodu nutné datový typ zvolit pevně, je snadné ho po otevření bloku v *Simulink*u vyhledat.



Obrázek 6.10: Nastavení proměnné v *Model Explorer*u. V tomto případě je nutné zvolit možnost *Input*, aby se vytvořil u diagramu port, ke kterému lze signál připojit. Je zde vidět i zvolená možnost datového typu proměnné.

Použití různých typů přerušovacích bloků má vliv na vygenerovaný kód. Při použití bloků *mux* a *rename* dojde k přejmenování signálu jak v modelu tak i v kódu. Drobný problém je, že *TargetLink* v tomto případě není schopný vyčíst z „drátu“ v *Simulink*u název a jméno pomocné proměnné ve zdrojovém kódu vygeneruje podle své interní logiky. Z funkčního hlediska to není na závadu, ale takto pojmenovaná proměnná ubírá čitelnosti vygenerovaného kódu viz. výpis 6.5.

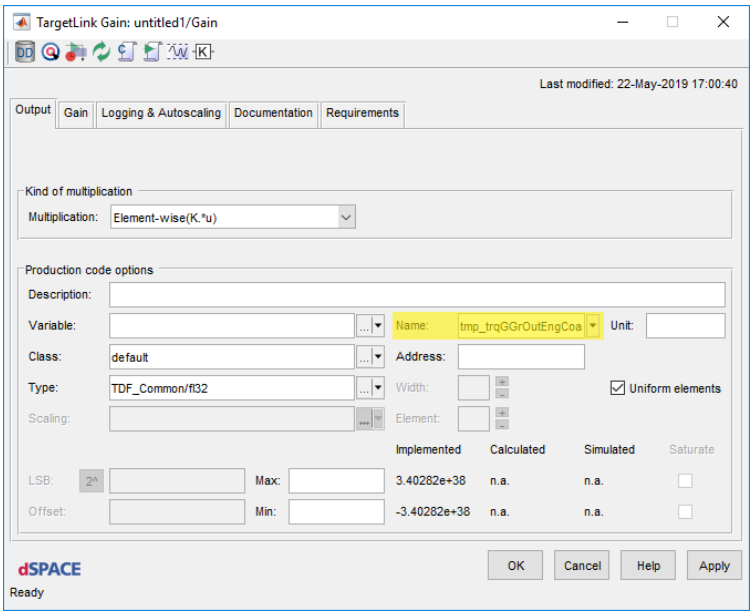
Listing 6.5: Ukázka kódu vygenerovaného při použití *mux* nebo *rename* bloků k přiřazení datefield parametru do nové proměnné.

```
/*   Gain: GBxMainGcp/GBxMainGcp/b_TrqUnloaded/a_bGGrShfOutTrqUnloaded/
     Subsystem/Get_GBx_D_trqGGrOutEngRetCoast_lth */
SMkpkbLdynV1_OutPort_a = Get_GBx_D_trqGGrOutEngRetCoast_lth();
```

Při přiřazování dočasné proměnné pomocí bloku *Gain* je umožněno použít rozšiřující nastavení bloku, kde je možné zadat další možnosti pro generování kódu, datový typ, škálování a především název, který má vygenerovaná a vynásobená proměnná mít 6.11. Názvy ve vygenerovaném kódu pak odpovídají požadovaným jménům proměnných. Příklad vygenerovaného kódu je možné vidět v následujícím výpisu 6.6.

Listing 6.6: Ukázka vygenerovaného zdrojového kódu kde je přiřazení datafield parametru do pomocné proměnné za použití bloku *Gain* s jednotkovým zesílením.

```
/*   Gain: GBxMainGcp/GBxMainGcp/b_TrqUnloaded/a_bGGrShfOutTrqUnloaded/
     Subsystem/Gain1
      # combined # call of function: GBxMainGcp/GBxMainGcp/b_TrqUnloaded
          /a_bGGrShfOutTrqUnloaded/Subsystem/
          Get_GBx_D_trqGGrOutEngRetCoast_lth */
tmp_trqGGrOutEngRetCoast_lth = Get_GBx_D_trqGGrOutEngRetCoast_lth();
```



Obrázek 6.11: Možnosti nastavení v *TargetLink* bloku *Gain*. Žlutě je zvýrazněna možnost zadání vlastního názvu proměnné.

6.4.2 Shrnutí

S ohledem na funkčnost a čitelnost vygenerovaného kódu se jeví možnosti použití přiřazení parametru do pomocné proměnné ve *Stateflow* a v *Simulink*u pomocí bloku *Gain* stejně. V okamžiku, kdy se bude brát na zřetel i uživatelská přívětivost při modelování, je z mého pohledu lepší použití k přiřazení postup v prostředí *Simulink* s blokem *Gain*. Při samotném modelování se provádí přibližně stejně úkonů, ale pozdější vyhledávání datového typu a ostatních informací použitého parametru je výrazně snazší.

6.5 Zhodnocení navržených optimalizačních metod

Tato sekce popisuje dopad navržených změn na běh procesoru.

6.5.1 Analýza nastavení *TargetLink*u

V této metodě byly analyzovány možnosti optimalizačních nastavení pro generování zdrojového kódu, kterými disponuje nástroj *TargetLink*. Byly vyzkoušeny různé varianty nastavení programu, ale žádná nevedla ke zlepšení již generovaného kódu. V naprosté většině případů byl vygenerován zcela shodný kód. Z toho důvodu nebylo u tohoto návrhu provedeno měření míry náročnosti dané funkce na běh procesoru. Nefunkčnost použitých nastavení byla nejspíše způsobena vysokou komplexitou testované části modelu.

6.5.2 Re–modelace podmínek v diagramech *Stateflow*

V metodě přemodelování podmínek v diagramech *Stateflow* byly vyzkoušeny různé návrhy diagramu, které měly za cíl eliminovat vícenásobné vykonávání stejných podmínek. Z logického pohledu by toto řešení mělo mít svůj potenciál a v širším použití ho jistě mít bude. V tomto případě však byly změny provedeny pouze v malé míře. Zvolená část softwaru, kde byly tyto změny aplikovány, měla sama o sobě na běh procesoru malý

dopad, z toho důvodu není z provedených měření jakýkoli dopad na runtime procesoru průkazný.

Byla provedena série patnácti měření, která byla zpracována výše zmíněným postupem 5.5. V následující tabulce 6.1 je možné si prohlédnout hodnoty v mikrosekundách před a po aplikování zvolené metody. Z uvedených hodnot je bohužel patrné, že i přesto, že bylo rozlišení přenastaveno na nejjemnější možné 1 mikrosekunda, nebylo možné prokazatelně určit, zda bylo dosaženo zlepšení. V tabulce je vidět mírný pokles náročnosti, vzhledem k prvku neurčitosti v měření při HIL simulaci a použitému rozlišení, nejsou takto malé rozdíly průkazné. Měření bylo prováděno na *Komponentě Powertrain C* (viz. tabulka 5.2).

měřený interval	referenční hodnoty [μs]	po aplikování metody [μs]
řazení 8. až 11. stupeň	0.70	0.70
rychlostní stupeň 11	1.17	1.15
jízda na manuál	0.66	0.66
jízda na zpátečku	0.67	0.66

Tabulka 6.1: Přehled naměřených hodnot procesorového stráveného vykonáváním funkcionality *komponenty Powertrain C* před a po aplikování zvolené metody optimalizace.

6.5.3 Nahrazení datafield parametrů dočasnými proměnnými

Princip této metody spočíval v nahrazení parametrů, které byly uloženy v paměti *ROM*, dočasnými proměnnými, které byly uloženy již v *RAM*. Toto nahrazení se týkalo pouze těch míst v modelu, kde docházelo k vícenásobnému čtení těchto datafield parametrů.

Ověření dopadu této optimalizace na běh procesoru byl měřen na *Komponentě Gearbox D* (tabulka 5.2). V jedné z funkci použité v této části modelu, byly nahrazeny 2 parametry dočasnými proměnnými.

Následně bylo provedeno celkem dvacet pět měření referenčních měření a stejný počet měření po změně modelu. Výsledná měření byla zpracována a výsledky jsou vidět v následující tabulce 6.2.

měřený interval	referenční hodnoty [μs]	po aplikování metody [μs]
řazení 8. až 11. stupeň	73.81	72.21
rychlostní stupeň 11	73.84	71.93
jízda na manuál	75.84	74.14
jízda na zpátečku	77.35	75.26

Tabulka 6.2: Přehled naměřených hodnot procesorového stráveného vykonáváním funkcionality *Komponenty Gearbox D* před a po aplikování zvolené metody optimalizace.

V tomto případě již dochází k prokazatelnému zlepšení náročnosti běhu aplikace. Použité rozlišení měření bylo v tomto případě také 1[μs]. Simulačně bylo ověřeno, že při klidné jízdě na 11. rychlostní stupeň dochází k celkem devadesáti volání změněné funkce. Celkem se tedy jedná o sto osmdesát volání parametrů. Při rozdílu 1.91 [μs] je možné jednoduchým výpočtem určit, že v průměru každé čtení dočasné proměnné z paměti *RAM* ušetří oproti standardnímu řešení přibližně 10.5 [ns].

$$73.84 - 71.93 = 1.91[\mu s] \tag{6.1}$$

$$\frac{1.91}{180} \doteq 10.5[ns] \tag{6.2}$$

Kapitola 7

Závěr

V této diplomové práci byla popsána problematika řešení nárůstu požadavků na výkon hardwaru, na kterém je spouštěn software, který je automaticky generován z modelu v *Simulink*u, a navržena pravidla modelování pro optimalizaci výsledného kódu. Práce byla vypracována ve společnosti ZF Engineering Plzeň s.r.o. na softwaru používaném v řídicích jednotkách převodovek pro nákladní automobilovou dopravu. Jednou z hlavních motivací pro zpracování toho tématu byla možnost podílet se na vylepšování softwaru ve vozidlech, které je možné potkat na ulici každý den.

Na začátku práce je představena společnost ZF Fridrishafen AG, čím se zabývá a jejím působením v České republice.

V následující části je práce věnována popisu základních standardů programování, které se používají v automobilovém průmyslu. Je zde rozepsáno, proč jsou v tomto odvětví normy nutné a co je jejich cílem. Dále je tato kapitola věnována základnímu popisu softwarových nástrojů, které byly v práci použity.

Další část práce seznamuje čtenáře s hardwarem, který je použit v automatické převodovce. Jsou zde popsány systémové prostředky (například použitý procesor a jeho frekvence, velikost RAM a ROM paměti) a jejich současná vytíženost a způsob hlídání běhu smyčky aplikace.

Čtvrtá kapitola je věnována automatickému generování zdrojového kódu pomocí nástroje *TargetLink*. Je zde popsáno výchozí rozhraní pro generování kódu a způsob generování kódu ve firmě ZF Engennering Plzeň.

Předposlední kapitola práce je věnována způsobům měření systémových prostředků zejména pak času běhu aplikace. Bylo zapotřebí najít vhodné metody měření těchto systémových prostředků a následně musel být nalezen způsob zajištění validity měření. Závěr této části je věnován popisu metod zpracování naměřených dat tak, aby mohla být následně mezi sebou porovnána.

Poslední část práce se zabývá návrhem optimalizačních metod, jejich zrealizováním a popsáním dosažených výsledků. V této části byly navrženy celkem tři způsoby optimalizace i když ne všechny se ukázaly jako užitečné. V prvním případě byla provedena analýza možností nastavení generování kódu přímo programu *TargetLink*. Následně byla provedena série pokusů generování s různými variantami optimalizačního nastavení, žádný ale nevedl k lepšímu zdrojovému kódu a v naprosté většině případů byl *TargetLink*em vygenerován identický kód. Druhá metoda spočívala v přemodelování složených podmínek tak, aby ve výsledném vygenerovaném kódu nebyla žádná část podmínky volána víckrát, než je nezbytně nutné. Tato metoda by mohla mít optimalizační potenciál při použití na větší část modelu. V tomto případě byly změny použity v příliš malém rozsahu a zlepšení nebylo možné prokázat. Princip poslední metody spočíval v nahrazení na jednom místě vícenásobně čtených parametrů, které jsou uloženy v *ROM* paměti, dočasnými proměnnými uloženými v *RAM*. U této metody bylo proka-

zatelně naměřeno zlepšení a byla zařazena do standardních postupů při další práci na modelu v běžné praxi.

V budoucnu by mohla být práce rozšířena o podrobnou analýzu rozdílu v optimalizaci ručně psaného kódu a kódu generovaného programem *TargetLink*. Dále by mohla být provedena analýza na téma rozdíly v generování kódu pomocí nástroje *dSPACE TargetLink* a *MathWorks Embedded Coder*, rozdíly v požadavcích vygenerovaného kódu na systémové prostředky řídicí jednotky a také na provázanost nástrojů s normami programování pro automobilový průmysl.

Literatura

[1] HEROUT, Pavel. *Učebnice jazyka C 1. díl.* České Budějovice: KOOP, 2008. ISBN 978-80-7232-383-8. [cit. 2018-12-16].

[2] DUFOUR, Emmanuel. *MISRA C++:2008 Guidelines for the use of the C++ language in critical systems.* Warwickshire, UK: Hobbs the Printers, 2008. ISBN 978-1-906400-04-0.

[3] COLGREN, Richard. *Basic MATLAB, Simulink and Stateflow.* Lawrence, Kansas: American Institute of Aeronautic and Astronautics, 2007. ISBN 978-1-56347-838-3.

[4] *ZF Friedrichshafen AG* [online]. Friedrichshafen: ZF Friedrichshafen, 2018 [cit. 2019-04-27]. Dostupné z: https://www.zf.com

[5] *MISRA* [online]. Nuneaton: HORIBA MIRA, 2019 [cit. 2019-04-12]. Dostupné z: www.misra.org.uk

[6] *AUTOSAR* [online]. Munich: AUTOSAR, 2019 [cit. 2019-04-15]. Dostupné z: https://www.autosar.org/

[7] ISO 26262 - bezpečnosti vozidel. *DNV GL* [online]. Praha: DNV GL, 2018 [cit. 2019-05-27]. Dostupné z: https://www.dnvgl.cz/assurance/automotive/iso-26262-bezpecnosti-vozidel.html

[8] Model Based Design. *Humusoft* [online]. Praha: Humusoft, 2019 [cit. 2019-03-18]. Dostupné z: https://www.humusoft.cz/matlab/mbd/

[9] *dSPACE* [online]. Paderborn: dSPACE, 2019 [cit. 2019-05-27]. Dostupné z: https://www.dspace.com/en/pub/home.cfm

[10] *MathWorks* [online]. Natick: MathWorks, 2019 [cit. 2019-05-01]. Dostupné z: https://www.mathworks.com/

[11] *Vector Informatik* [online]. Stuttgart: Vector Group, 2019 [cit. 2019-05-27]. Dostupné z: https://www.vector.com/

[12] *MicroNova - Software and System* [online]. Vierkirchen: MicroNova, 2019 [cit. 2019-04-17]. Dostupné z: https://www.micronova.de/

[13] Simulace PIL. *REX Controls* [online]. Plzeň: REX Controls, 2019 [cit. 2019-04-27]. Dostupné z: https://www.rexcontrols.cz/simulace-pil