Master thesis

Czech
Technical
University
in Prague

Faculty of Electrical Engineering
Department of Control Engineering

Converter for Simulation Scenarios of Automotive Control Units

Leoš Mikulka

May 2015

Thesis supervisor: Ing. Michal Sojka, Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Leoš Mikulka**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Nástroj pro převod simulačních scénářů automobilových řídicích jednotek**

Pokyny pro vypracování:

1. Seznamte se s nástroji CANoe a Provetech:TA a jejich využitím pro testování automobilových řídicích jednotek.
2. Zprovozněte řídicí jednotku autoradia a její připojení ke sběrnici CAN. Pomocí výše zmíněných nástrojů realizujte tzv. rest-bus simulaci pro otestování správné funkce řídicí jednotky.
3. Navrhněte a implementujte nástroj pro automatický převod simulačních skriptů z nástroje CANoe do jiných formátů (minimálně Provetech:TA a C++ či Python). Umožněte načítání informací o signálech na sběrnici ze souborů ve formátu DBC a ARXML.
4. Nástroj implementujte modulárním způsobem, aby bylo možné doplňovat další funkce a uvolněte ho pod open-source licencí.
5. Vše důkladně otestujte a zdokumentujte.

Seznam odborné literatury:

Vector: CANoe User Manual,
http://www.vector.com/portal/medien/cmc/manuals/CANoe75_Manual_EN.pdf
AUTOSAR TPS ECUConfiguration, http://www.autosar.org/fileadmin/files/releases/4-2/methodology-templates/templates/standard/AUTOSAR_TPS_ECUConfiguration.pdf

Vedoucí: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

[redacted]

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta elektrotechnická -4- L.S.

[redacted]

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 30. 1. 2015

Declaration

I hereby declare that I worked out this thesis independently and that I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

7/5/2015

In Prague on

[illegible]

Signature

Acknowledgment

Let me thank my supervisor Ing. Michal Sojka, Ph.D. for his professional guidance and helpful recommendations during the course of this work.

Furthermore, I would like to thank Ing. Milan Mráz for his willingness and useful advice, primarily from the technical point of view.

Besides, I would like to send my last thanks to all other around me who contributed with even a little piece of advice and supported me through all the time.

Abstract

The main goal of this diploma thesis was the development of a converter for simulation scenarios of electronic control units. The thesis mainly deals with the simulation of a type rest-bus. The overview of available software tools for testing and development of automotive control units is provided. The rest-bus simulation representing the control of a head unit has been developed for two specific tools – Vector CANoe by the Vector company and PROVEtech:TA by the MBtech Group company. Consequently, the converter ensuring mainly the conversion of testing scripts and basic graphical user interface from CANoe to PROVEtech:TA has been developed. Testing scripts can be converted from CAPL language to WinWrap Basic and C language. The whole conversion process as well as general usage of the developed tool are described in this document. The thesis was developed in a cooperation with company MBtech Group. The conversion of simulation scenarios has been successfully tested on developed rest-bus simulations for the head unit, which was provided by MBtech. The testing process showed functional behavior of translating various simulation scripts from the CAPL language to WinWrap Basic and C. The modularity of the tool enables adding conversions to different tools.

Keywords

rest-bus simulation; electronic control units; converter; open-source

Abstrakt

Hlavním cílem této diplomové práce byl vývoj převodníku simulačních scénářů automobilových řídicích jednotek. Během této práce je převážně pracováno se simulací typu rest-bus. V práci je uveden přehled softwarových nástrojů pro testování a vývoj řídicích jednotek. Rest-bus simulace představující ovládání palubního počítače byla vytvořena pro dva specifické nástroje – Vector CANoe od společnosti Vector a PROVEtech:TA od společnosti MBtech Group. Následně byl vyvinut nástroj zajišťující převážně převod testovacích skriptů a základního grafického uživatelského prostředí z CANoe do PROVEtech:TA. Testovací skripty mohou být převáděny z jazyku CAPL do jazyku WinWrap Basic a C. Tato práce byla vyvíjena ve spolupráci se společností MBtech Group. Převod simulačních scénářů z CANoe do PROVEtech:TA byl úspěšně otestován na vyvinutých rest-bus simulacích pro řídicí jednotku palubního počítače, která byla poskytnuta firmou MBtech. Testování prokázalo funkčnost převodu různých simulačních skriptů z jazyka CAPL do WinWrap Basic a C. Modularita nástroje umožňuje přidání převodu do dalších nástrojů.

Klíčová slova

rest-bus simulace; řídicí jednotky; převodník; open-source

Contents

List of Figures

List of Tables

Abbreviations

AUTOSAR	Automotive Open System Architecture
ARXML	AUTOSAR XML
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CAN	Controller Area Network
CAPL	CAN Access Programming Language
DBC	Database Container
ECU	Electronic Control Unit
FIBEX	Field Bus Exchange Format
GUI	Graphical User Interface
HiL	Hardware-in-the-Loop
I-PDU	Interaction Layer Protocol Data Unit
IL	Interaction Layer
LIN	Local Interconnect Network
LDF	LIN Description File
LR	Left-to-Right, rightmost derivation
LALR	Look-Ahead LR
OEM	Original Equipment Manufacturer
PDU	Protocol Data Unit
PLY	Python Lex-Yacc
RBS	Rest-bus Simulation
RTAE	Real Time Automation Engine
TA	Test Automation
WWB	WinWrap Basic
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformation

Chapter 1 Introduction

Electronic in today's vehicles is becoming more and more complex as their overall technological capacity grows. With this increased vehicle capacity comes a significant growth in the number of electronic control units (ECUs) inside a vehicle. Due to this increased complexity, each of individual ECUs interacts with each other and depends on encompassing information, which is transferred through in-vehicle buses. Since safety is an important factor, testing and validation of the right functionality of an ECU plays a critical role. However, often not all ECUs of a network are available during testing. The remaining bus simulation (rest-bus simulation) is an ideal solution when the missing ECUs (i.e. the "rest") of the bus need to be simulated. This allows to simulate missing functionality of any control unit of an in-vehicle bus, mostly CAN, LIN or FlexRay. Moreover, it can greatly reduce testing time and costs.

A wide range of companies offers software tools that support rest-bus simulation. Some of these tools are provided as a part of a complete software for the development and testing of control units. During this thesis, CANoe by Vector company and PROVEtech:TA by MBtech company were used. Many costumers of MBtech company use PROVEtech:TA software but very frequently are supplied with rest-bus simulation created by an original equipment manufacturer (OEM) in CANoe software. This leads to the need of a tool that would be able to convert created rest-bus simulation from CANoe to PROVEtech:TA.

The main focus of this thesis is the creation of the converter from CANoe to PROVEtech:TA. In order to achieve that, a developer needs to get familiar with the steps needed for creating a rest-bus simulation for ECUs in both above mentioned tools. For this purpose, the rest-bus simulation for the ECU of the head unit was created manually for both tools as the first step of this work. These simulations represent the communication of the head unit with control buttons. The next step was to create the converter from CANoe to PROVEtech:TA. This tool was developed in Python. Furthermore, the emphasis was put on the modular development of the tool. This means, additional features might be easily added. The tool runs on Windows and Linux platforms. The relevant code is placed on GitHub (`https://github.com/mikulleo/RestbusSim-Converter`) and is released under the GPL license.

Furthermore, it is important to mention that an automatically generated simulation of signals and messages on a bus based on input description files[1] is generally the main part of any rest-bus simulation. During this thesis, the simulation ensuring the transmission of signals, leading to the functional behavior of the head unit was developed. The rest-bus simulation was developed only for the CAN bus according to requirements by the MBtech company. Nevertheless, despite many differences between bus types, working with their rest-bus simulations is similar in both tools, CANoe and PROVEtech:TA.

Automated test scripts can be developed to simulate behavior of the ECU by various operations with signals and messages. The scripts in Vector CANoe are written in the CAPL language, and in PROVEtech:TA they are written in the WinWrap Basic language.

The thesis is structured as follows: Chapter 2 provides an overview of available software tools on the market, as well as some open-source software tools that might be helpful during the development of rest-bus simulations. An introduction to CANoe and PROVEtech:TA is provided as well. The term rest-bus simulation as well as the configuration and overall setup of rest-bus simulation for PROVEtech:TA and CANoe, together with formats of their description files and test scripting languages, are described in Chapter 3. Chapter 4 describes the development of the converter from CANoe to PROVEtech:TA, which is the main result of this work. The examples of built abstract syntax tree as well as converted code are given in Section 4.5. A conclusion is given in Chapter 5.

[1]description file: CAN – DBC, ARXML (described in Sections 3.2.1 and 3.2.2)

Chapter 2

Software Tools

This chapter describes several tools for the development of control units that could be relevant to this work. Most software tools for control and test automation of ECUs are under a commercial license. Regarding commercial software, only PROVEtech:TA and Vector CANoe were available during this work. These two tools are described in more detail in Sections 2.3 and 2.4. On the other hand, open-source software contains in most cases tools for analysis and modification of description files that include signals and messages belonging to a particular ECU. This might be helpful during the development of an ECU. Commercial and open-source software tools are presented in Section 2.1 and Section 2.2.

2.1 Commercial Software

There are numerous commercial software tools for control and test automation of ECUs on the market. First of all, Vector CANoe, and PROVEtech:TA by MBtech company, both used in this project, are frequently used tools. Among others, TestStand by National Instruments company is often used together with the VeriStand tool mostly for Hardware-in-the-Loop (HiL) testing [1]. A further example are Automation Desk by dSpace[2][3] or samDia by Samtec company[4]. Next, QTronic company and their tools TestWeaver and Silver are mentioned because they focus on the control and automated validation of simulated systems and virtual ECUs [3]. Separate tools for the rest-bus simulation are, e.g. tresos Busmirror by Elektrobit[5], or FlexConfig RBS developed by Eberspächer company for the rest-bus simulation on FlexRay[6].

Automated test scripts simulating behavior of an ECU are often part of rest-bus simulations. Regarding these test scripts, it is interesting to mention the tool MaTeLo

[1] `http://www.ni.com/teststand/`

[2] Control Desk must be obtained in order to connect to a real bus

[3] `https://www.dspace.com/en/pub/home/products/sw/test_automation_software/automdesk.cfm`

[4] `http://www.samtec.de/en/hauptmenu/products/software/samdia/`

[5] `https://automotive.elektrobit.com/products/ecu/eb-tresos/busmirror/`

[6] `http://www.eberspaecher-electronics.com/en/products/flexconfig-rbsgateway.html`

(Markov Test Logic) by ALL4TEC company. MaTeLo is so-called Model-based testing tool. Model-based testing is a black-box approach, i.e. it is a method that examines the application functionality without peering into its internal structures or tasks [4]. It allows export of a created test script based on a created model to different languages as CAPL, Visual Basic, C, C# or Python. Also, it allows the export to formats XML or PDF by using XSLT[7]. This software is supported by many test tools. Among others, CANoe, PROVEtech:TA or TestStand – VeriStand. Using MaTeLo, it is possible to generate scripts in many languages but it does not support any conversion between those languages.

From the perspective of the rest-bus simulation, during Electric Taxi EVA project under TUM CREATE program, the tool CANTool was developed [5]. It allows the export of rest-bus simulation files. Even though, this tool is not commercial, it is not freely available for download.

2.2 Open-source Software

Available open-source software tools can be in most cases used for analysis or setup of configuration files used for the rest-bus simulation. Rarely, they allow the conversion between different formats of configuration files. More about specific configuration files' formats can be found in Section 3.1. The configuration files contain defined signals and their parameters. Most of the open-source tools are collectively called "viewers". Example of some the tools is shown in Table 2.1.

LDF and Fibex Viewer are simple tools offered by Intrepid Control Systems for personal use [89]. Next, FIBEXplorer is the well-arranged tool for Fibex files analysis[10]. There are many more tools for DBC files analysis. The first of these, Canmatrix Convert is able to export DBC files to Excel [11]. This function has been verified. Above that, the feature of converting ARMXL files to DBC files or XLS files is mentioned. This functionality has not been working reliably though. This is mainly because there are several AUTOSAR standard versions of ARXML files which differ from each other. Other tool is cantools which allows extracting information from DBC files under Linux platform [12]. The software Busmaster can be used as the demonstration of the conversion from CAPL script language into C++. However, the generated C++ code is specifically designated only for usage in Busmaster tool [13].

[7]Extensible Stylesheet Language Transformation
[8]http://www.intrepidcs.com/support/ldftool.htm
[9]http://www.intrepidcs.com/support/fibexviewer.htm
[10]http://sourceforge.net/projects/fibexplorer/
[11]https://github.com/ebroecker/canmatrix
[12]http://sourceforge.net/projects/cantools/
[13]http://rbei-etas.github.io/busmaster/

Name	Function	open-source
LDF Viewer	viewer	YES (personal use)
Fibex Viewer	viewer	YES
Cannmatrix Conver	conversion DBC $\rightarrow$ Excel	YES
cantools	viewer	YES
BUSMASTER	viewer; conversion CAPL $\rightarrow$ C++	YES

Table 2.1: Summary of available open-source tools

2.3 PROVEtech:TA

PROVEtech:TA is a software tool developed by MBtech Group for control and automation of test systems. It allows a user to interactively set and measure all relevant state variables of a test system. Thus, the user can set signals and transmit messages for different test scenarios. For the optimal testing, it offers a user-friendly test manager for administration of test scripts and test results. The test manager is further described in Section 2.3.2. During a measurement, the values of one or multiple signals can be acquired and stored for further evaluation. PROVEtech:TA can also be used for implementation of real-time test scripts which run directly on a connected real-time hardware. PROVEtech:TA supports systems description based on formats DBC, LDF (LIN bus), Fibex (FlexRay bus) and ARXML. Other important part of PROVEtech:TA is the workspace. A user can add different controls such as switch controls, sliders, etc. The workspace exchanges signals with the simulation model which usually runs on a real real-time computer and provides the ECU environment. The configuration of the workspace is described in Section 3.3.3. Moreover, the diagnostic module with the user interface is included in the software. It allows access to all diagnostic services, that are provided by the ECU under test. PROVEtech:TA also includes the fault simulation model that enables injecting of electrical faults in order to test ECU reaction in hardware and software. The test language for writing scripts in PROVEtech:TA is WinWrap Basic. It is described in Section 3.5.1.

The PROVEtech tool suite is equipped with PROVEtech:RE as well. PROVEtech:RE is a test platform designed for PCs running under Microsoft Windows or INtime operating system. PROVEtech:RE is described in Section 2.3.5.

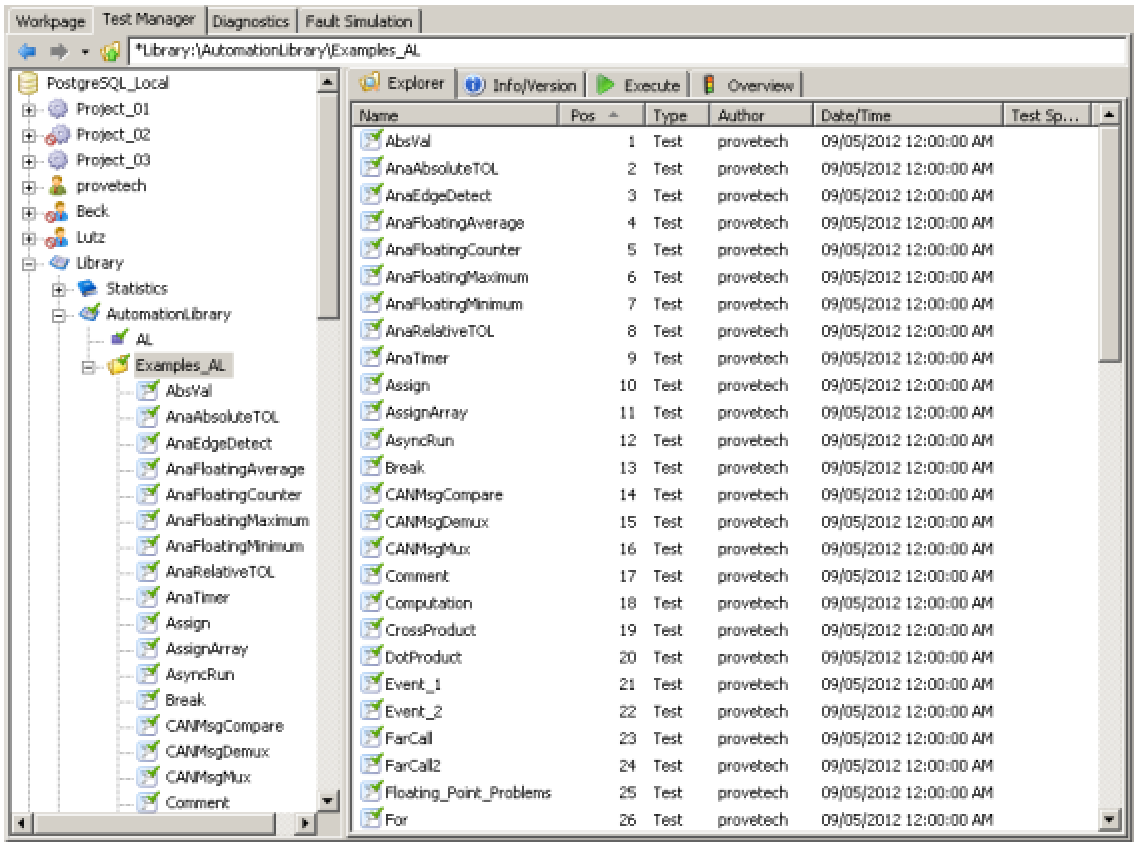
Moreover, PROVEtech:TA includes the Automation Library which provides fundamental methods for accessing a model computer which is running the Real-Time Automation Engine (RTAE), such as PROVEtech:RE or a real-time computer. [6]

2.3.1 Workspace

The workspace of PROVEtech:TA is divided into two parts, the workpage and the cockpit. They offer the same features, though the cockpit always stays visible. The workpage offers a wide range of display elements for the simulation and signal visualization such as sliding displays or switch controls. A user can place several workpages to the workspace. The workpage exchanges signals with a simulation model, which usually runs on a real-time computer or real-time automation engine such as PROVEtech:RE, and represents the ECU environment. Measurements, which protocol all signal changes, are configured and evaluated on the workpage. The workspace created during this project can be found in Fig. 3.4.

2.3.2 Test Manager

The test manager includes the environment for programming test scripts, as well as the management of existing and executed test cases. Test scripts can be executed separately or in groups and they are stored together with the test results and reports. If required, tests can run on a real-time computer with a real time automation engine (RTAE). The test manager supports Oracle, Microsoft SQL server and PostgreSQL database systems for storage of test scripts and results. Several test systems can be operated independently from each other, but nevertheless on the same database. An example of the test manager window is shown in Fig. 2.1.



PostgreSQL_Local
- Project_01
- Project_02
- Project_03
- provetech
- Beck
- Lutz
- Library
  - Statistics
  - AutomationLibrary
    - AL
    - Examples_AL
      - AbsVal
      - AnaAbsoluteTOL
      - AnaEdgeDetect
      - AnaFloatingAverage
      - AnaFloatingCounter
      - AnaFloatingMaximum
      - AnaFloatingMinimum
      - AnaRelativeTOL
      - AnaTimer
      - Assign
      - AssignArray
      - AsyncRun
      - Break
      - CANMsgCompare
      - CANMsgDemux
      - CANMsgMux
      - Comment

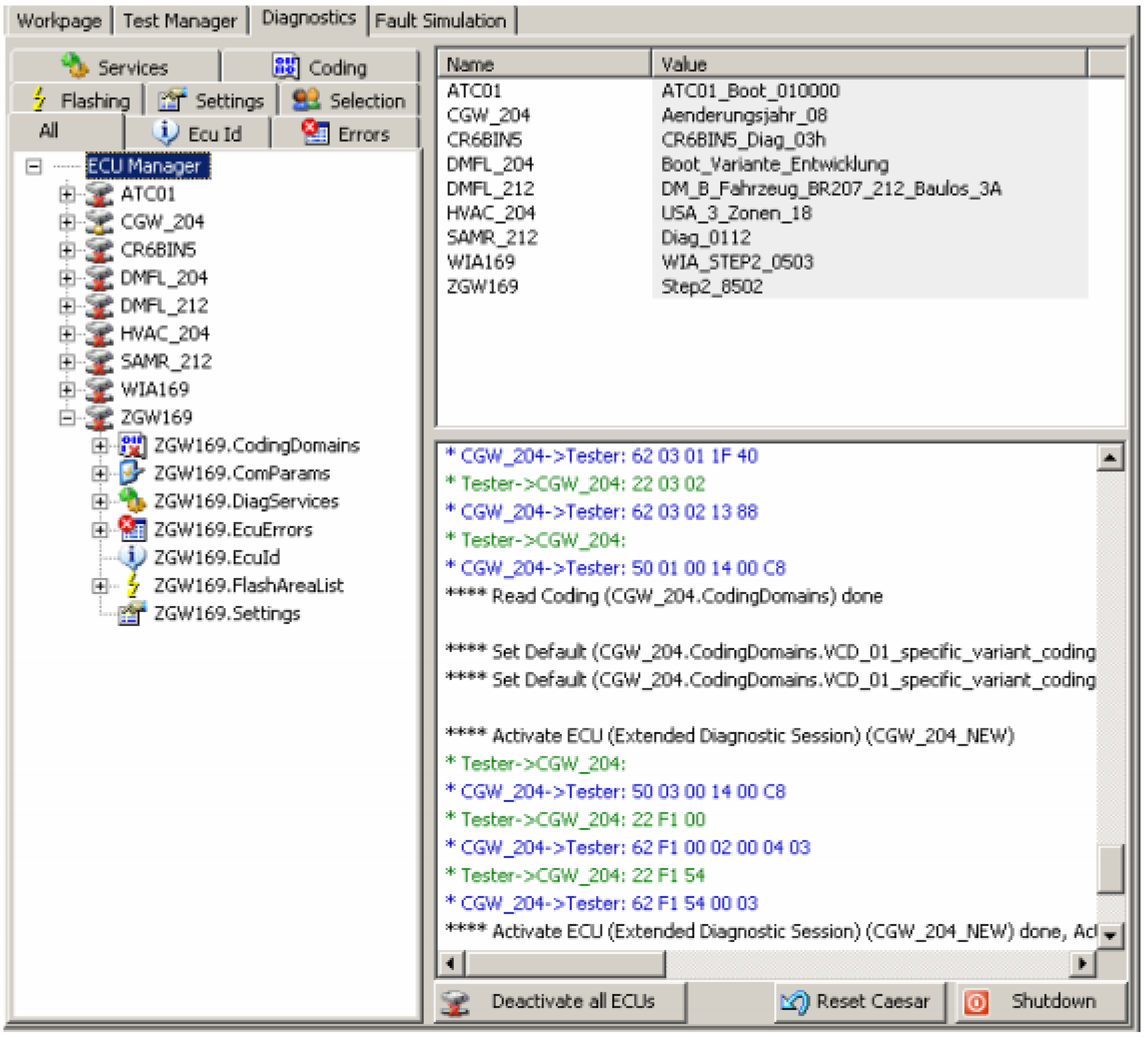
Name	Pos	Type	Author	Date/Time	Test Sp...
AbsVal	1	Test	provetech	09/05/2012 12:00:00 AM	
AnaAbsoluteTOL	2	Test	provetech	09/05/2012 12:00:00 AM	
AnaEdgeDetect	3	Test	provetech	09/05/2012 12:00:00 AM	
AnaFloatingAverage	4	Test	provetech	09/05/2012 12:00:00 AM	
AnaFloatingCounter	5	Test	provetech	09/05/2012 12:00:00 AM	
AnaFloatingMaximum	6	Test	provetech	09/05/2012 12:00:00 AM	
AnaFloatingMinimum	7	Test	provetech	09/05/2012 12:00:00 AM	
AnaRelativeTOL	8	Test	provetech	09/05/2012 12:00:00 AM	
AnaTimer	9	Test	provetech	09/05/2012 12:00:00 AM	
Assign	10	Test	provetech	09/05/2012 12:00:00 AM	
AssignArray	11	Test	provetech	09/05/2012 12:00:00 AM	
AsyncRun	12	Test	provetech	09/05/2012 12:00:00 AM	
Break	13	Test	provetech	09/05/2012 12:00:00 AM	
CANMsgCompare	14	Test	provetech	09/05/2012 12:00:00 AM	
CANMsgDemux	15	Test	provetech	09/05/2012 12:00:00 AM	
CANMsgMux	16	Test	provetech	09/05/2012 12:00:00 AM	
Comment	17	Test	provetech	09/05/2012 12:00:00 AM	
Computation	18	Test	provetech	09/05/2012 12:00:00 AM	
CrossProduct	19	Test	provetech	09/05/2012 12:00:00 AM	
DotProduct	20	Test	provetech	09/05/2012 12:00:00 AM	
Event_1	21	Test	provetech	09/05/2012 12:00:00 AM	
Event_2	22	Test	provetech	09/05/2012 12:00:00 AM	
FarCall	23	Test	provetech	09/05/2012 12:00:00 AM	
FarCall2	24	Test	provetech	09/05/2012 12:00:00 AM	
Floating_Point_Problems	25	Test	provetech	09/05/2012 12:00:00 AM	
For	26	Test	provetech	09/05/2012 12:00:00 AM	

Figure 2.1: An example of test manager [1]

2.3.3 Diagnostics

The diagnostics module allows the access to all diagnostics services which are provided by an ECU under test. Furthermore, the error storage can be accessed and internal

ECU data can be displayed. The other diagnostic functions like coding and flashing of ECUs are also supported. All diagnostic functions can be accessed via the graphical user interface as well as test scripts. The communication with the ECU is handled by using a dedicated diagnostic hardware. An example of the diagnostics window is shown in Fig. 2.2.



```
ECU Manager
  ATC01
  CGW_204
  CR6BIN5
  DMFL_204
  DMFL_212
  HVAC_204
  SAMR_212
  WIA169
  ZGW169
    ZGW169.CodingDomains
    ZGW169.ComParams
    ZGW169.DiagServices
    ZGW169.EcuErrors
    ZGW169.EcuId
    ZGW169.FlashAreaList
    ZGW169.Settings
```

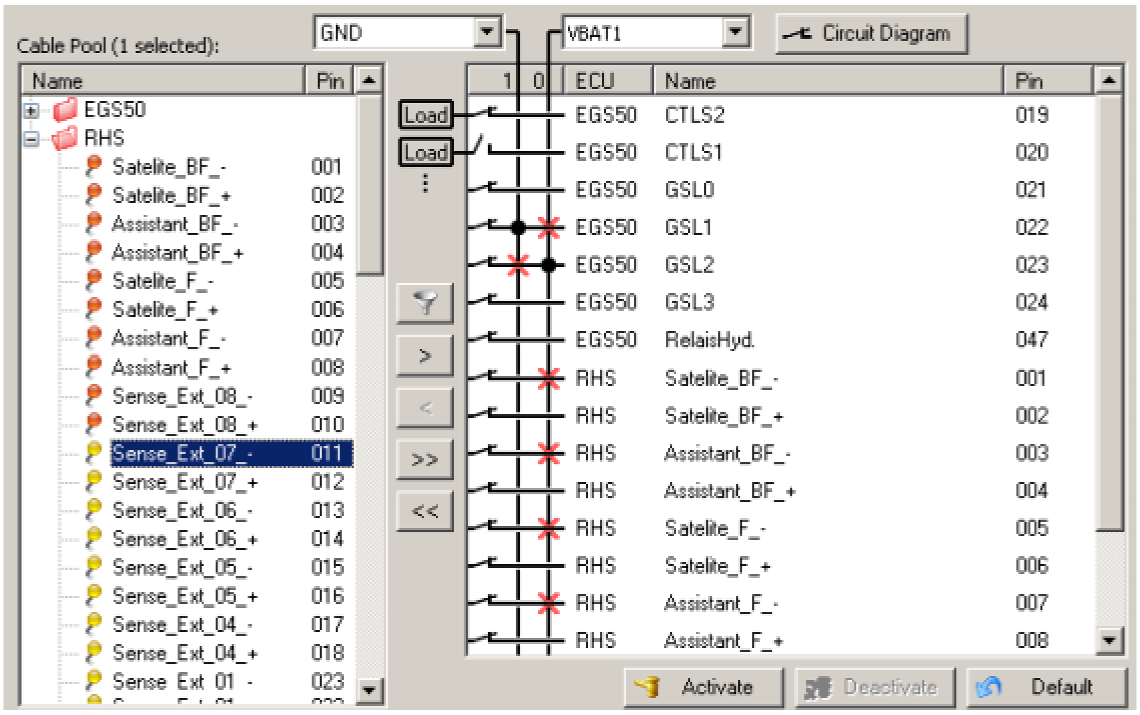
Name	Value
ATC01	ATC01_Boot_010000
CGW_204	Aenderungsjahr_08
CR6BIN5	CR6BIN5_Diag_03h
DMFL_204	Boot_Variante_Entwicklung
DMFL_212	DM_B_Fahrzeug_BR207_212_Baulos_3A
HVAC_204	USA_3_Zonen_18
SAMR_212	Diag_0112
WIA169	WIA_STEP2_0503
ZGW169	Step2_8502

```
* CGW_204->Tester: 62 03 01 1F 40
* Tester->CGW_204: 22 03 02
* CGW_204->Tester: 62 03 02 13 88
* Tester->CGW_204:
* CGW_204->Tester: 50 01 00 14 00 C8
**** Read Coding (CGW_204.CodingDomains) done

**** Set Default (CGW_204.CodingDomains.VCD_01_specific_variant_coding
**** Set Default (CGW_204.CodingDomains.VCD_01_specific_variant_coding

**** Activate ECU (Extended Diagnostic Session) (CGW_204_NEW)
* Tester->CGW_204:
* CGW_204->Tester: 50 03 00 14 00 C8
* Tester->CGW_204: 22 F1 00
* CGW_204->Tester: 62 F1 00 02 00 04 03
* Tester->CGW_204: 22 F1 54
* CGW_204->Tester: 62 F1 54 00 03
**** Activate ECU (Extended Diagnostic Session) (CGW_204_NEW) done, Ac
```

Figure 2.2: An example of diagnostics dialog [1]

2.3.4 Fault Simulation

With the fault simulation module, electrical faults can be injected in order to test the respective ECU reaction in hard- and software. By means of this fault simulation, components and systems can be tested related to their robustness and reliability. An external failure insertion unit (relay box) is required. PROVEtech:TA controls the failure simulation by setting the corresponding relays in the connected failure insertion unit. This effects the simulation of electrical faults, e.g. wiring interruptions, short cuts and pin-to-pin failures. Afterwards, it can be checked whether the fault was recorded in the diagnostic memory. Additionally, PROVEtech:TA also contains visual panels to control the failure simulation for both, ECU pin failures and CAN line failures. An example of the fault simulation is shown in Fig. 2.3.



Name	Pin
EGS50	
RHS	
Satelite_BF_-	001
Satelite_BF_+	002
Assistant_BF_-	003
Assistant_BF_+	004
Satelite_F_-	005
Satelite_F_+	006
Assistant_F_-	007
Assistant_F_+	008
Sense_Ext_08_-	009
Sense_Ext_08_+	010
Sense_Ext_07_-	011
Sense_Ext_07_+	012
Sense_Ext_06_-	013
Sense_Ext_06_+	014
Sense_Ext_05_-	015
Sense_Ext_05_+	016
Sense_Ext_04_-	017
Sense_Ext_04_+	018
Sense_Ext_01_-	023

ECU	Name	Pin
EGS50	CTLS2	019
EGS50	CTLS1	020
EGS50	GSL0	021
EGS50	GSL1	022
EGS50	GSL2	023
EGS50	GSL3	024
EGS50	RelaisHyd.	047
RHS	Satelite_BF_-	001
RHS	Satelite_BF_+	002
RHS	Assistant_BF_-	003
RHS	Assistant_BF_+	004
RHS	Satelite_F_-	005
RHS	Satelite_F_+	006
RHS	Assistant_F_-	007
RHS	Assistant_F_+	008

Figure 2.3: An example of page for wiring faults [1]

2.3.5 PROVEtech:RE

PROVEtech:RE is runtime environment which enables PROVEtech:TA to access many types of hardware devices, for example Vector CANcase used during this project. It also contains a simulation engine which can be used for rest-bus simulation. It is a Windows-based program. It is basically an ”invisible” program running in the background without its own user interface[14]. Furthermore, it upports standard automotive interfaces and bus systems such as CAN, LIN, FlexRay, etc. The most important feature for this project includes the rest-bus simulation creation. The configuration of rest-bus simulation is described in Section 3.3. However, additional features comprise generation of signal waveforms or behavioral models, or Simulink model integration. [6]

2.4 Vector CANoe

Vector CANoe is the comprehensive software tool developed by Vector Group for development, test and analysis of ECU networks. At the beginning of the development

[14]PROVEtech:RE does not need any user interface because it is controlled through PROVEtech:TA

process, CANoe can be used to create simulation models which simulate the behavior of an ECU. Further these models serve as the basis for analysis and testing of whole systems. Similarly as in PROVEtech:TA, a user can interactively adjust signals and messages for different test scenarios. CANoe contains several windows for the analysis. These windows are described in Section 2.4.2. The workspace of CANoe can consist of the windows mentioned. Furthermore, the workspace usually consists of panels. The panels are graphical control elements. They are described in Section 2.4.1. CANoe also contains a diagnostic feature set for diagnostic communications with the ECU. Various bus systems are supported. Among them are CAN, LIN, FlexRay, or Ethernet. Above that, CAN-based protocols are e.g. J1939, CANopen, or CANaerospace. Among supported system descriptions are DBC, LDF, Fibex or ARXML formats. The test language for writing scripts in Vector CANoe is CAPL. It is described in Section 4.4.3.

2.4.1 Panels

The workspace of Vector CANoe may consist of panels. The panels are graphical control elements such as standard buttons, check boxes, radio buttons, up to LCD controls, meters or even media players. These elements can be used to modify signal or variable values. The configuration of the panels is carried out in the panel designer, and is described in Section 3.4.2. The panels can then be opened in the workspace as separate windows.

2.4.2 Analysis Windows

CANoe provides a user with several windows and blocks that helps to easily analyze the network during all development phases. Most important windows are described below. The description is based on [2].

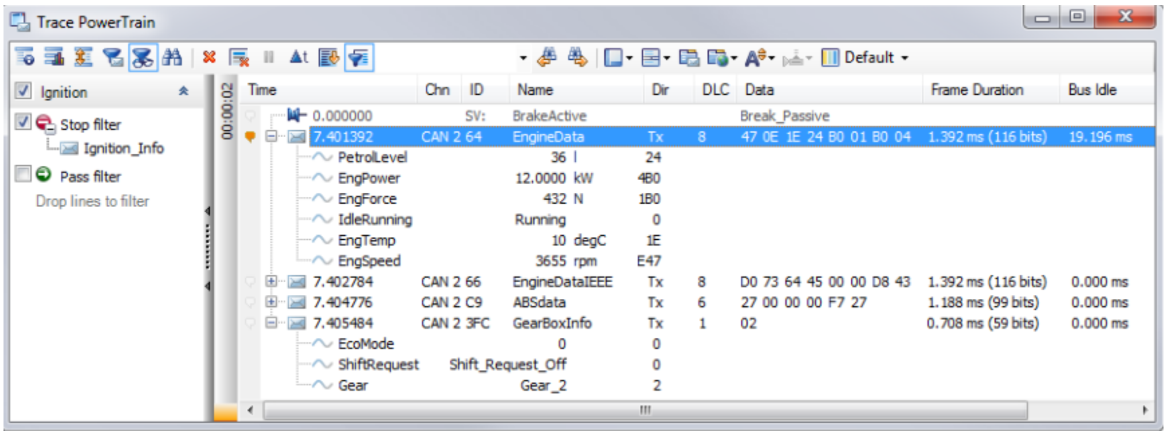
2.4.2.1 Trace Window

Bus activities, such as transmission of messages or error frames, are listed in the trace window. Moreover, individual signal values may be displayed for each message. For the data analysis, the following functions are available:

- *Insert filters* – used to reduce the amount of data displayed, or even delete some data from the data stream
- *Hide unchanged data*
- *Color events* – highlight important data with different colors
- *Set markers* – markers are assigned to an event and its time stamp; used to quickly find events
- *Show statistics* – used to display signals and messages in details

- *Log data*

An example of the trace window is shown in Fig. 2.4.

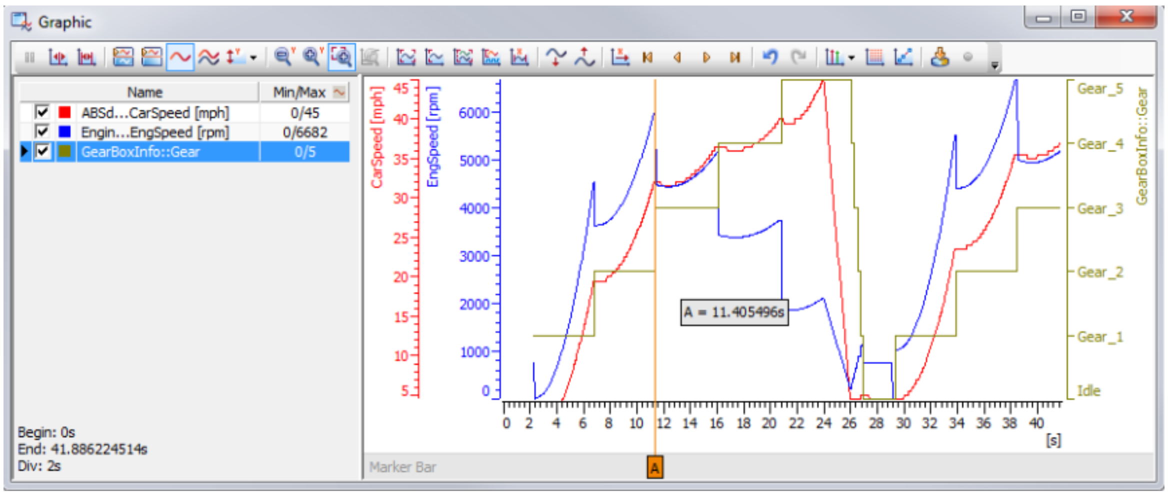


```
Trace PowerTrain
Default

Ignition            Time             Chn       ID   Name            Dir  DLC  Data                      Frame Duration     Bus Idle
Stop filter         0.000000                   SV:  BrakeActive               Break_Passive
  Ignition_Info     7.401392         CAN 2 64       EngineData      Tx   8    47 0E 1E 24 B0 01 B0 04   1.392 ms (116 bits) 19.196 ms
Pass filter           PetrolLevel                           36 l    24
Drop lines to filter  EngPower                      12.0000 kW      4B0
                      EngForce                          432 N       1B0
                      IdleRunning                   Running         0
                      EngTemp                            10 degC    1E
                      EngSpeed                         3655 rpm     E47
                    7.402784         CAN 2 66       EngineDataIEEE  Tx   8    D0 73 64 45 00 00 D8 43   1.392 ms (116 bits) 0.000 ms
                    7.404776         CAN 2 C9       ABSdata         Tx   6    27 00 00 00 F7 27         1.188 ms (99 bits)  0.000 ms
                    7.405484         CAN 2 3FC      GearBoxInfo     Tx   1    02                        0.708 ms (59 bits)  0.000 ms
                      EcoMode                             0         0
                      ShiftRequest           Shift_Request_Off      0
                      Gear                               Gear_2     2
```

Figure 2.4: An example of trace window with active filter and marker [2]

2.4.2.2 Graphics Window

Similar to the trace, the graphics windows are used to monitor bus activities. Moreover, it can be used to display environment data and diagnostic parameters as curves. The functions available in the graphics window include logging data, showing statistics or setting markers. An example of the graphics window is shown in Fig. 2.5.



Graphic

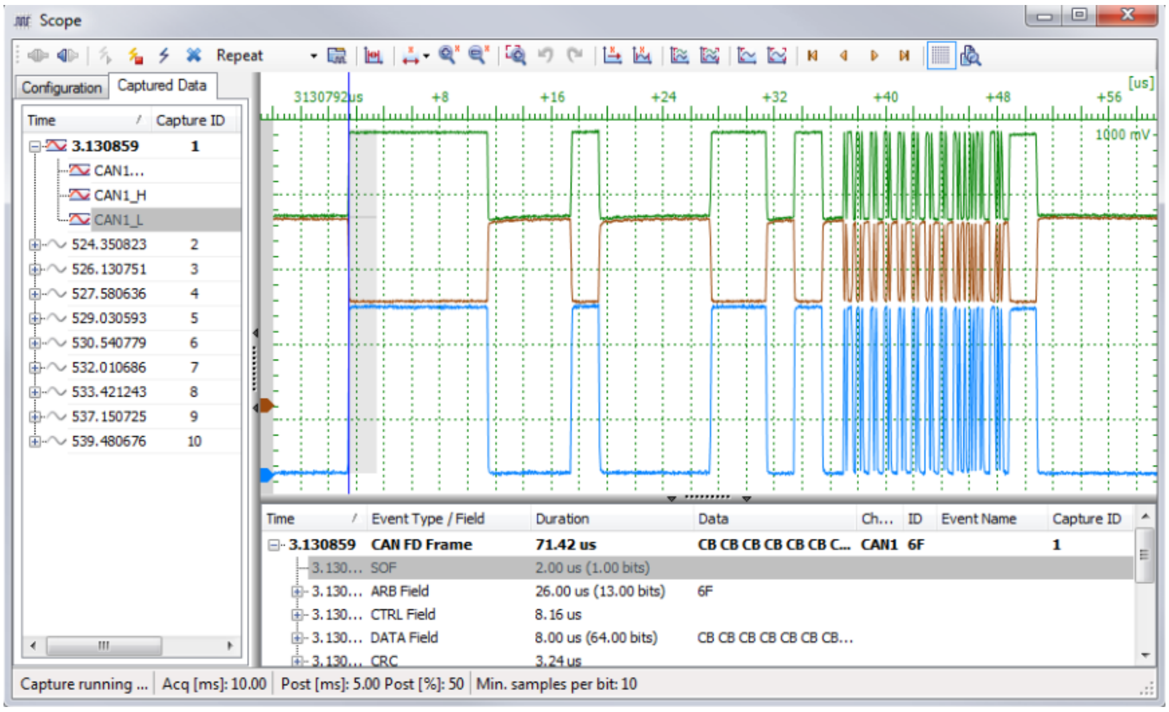
Name	Min/Max
✓ 🟥 ABSd…CarSpeed [mph]	0/45
✓ 🟦 Engin…EngSpeed [rpm]	0/6682
✓ 🟩 GearBoxInfo::Gear	0/5

Begin: 0s
End: 41.886224514s
Div: 2s

Figure 2.5: An example of graphics window with marker [2]

2.4.2.3 Scope Window

The scope window graphically depicts bus level measurements and is used for the analysis of protocol errors. In the scope windows, it is possible to set triggers manually, via CAPL or via pre-configured events. Individual trigger conditions can be combined using the logical OR function. Another function is the ability to compare signals in different approaches. The scope windows can be completely controlled from a CAPL test module. An example of the scope window is shown in Fig. 2.6.



Time	Capture ID
3.130859	1
CAN1...	
CAN1_H	
CAN1_L	
524.350823	2
526.130751	3
527.580636	4
529.030593	5
530.540779	6
532.010686	7
533.421243	8
537.150725	9
539.480676	10

Time	Event Type / Field	Duration	Data	Ch...	ID	Event Name	Capture ID
3.130859	CAN FD Frame	71.42 us	CB CB CB CB CB CB C...	CAN1	6F		1
3.130...	SOF	2.00 us (1.00 bits)					
3.130...	ARB Field	26.00 us (13.00 bits)	6F				
3.130...	CTRL Field	8.16 us					
3.130...	DATA Field	8.00 us (64.00 bits)	CB CB CB CB CB CB CB...				
3.130...	CRC	3.24 us					

Capture running ... | Acq [ms]: 10.00 | Post [ms]: 5.00 Post [%]: 50 | Min. samples per bit: 10

Figure 2.6: An example of scope window [2]

2.4.2.4 Statistics Window

The statistics window shows statistical information about bus activities during a measurement. This includes information such as bus load on a node and frame level, burst counter or counters and rates for messages and error frames. Certain bus statistics can be evaluated in the analysis windows such as the graphics window.

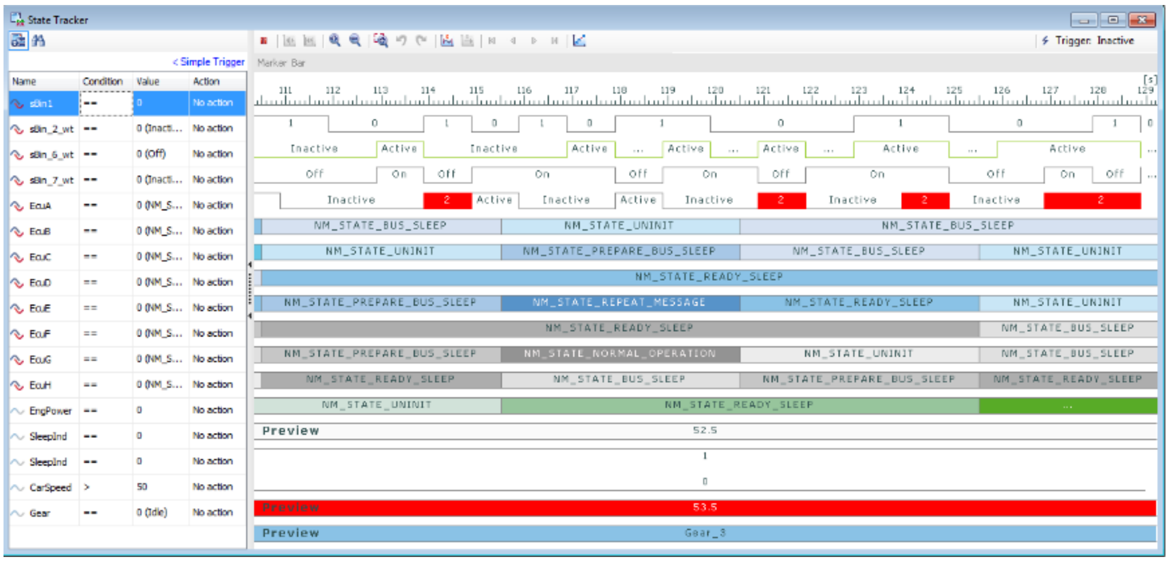
2.4.2.5 State Tracker

The state tracker can be used to analyze states, state transitions and signals, as well as to visualize time dependencies. It is especially well-suited for displaying digital inputs and outputs as well as status information such as terminals status or network management states. The functions of the state tracker are following:

- *Search for error* – errors can be searched and functions can be monitored based on analysis of the time response of states, signals and state transitions
- *Analyze information* – analysis of various information such as the states of internal ECU communications, bus signals or ECU I/Os

- *Monitor AUTOSAR runnables* – monitoring of runnable states and reading of these states via the Vector hardware

An example of the state tracker is shown in Fig. 2.7.



Name	Condition	Value	Action
sBin1	==	0	No action
sBin_2_wt	==	0 (Inacti...	No action
sBin_6_wt	==	0 (Off)	No action
sBin_7_wt	==	0 (Inacti...	No action
EcuA	==	0 (NM_S...	No action
EcuB	==	0 (NM_S...	No action
EcuC	==	0 (NM_S...	No action
EcuD	==	0 (NM_S...	No action
EcuE	==	0 (NM_S...	No action
EcuF	==	0 (NM_S...	No action
EcuG	==	0 (NM_S...	No action
EcuH	==	0 (NM_S...	No action
EngPower	==	0	No action
SleepInd	==	0	No action
SleepInd	==	0	No action
CarSpeed	>	50	No action
Gear	==	0 (Idle)	No action

Figure 2.7: An example of state tracker [2]

Chapter 3
Rest-bus Simulation

This chapter explains the term rest-bus simulation, as well as steps needed for creating of such a simulation. The whole procedure of creating a specific rest-bus simulation includes several steps. The procedure obviously varies for different software tools. For PROVEtech:TA these steps are:

- Configuration of description files
- Creating of XML file called RBS descriptor (Section 3.3.1)
- Creating of PROVEtech XML configuration file (Section 3.3.2)
- Setup of a workspace, creating of automated test scripts

For Vector CANoe these steps are:

- Configuration of description files
- Adding database containing signals and messages to a project
- Configuration of simulation setup (Section 3.4.1)
- Setup of a workspace, creating of automated test scripts

However, it is appropriate to mention that all rest-bus simulations need configuration files containing all simulated messages and signals. The format of the configuration files depends on the bus type used during a rest-bus simulation. These files are described in Section 3.2. How the rest-bus simulation should be created in PROVEtech:TA is then described in Section 3.3. Steps needed for configuration of rest-bus simulation in CANoe are described in Section 3.4. Automated scripts for the rest-bus simulation are written in the language specific for a particular software tool. The programming languages used in PROVEtech:TA and Vector CANoe, together with differences between them, are described in Section 3.5.

3.1 Definition of Rest-bus Simulation

The term rest-bus simulation comes from the phrase Remaining Bus Simulation. During the development of a vehicle network, the system integrators often face the problem that not all control units are available on a bus. Since performing operations on an incomplete network may lead to an improper behavior, the rest-bus simulation was developed. It simulates communication of any missing control units. This means, all real measurements and signals are replaced by simulated ones. Control units therefore can be tested without the need of creating a whole network. Moreover, using the rest-bus simulation can greatly reduce testing time and costs of setting up an entire network or performing expensive field tests. Fig. 3.1 shows the scheme of rest-bus simulation in this project. An example of the demo rest-bus simulation from Vector CANoe is displayed in Fig. 3.2.

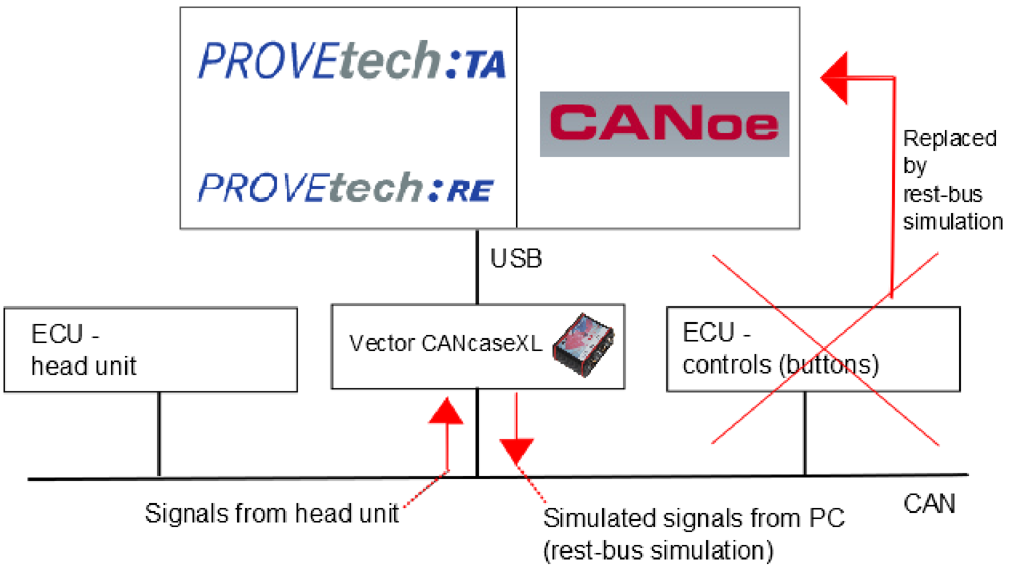
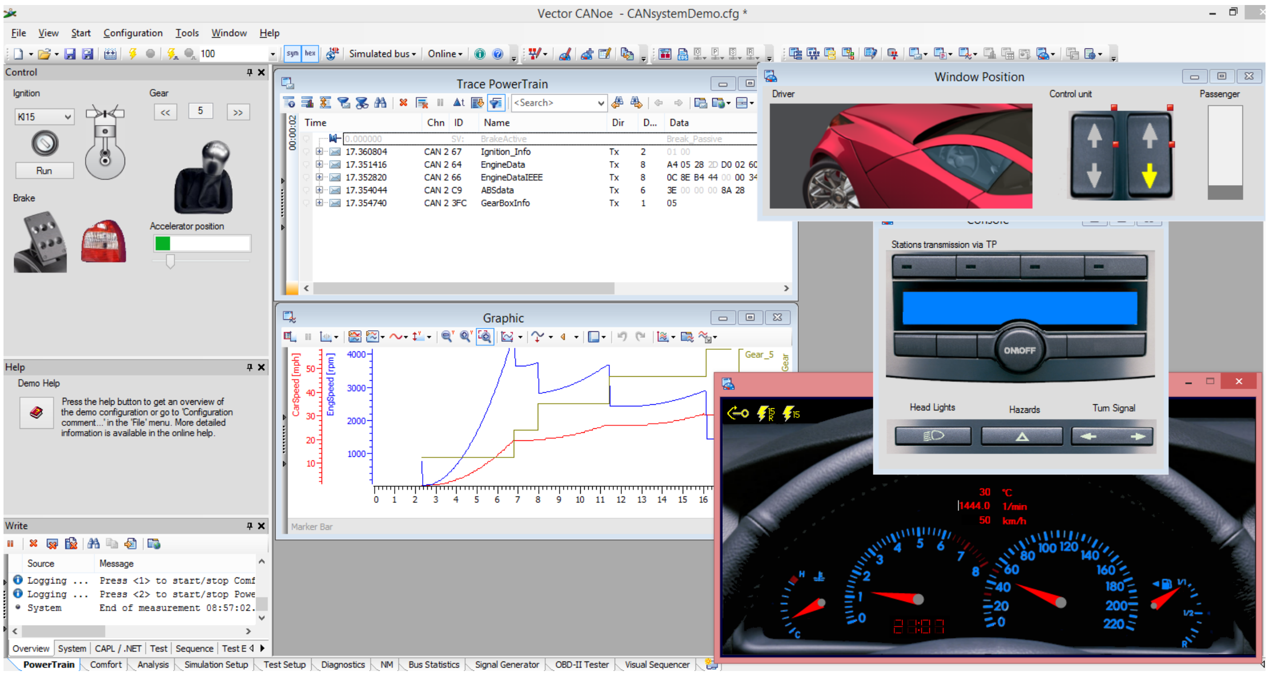


Figure 3.1: Rest-bus simulation scheme

3.2 Description Files Formats

A format of description files differs for different buses. For CAN bus, the configuration files are called DBC files. For LIN bus, they are called LDF files. Above that, ARXML files can be used instead. ARXML is an XML-based format used for the description of an ECU based on AUTOSAR standard which is generally applicable for various kinds of buses. These files will always serve as the input for a certain step in the rest-bus simulation configuration process. Therefore, the structure of DBC files is described in Section 3.2.1. The basic structure of ARXML files used during this project is described in Section 3.2.2.



```
Time          Chn    ID   Name             Dir  D...  Data
0.000000             SV:  BrakeActive                 Break_Passive
17.360804     CAN 2  67   Ignition_Info    Tx   2     01 00
17.351416     CAN 2  64   EngineData       Tx   8     A4 05 28 2D D0 02 60
17.352820     CAN 2  66   EngineDataIEEE   Tx   8     0C 8E B4 44 00 00 34
17.354044     CAN 2  C9   ABSdata          Tx   6     3E 00 00 00 8A 28
17.354740     CAN 2  3FC  GearBoxInfo      Tx   1     05
```

```
Source        Message
Logging ...   Press <1> to start/stop Comf
Logging ...   Press <2> to start/stop Powe
System        End of measurement 08:57:02.
```

Figure 3.2: Demo example of a rest-bus simulation

3.2.1 DBC File

A DBC file describes signals and messages for the communication of a CAN network. This file is sufficient to monitor and analyze the network and carry out the rest-bus simulation, i.e. to simulate nodes that are not physically available. The functional behavior of an ECU is not addressed by the DBC file. The following sections describe the overall structure of a DBC file. A DBC file consists of several sections. There are specific keywords that characterize these sections inside a DBC file. These keywords can be followed by attributes needed for accurate description of a particular DBC section. During this project, a DBC file delivered by OEM was used as the base. The information about DBC files are based on [7].

3.2.1.1 Version and Additional Symbols Specification

The DBC file begins with a header describing the version and new keywords (symbol entries) for other sections. The version is either empty or is a string used by CANdb editor. The CANdb editor is the editor by Vector for editing DBC files. The header has the following structure:

```
VERSION "version string"
NS_:
            keyword
            keyword
            .
            .
            .
```

3.2.1.2 Bit Timing

This DBC section is obsolete and is usually empty. Nonetheless, it must appear in the DBC file. Thus, only the keyword BS will be listed as follows:

```
BS_:
```

3.2.1.3 Node Definitions

All nodes participated on the bus are listed in this DBC section. The section contains only one attribute – `node_name`. The structure and an example of the implementation are as follows:

```
BU_: node_name

BU_: Controls Display TransmitGateway
```

3.2.1.4 Message Definitions

This DBC section defines the names of all messages as well as their properties and signals transferred in the messages. Signal definition, together with the exact structure and an example of the message implementation, are described in Section 3.2.1.5. The message definition has following attributes:

`message_id`

`message_name`

`message_size`: size in bytes

`transmitter`: name of a node transmitting the message; if the message has no sender or is unknown, the name is `Vector__XXX`

`signals`: signals defined according to Section 3.2.1.5

3.2.1.5 Signal Definitions

The signal definition segment includes properties of a given signal and its position in a message. The definition has the following attributes:

`signal_name`

`multiplexer_indicator`: defines whether the signal is a normal signal (empty string), a multiplexer switch (`M`) or a multiplexed by the multiplexer switch (`mX`, where X is a number of the multiplexer)

`start_bit`: the position of the first bit; for little endian byte order the position of least-significant bit is given, for big endian byte order the position of most-significant bit is given `signal_size`: number of bits

`byte_order`: `0` for little endian, `1` for big endian

`value_type`: `+` for unsigned, `-` for signed

`(factor, offset)`: defines values for the linear conversion rule to convert the signal's raw value into the signal's physical value – `physical_value = raw_value * factor + offset`

`[min, max]`: the range of valid physical values of the signal

`unit`: e.g. m/s, kg, etc.

`receiver`: name of the node receiving the signal; if the signal has no receiver or is unknown, the name is `Vector__XXX`

The message structure with defined signals and an example of the implementation are as follows:

```
BO_ message_id message_name : message_size transmitter
    SG_ signal_name multiplexer_indicator : start_bit | signal_size
        @ byte_order value_type (factor, offset) [min, max] unit
       receiver

BO_ 499 Controls_Group1 : 8 Controls
        SG_ Control_North_Pressed : 17 | 1@1+ (1,0) [0,1] ""
           TransmitGateway
        SG_ Control_South_Pressed : 21 | 1@1+ (1,0) [0,1] ""
           TransmitGateway
        SG_ RadioKey_Pressed : 51 | 1@1+ (1,0) [0,1] ""
           TransmitGateway
        SG_ RadioKey_Pressed_Ack : 52 | 1@1+ (1,0) [0,0] ""
           TransmitGateway
```

3.2.1.6 Comment Definitions

In this section, comments for nodes, messages and signals are included. An example of a comment is following:

```
CM_ BO_ 499 "Central controls operation"
CM_ SG_ 499 Control_North_Pressed "Central control elemement North
   operation"
```

3.2.1.7 User Defined Attributes Definitions

User defined attributes are meant to extend the object properties of the DBC file. These attributes must be defined by defining the keyword BA_DEF_ and they must be assigned a value entry by the keyword BA_. The section contains the following attributes:

`object_type`: i.e. node (BU_), message (BO_), signal (SG_) `attribute_name`

`attribute_value_type`: i.e. INT, HEX, FLOAT, STRING or ENUM; in case of numeric values, minimum and maximum value must be assigned

After this definition, it it possible to assign a value entry to a specific attribute in similar way as comments are assigned a text. The structure and an example of the implementation are as follows:

```
BA_DEF object_type attribute_name attribute_value_type

BA_DEF SG_ "GenSigStartValue" INT 0 10000
BA_DEF SG_ "CycleTime" INT 0 3600000
BA_ "GenSigStartValue" SG_ 499 Control_North_Pressed 0
BA_ "CycleTime" SG_ 499 Control_North_Pressed 100
```

3.2.2 ARXML File

The AUTOSAR XML files (ARXML) generally contain the whole system description. The system description consists of a network topology, i.e. bus systems (CAN, FlexRay, etc.), connected ECUs, gateways, as well as of the description of communication such as mapping signals to CAN channels, or even mapping of software components. The ARXML files are generated by special tools. Since they are very verbose, it is not convenient to edit them by the hand. They usually consist of tens of thousands lines of XML code.

The whole ARXML structure is not explained in details because the whole explanation could be as long as this whole thesis. However, the vary basic structure of the ARXML file relevant for this project is mentioned. During this project, the ARXML file has been obtained from OEM. This file originates from the same database as the DBC file mentioned in Section 3.2.1. It describes the bus topology and data, such as clusters, ECU instances, signals, and so on. The whole file is divided into several AUTOSAR packages (`AR-PACKAGE`). This allows to create top level packages to structure the contained AUTOSAR elements. All packages include the tag `SHORTNAME` that contains a unique name of elements. The description of the ARXML file is based on [8]. Some of the elements are

`UNIT`: the physical measurement unit; all units defined should be SI units

`SW-BASE-TYPE`: the meta class; represents a base type used within ECU software

`COMPU-METHOD`: the meta class; expresses the relationship between a physical value and a mathematical representation

`SYSTEM-SIGNAL`: allows to represent the communication system's view of data exchanged between SW components which reside on different ECUs in a flattened structure (with exactly one system signal defined for each data element prototype sent and received by connected SW component instances)

`ECU-INSTANCE`: used to define the ECUs used in the topology; includes these important sub-tags

> `ASSOCIATED-COM-I-PDU-GROUP-REFS`: helps to identify which ISignalIPduGroup are applicable for which ECU
>
> `COMM-CONTROLLERS`: communication controllers of the ECU, e.g. `CAN-COMMUNICATION-CONTROLLER`
>
> `CONNECTORS`: all channels controlled by a single controller
>
> `SLEEP-MODE-SUPPORTED`: specifies whether the ECU instance can be put to the low power mode
>
> `WAKE-UP-OVER-BUS-SUPPORTED`: specifies whether wake-up over bus is supported

`CAN-CLUSTER`: CAN bus specific cluster attributes

`I-SIGNAL`: Signal of the interaction layer (IL). The run-time environment (RTE) supports so called ”signal fan-out” where the same system signal is sent in different SignalIPdus to multiple receivers. The System Signal is unique per System. To support the RTE ”signal fan-out” each SignalIPdu contains ISignals. If the same System Signal is to be mapped into several SignalIPdus there is one ISignal needed for each ISignalToIPduMapping.

`FRAME`: describes the attributes of each frame; important sub-tag is `PDU-TO-FRAME-MAPPING` which defines the composition of Pdus in each frame

`I-PDU-GROUP`: refers to the I-PDUs that should be always kept together

`END-TO-END-PROTECTION`: the meta class; represents the ability to describe a particular end to end protection

3.3 Project Setup Procedure in PROVEtech:TA

As mentioned at the beginning of this chapter, the project setup procedure contains from several steps that must be carried out in order to obtain a full-fledged rest-bus simulation.

Naturally, the first step during the setup of any rest-bus simulation is to connect an ECU to the software. The ECU used in this project is connected to the software over a CAN-to-USB interface, namely Vector CANCaseXL VN1630. The ECU has two buses – CAN1 and CAN2. Due to the fact that the bus type is High Speed CAN, 120 Ω termination resistors must be placed at both ends.

Next step is to create an XML file for the rest-bus simulation known as RBS descriptor. This file can be generated by a tool called RBSConfig which comes with PROVEtech:RE. To generate this file, the DBC file or the ARXML file is needed as an input. The generation and configuration of the RBS descriptor is described in Section 3.3.1.

After that, other configuration file in the XML format must be created in order for PROVEtech:RE to be able to communicate on the CAN bus. This configuration file is PROVEtech specific. At this time, no tool is available for the generation. Thus, it must be created manually. To generated this XML specific file, DBC file or ARXML file, and RBS descriptor XML are needed. The structure of this XML is described in Section 3.3.2.

Now, the rest-bus simulation configuration can be loaded in PROVEtech:TA. PROVEtech:RE is connected by selecting *Settings* → *Configurations* → *Add Configuration*. In the new configuration, a new model must be created. In the new model, hardware type *PROVEtech:RE* must be selected, and the path to the XML configuration file is added. Finally, the workspace can be adjusted in PROVEtech:TA. This means, for example adding signals needed during the rest-bus simulation or adding a message window where all sent and received messages are displayed. The setup of the workspace specific for this project is described in Section 3.3.3.

It is important to mention that the most of the rest-bus simulation in PROVEtech:TA is carried out by PROVEtech:RE. It enables connecting the real hardware into PROVEtech:TA, as well as creating the rest-bus simulation. More about PROVEtech:RE can be found in Section 2.3.5.

3.3.1 XML – RBS Descriptor

The RBS descriptor in the XML format is generated using the RBSConfig tool. After launching this tool, a new network is created by selecting *Datapool* → *New network* → *CAN network*. During this step, a user chooses the path to the DBC or the ARXML file. This adds all messages and signals defined in the DBC or the ARXML file to the configuration. As the next step, the transmission type of every message must be defined. Some of the supported transmission types are the following:

`Cyclic` – the message in sent periodically

`Spontaneous` – the message is always sent when at least one of its signals changes its value

`Cyclic if active` – the message is sent periodically if at least one of its signals differs from its inactive value

`Spontaneous with minimum delay` – the message is sent when at least one of its signals changes its value and the last transmission was sent at least the specified interval ago

`Spontaneous with repetition and minimum delay` – if a signal of the message changes its value, the message is sent with a minimum delay; after that it gets repeated for n times with a cycle time of t

There are other transmission types such as `Cyclic and Spontaneous`, `Cyclic and Spontaneous with minimum delay` or `Cyclic if active and spontaneous with minimum delay`. These types are combination of the above mentioned. More information about these transmission types can be found in [9].

The RBSConfig tool enables changing other attributes, e.g. Cycle time, Delay time, etc. To make the rest-bus simulation enabled, the attribute `IsRbsEnable` must be set to `True`. After that, the RBS descriptor can be generated by selecting *Configuration* → *Save all as XML*. The generated XML file also includes a new signal TX_enable which enables sending messages in PROVEtech:TA.

3.3.2 XML – PROVEtech Configuration File

The specific configuration XML file for PROVEtech:RE must be created manually. At this time, no tool that would be able to create this kind of file is available. First, a clock source is defined between `EventSource` tags. There are two types of clocks supported, namely `Idle` and `SysClock`. For the system clock `SysClock`, a time period must be set in ms to tell the source how often it should generate events. Furthermore, the CAN hardware device must be defined between `Device` tags. The device used in this project is VectorCANCaseXL VN1630A. Individual channels are defined between `Port` tags. For the rest-bus simulation, a channel must contain the following tag attributes:

`HWPort` – hardware channel, in this project e.g. VN1630 Channel 1, 0, 0

`BitRate` – in this project 250 000 bit/s for channel 1, 500 000 bit/s for channel 2

`NWDescriptor` – path to DBC or ARXML file

`RBSDescriptor` – path to RBS descriptor file

Next, a so called `Peer` must created which informs the software that the rest-bus simulation should be created on the given CAN channel. For the demonstration, a part of the created XML configuration file is shown in Fig. 3.3.

3.3.3 Setup of Workspace

It is completely up to a user how the workspace is arranged. It may be set up in innumerably many ways. Signals that are needed during the rest-bus simulation are added by selecting the button *Signal Selection*. Signal controls can be added to the workspace with different appearance. A user can choose for example a binary switch, a slider, a strip chart, a gauge, etc. Besides controls, a message window can be added to the workspace by right-click and selecting *Create Message Window*. All sent and received messages can be seen in the message window. For the purpose of this project, only binary switches are needed. The following signals of the buttons for head-unit menu navigation have been used in this thesis: NORTH – move up, SOUTH – move down, WEST – move left, EAST – move right, and then the buttons for mode switching and return, i.e. NAVIGATION, RADIO, PHONE, RETURN, etc. Two important steps must be carried out for the rest-bus simulation to be functional. First, before pressing a button, a signal called update bit[1] must be set. Otherwise, the corresponding button will not work! Moreover, a signal called TX_enable must be set to allow transmitting of messages. The workspace for this project is shown in Fig. 3.4.

[1] e.g. NORTH_UB

3.4 Project Setup Procedure in CANoe

The configuration of the rest-bus simulation in CANoe is carried out only by the software itself. Except DBC or ARXML file, there is no need for other configuration files. On the other hand, the configuration inside the software includes more steps than in PROVEtech:TA.

At the beginning, as described in Section 3.3, an ECU is again connected to the software using the CAN-to-USB interface Vector CANCaseXL VN1630 and placing termination resistors at both ends. The next step is to configure hardware in the menu *Configuration* → *Network Hardware Configuration*. It is important to set the correct baud rate and bit timing registers here.

After that, a database should be added into the new project. This step, as well as the configuration of the ECU, is described in Section 3.4.1. If the added database is in DBC format, it can be edited using CANdb editor. It allows changing attributes of nodes, messages and signals, and therefore modify a DBC file. Some of the properties of a message can be directly changed by selecting *Configuration* → *Interaction Layers*. This will not modify the DBC file.

Finally, the workspace can be created in *Panel Designer*. The workspace contains controls for signals needed during the rest-bus simulation. The creation of the workspace is described in Section 3.4.2.

3.4.1 Simulation Setup

First of all, the database must be added to the project. This is done in *Simulation Setup* window by selecting *Networks* → *CAN* → *Database* → *New*. The relevant DBC or ARXML file is selected.

After that, a network node, which represents the ECU, must be added to the network. This is done by selecting *Networks* → *CAN* → *Nodes* → *Insert Network Node*. After that, the node must be configured by right-clicking and selecting *Configuration*. The most important step during this configuration is to add so-called components to the node. These components are runtime libraries in the form of DLL files. They add relevant functions to the rest-bus simulation that can be further used during creation of a test script in CAPL. The basic library used during the rest-bus simulation is Vector Interaction Layer (IL). Vector IL provides signal-oriented means of accessing the bus. It also performs mapping of signals to their sent messages and controls the sending of these messages as a function of the so called Send Model. This send model contains functions as `ILSetSignal` that performs setting a signal bit on the ECU. However, the modeling library can be OEM specific. In this case, different Send Models may be provided by the OEM as well. Subsequently, the conversion of the rest-bus simulation from CANoe to other software may become virtually impossible. For example, PROVEtech:TA currently does not support vendor specific DLL libraries.

In the last step, the state is switched to *Simulated* and CAPL script can be added in *Node Specifications*. The CAPL script then may be opened by selecting *Edit* on the node in the simulation setup window. This opens the CAPL browser where test scripts can be edited. After the node configuration has been established and test scripts compiled, whole project must be compiled by selecting *Configuration* → *Compile All Nodes*.

3.4.2 Panel Designer

The panel designer is used to create a custom workspace that will be displayed during the rest-bus simulation as separate windows. A user has wide range of controls available. Starting from standard button, check box, radio button, up to LCD control, meter or even media player. After adding a control to the workspace, the signal must be assigned to the control. In the properties of the control, *Symbol* → *Attach Signal* is selected. Moreover, the control can be assigned an environment or a system variable after it has been initialized in CANoe. The variable may serve, for example as the initializer of a CAPL script. Standard buttons and check boxes are sufficient for usage in this project. The workspace of this project is shown in Fig. 3.5.

3.5 Test Languages

This section describes programming for writing automated test scripts in tools PROVEtech:TA and CANoe. PROVEtech:TA uses WinWrap Basic (WWB) as the programming language for writing test scripts. WWB has the basis of Visual Basic, and has several similar variants. The variant used during this project is WWB-COM. CAN Access Programming Language (CAPL) is used for writing test scripts in Vector CANoe. This language is very similar to the C programming language. WWB is described in Section 3.5.1. CAPL is then described in Section 4.4.3. The both languages are quite different because they come from completely different basic programming languages. Therefore, some of the main differences are described in Section 3.5.3.

3.5.1 WinWrap Basic

WinWrap Basic (WWB) is the programming language based on Visual Basic. It has been developed by Polar Engineering and Consulting for programs of various types as an alternative to ActiveX, Visual Basic for Applications, and Visual Studio Tool for Applications. Currently the WWB supports three compatibility modes – WWB-COM for Visual Basic for Applications compatibility, WWB.NET for Visual Basic.NET compatibility and WWB.NET/Compiled for Visual Basic.NET compiled script code. The COM mode supports some extensions in comparison to the compatibility mode, e.g. additional data types, operators or conversion functions. The syntax and semantics is mentioned in Section 3.5.1.2. PROVEtech:TA tool includes extensions to WWB covering objects and functions for different tasks. These additional features enable performing bus testing and analysis. Therefore, the full rest-bus simulation does not have to rely only on PROVEtech:TA itself but can be enhanced by using automation scripts. Details about extensions to WWB are described in Section 3.5.1.1.

3.5.1.1 Extensions to WWB

The extensions to WinWrap Basic for PROVEtech:TA include classes, objects and functions with various tasks for the rest-bus simulation. Most of them are global objects and can be used directly with no need for creating new instances. These global objects are described below:

- *Diag* – contains the necessary methods for the diagnosis of control units via the diagnostic tool CAESAR from Daimler and I+ME ACTIA (built into the test system) available
- *Dgn* – allows access to the MCD-3 D diagnostic classes, i.e. to every diagnostic service and diagnostic job
- *FaultSim* – allows control of the PROVEtech:TA fault simulation module
- *GUI* – commands for controlling the workpage and the cockpit of PROVEtech:TA
- *Measure* – provides methods for managing data acquisitions, i.e. start/stop data acquisition, save signal curves, etc.
- *System* – manages signals, a test protocol with sub-protocols and provides general system functionality
- *TM* – contains methods and classes to automatize the Test Manager functionalities

Further, *Evaluation* class provides commands for evaluating data acquisition. In order to use this class, it is necessary to create its instance first.

3.5.1.2 Syntax and Semantics

PROVEtech:TA supports different WinWrap Basic variants. The special comment `#Language` must be placed at the beginning of the code in order to determine the variant used. The syntax and semantics is the same as for Visual Basic. During this project WWB-COM is used. This includes enhanced Visual Basic for Applications compatibility. The new enhancements are:

- data types – Decimal, SByte, UHuge_, UInteger, ULong
- instruction `Return`
- conversion functions – `CSByte`, `CUHuge_`, `CUInt`, `CULng`
- logical operators – `AndAlso`, `IsNot`, `OrElse`

Visual Basic in general does not use semicolons to terminate fragments of code. All functions or procedures, as well as control statements must be ended by the relevant

`End`, e.g. `End Sub`, `EndIf`, `Wend`[2], etc. WWB-COM offers many data types (only the type char is missing[3]). A full documentation of syntax can be found at [10].

As mentioned in Section 3.5.1.1, PROVEtech:TA contains extensions that enable the development of the rest-bus simulation by using WWB. Most importantly, it includes the data type `CanMsg` that is used for the manipulation with messages on a bus. Other message types for LIN or FlexRay messages are introduced as well. Moreover, it contains objects for message filters or message queues. The name of an object must always precede a particular function. Example is shown below:

```
Sub Main
        System.SetSignal("testSignal",10.5)
        Measure.Start
        Wait 10
        Measure.Stop
End Sub
```

3.5.2 CAPL

CAN Access Programming Language (CAPL) is a programming language inspired by the syntax of the C programming language. CAPL is primarily used with PC-based environments Vector CANalyzer and mostly CANoe. This language has been developed in order to meet requirements for the development of distributed embedded systems based on CAN bus. During these days however, other buses such as LIN or FlexRay are supported. With CAPL programs it is possible to simulate a network and node behavior, and to perform bus testing and analysis. Therefore, comprehensive tests can be carried out because the user does not rely only on CANoe tool itself. CANoe however, includes CAPL browser that is the primary editor for CAPL language, including the compiler for a created code.

3.5.2.1 Syntax and Semantics

The development of a CAPL program is possible outside the built-in CAPL browser. However, the special beginning and ending comment sequence must be included at the beginning/end of every section. These sequence have the following format: beginning – `/*@@xxx:  */`, end – `/*@@end */`. The exact form of comments available for this project is mentioned in Section 4.1.2.2.

As mentioned above, the similarity to C language makes CAPL syntax and semantics easy to understand. It is important to keep in mind though that CAPL does not support many concepts from C language, such as pointers, structures, unions, etc. CAPL does not support symbolic constants (i.e. macros) either. The supported data types are following:

char, byte, int, word, long, dword, float, double, message, timer, msTimer

[2]ending of while statement

[3]data type Byte or String can be used instead

For creating expressions, several operators are introduced, including arithmetic operators, assignment operators, bitwise operators, etc. Then there are statements that are equivalent to those in the C programming language. Among them

- Declaration and initialization of variables
- Arrays
- Expressions
- Control statements.

After that, there are features that are CAPL specific. First, the declaration of global variables is done inside the section `variables`. The example is given below:

```
/*@@var */
variables { ... }
/*@@end */
```

Next, CAPL is organized in so called event procedures. Each event is associated with a single event. When the event occurs, the corresponding procedure will execute. These events include e.g. reception of a message (`on message`), expiration of a timer (`on timer`), change in an environment variable (`on envVar`), etc. These event are classified as follows:

- Message events
- Timer events
- Keyboard events
- Controller events
- Error frame events
- System (tool) events
- Variable events

Furthermore, a user can define its own functions like in C language. However, CAPL has additional functions that provide a variety of special purpose operations that are useful during the rest-bus simulation. These functions may be e.g. setting a value of a signal, reading a name of database name, sending a message, resetting a controller, etc.

3.5.2.2 CAPL DLLs

Additional functions that would be used during the rest-bus simulation can be created by modeling gateway nodes. This gateway enables communication between two network buses, and is implemented by using so called node layer DLLs. CANoe includes the interaction layer as the base which provides most of the functionality needed for communication between buses. However, these DLLs can sometimes be created by OEM and delivered to a customer with a created rest-bus simulation. These OEM specific DLLs can introduce new functions that can be later used during the rest-bus simulation for various purposes.

3.5.3 Main Differences

This sections points out the main differences between the CAPL and WWB-COM language. Naturally, the syntax and semantics is different from each other. It is not the goal of this section to describe all syntax differences in detail though. The section should be primarily used as a guide when determining which translations are not possible, or what modifications must be made in order to make them translatable. Due to the reason that both languages are quite different from each other (CAPL based on C language, WWB based on Visual Basic), not all details are described, plus general syntax and semantic differences are not explained since it would be over the scope of this project.

Data Types

Some of the data types exist in WWB under different name. First of all, `char` is not present in WWB-COM. The type `String` is used instead. The numerical data types are in Table 3.1.

Furthermore, CAPL contains enumeration data types `message`, `timer` and `msTimer`. These data types do not have exact equivalents in WWB. The data type `message` is provided to WWB by PROVEtech:TA library. However, messages need to be declared little differently. CAPL declaration includes both, an identifier and variable name, at once. In WWB, the name must be declared first and the ID can be subsequently assigned by accessing an object. The example of both is given below:

```
message 0x101 msg;       /* CAPL */
```

```
Dim msg An New CanMsg    ' WWB '
msg.ID = &H101
```

The type `timer`, or similar, does not exist in WWB. Due to this fact, the straightforward usage of timers is possible only in CAPL. One possible option is to use the function `Wait`. However, this command is not exact by means of real-time. For exact timing, the Automation Library should be used [1]. By the time of writing this thesis, latest information has been that timers should be available for PROVEtech:TA in the future.

Array Indexing

When declaring an array, a number specified in the array brackets in CAPL corresponds, as in C language, to the size of an array. However, when declaring an array in WWB, a number corresponds to the highest index as the lowest index is zero. Thus, the amount of array elements is not specified. The following example shows the difference:

```
int arr[10]   /* CAPL */
```

CAPL	WWB	size
int	Integer	16 bit signed
word	UInteger	16 bit unsigned
long	Long	32 bit signed
dword	ULong	32 bit unsigned
-	Single	32 bit signed
float, double	Double	64 bit signed
-	Huge	64 bit signed
-	UHuge	64 bit unsigned
-	Decimal	96 bit signed

Table 3.1: Numerical data types in CAPL and WWB

```
Dim arr(9) as Integer  ' WWB '
```

The array declared in WWB contains elements 0 to 9. Thus, it does contain same amount of elements as the array declared in CAPL.

Moreover, CAPL supports expressions inside the array brackets, e.g. `[index++]`. This is not possible in WWB and has to be solved in another way.

For Loop

CAPL includes the C-style For loop. WWB allows two syntaxes of the For loop, both different from CAPL though. First type is the iteration from one value to some final value with an iteration step. For example:

```
For i = 1 to 1000 Step 100
        Debug.Print i
Next i
```

Next possible implementation is Python-like For loop for iteration over set of items. This kind of For loop has no equivalent in CAPL. For example:

```
For Each Document In App.Documents
        Debug.Print Document.Title
Next Document
```

Jump Statements

WWB does not include these jump statements – `break`, `continue`. The statement `return` is included in WWB-COM, therefore, it can be used in the same manner as in CAPL.

Global Variables

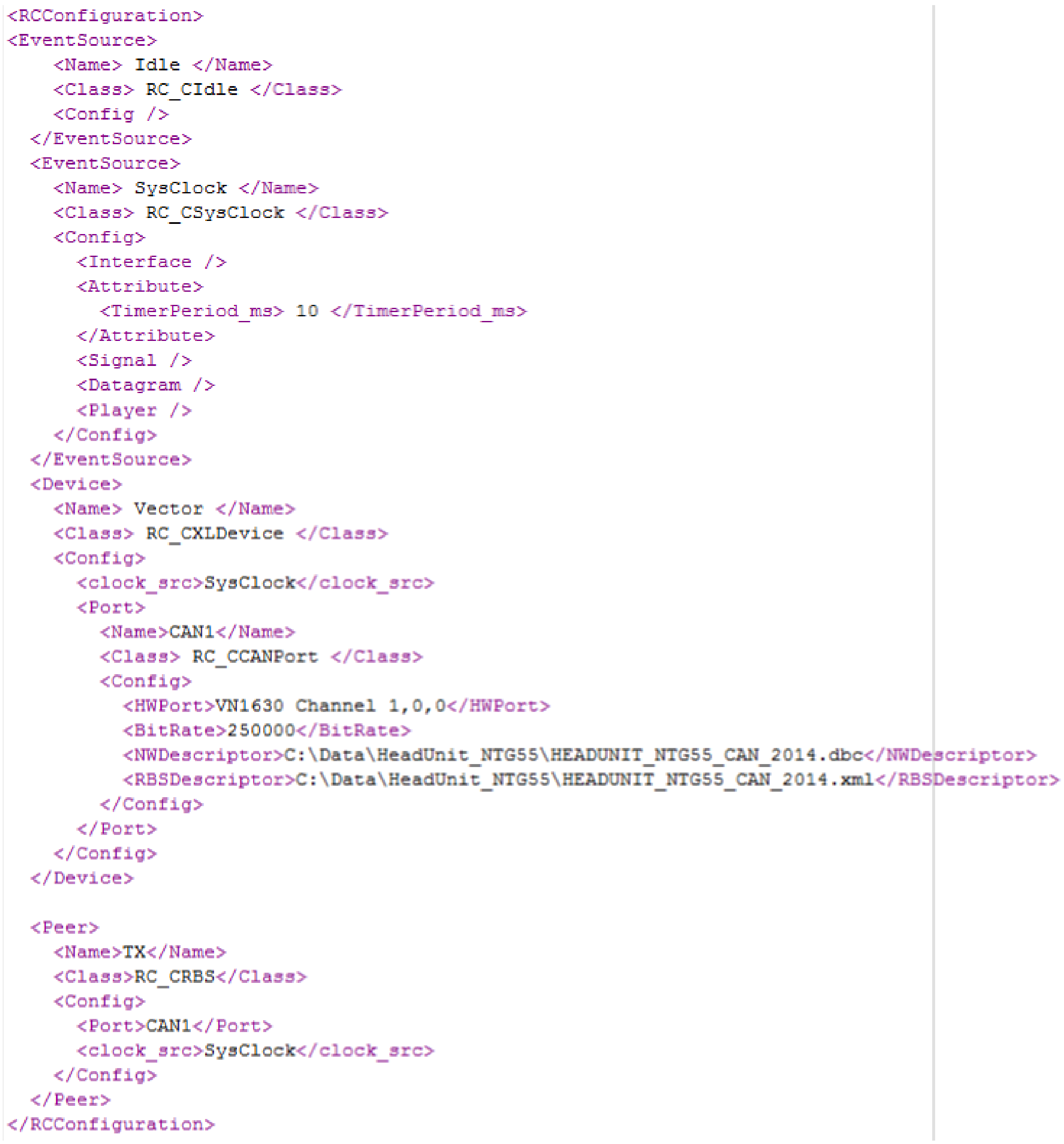
The global variables in CAPL are defined inside the section `variables { ... }`. Because no such a section exists in WWB, it is recommended to place the global variables inside the main method in a WWB script[4].

Events

As mentioned in Section 3.5.2.1, a CAPL program may include events. Whenever an event occurs, the procedure belonging to the event is triggered. There are no such events in WWB. It means, it is not possible to execute code based on e.g. pressed key, changed variable, expired timer, etc. Because of this reason, the keyword `this` has no usage in a WWB program. The keyword `this` can be thought of as a "pointer" to a particular message, timer, variable, etc. for which an event is declared.

Nevertheless, PROVEtech:TA enables code execution based on a received message. First option would be to implement special DLL files for this purpose. Another, more convenient option is to use the Automation Library. By using a real-time script, it is possible to wait for reception of a message and execute some code afterwards. It can be seen that this is not the full equivalent of the CAPL event. A message can be received any time during a simulation in CANoe and the particular code will execute. In PROVEtech:TA however, we must start a real-time program and wait for the reception of that message, so it is not possible to make one-to-one translations from CAPL events to Automation Library events.

[4]i.e. right after `Sub Main`



```
<RCConfiguration>
<EventSource>
    <Name> Idle </Name>
    <Class> RC_CIdle </Class>
    <Config />
  </EventSource>
  <EventSource>
    <Name> SysClock </Name>
    <Class> RC_CSysClock </Class>
    <Config>
      <Interface />
      <Attribute>
        <TimerPeriod_ms> 10 </TimerPeriod_ms>
      </Attribute>
      <Signal />
      <Datagram />
      <Player />
    </Config>
  </EventSource>
  <Device>
    <Name> Vector </Name>
    <Class> RC_CXLDevice </Class>
    <Config>
      <clock_src>SysClock</clock_src>
      <Port>
        <Name>CAN1</Name>
        <Class> RC_CCANPort </Class>
        <Config>
          <HWPort>VN1630 Channel 1,0,0</HWPort>
          <BitRate>250000</BitRate>
          <NWDescriptor>C:\Data\HeadUnit_NTG55\HEADUNIT_NTG55_CAN_2014.dbc</NWDescriptor>
          <RBSDescriptor>C:\Data\HeadUnit_NTG55\HEADUNIT_NTG55_CAN_2014.xml</RBSDescriptor>
        </Config>
      </Port>
    </Config>
  </Device>

  <Peer>
    <Name>TX</Name>
    <Class>RC_CRBS</Class>
    <Config>
      <Port>CAN1</Port>
      <clock_src>SysClock</clock_src>
    </Config>
  </Peer>
</RCConfiguration>
```

Figure 3.3: Example of a configuration file for PROVEtech:RE

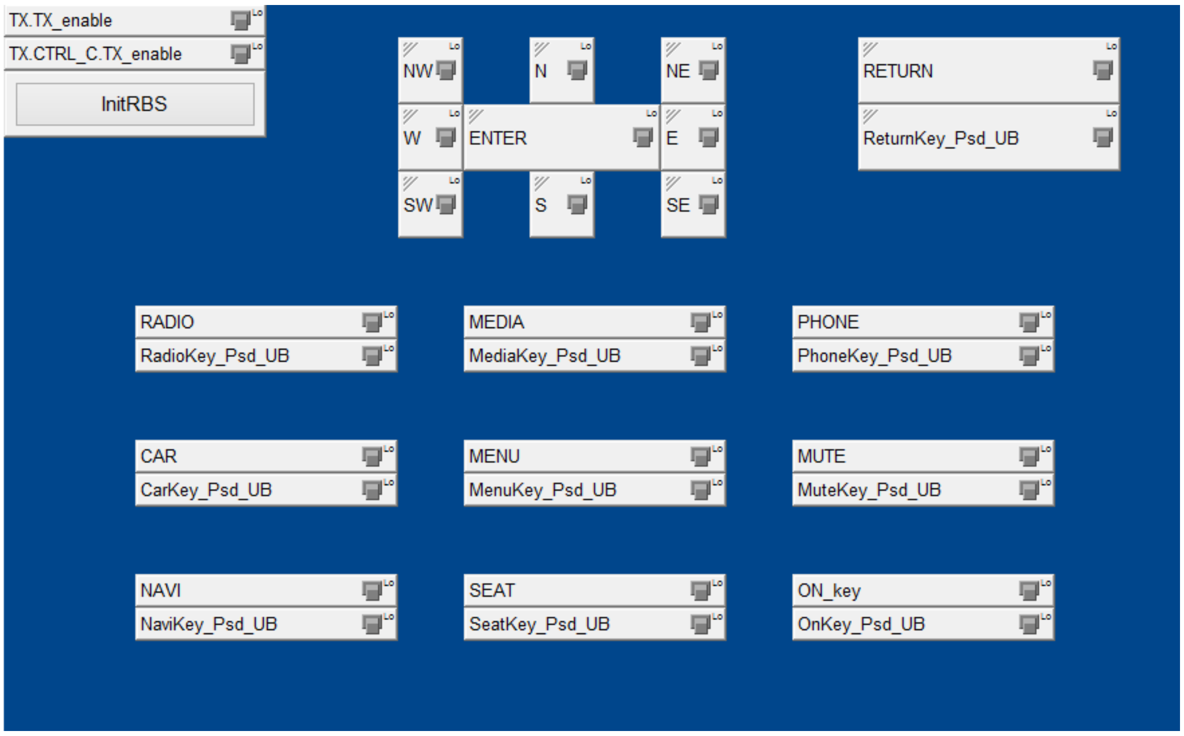


Figure 3.4: Created workspace in PROVEtech:TA

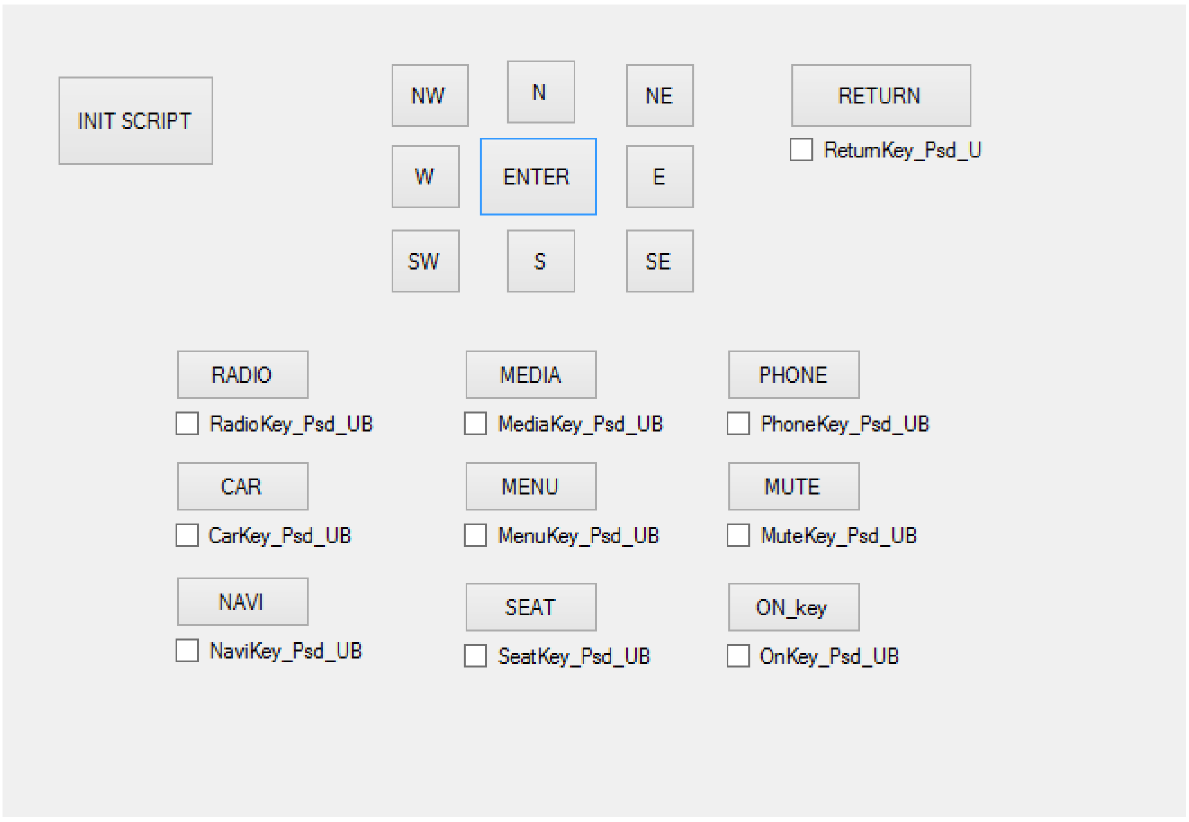


Figure 3.5: Created workspace in CANoe

Chapter 4

Converter

The goal of the converter is to convert a rest-bus simulation created in CANoe to other software tools. This thesis deals mainly with the conversion into PROVEtech:TA. The main focus of the conversion is on converting test scripts from CAPL to another language. For this purpose, the lexer and parser have been developed in Python with Python Lex-Yacc (PLY) package. More about the developed lexer and parser can be found in Section 4.1. The converter supports two target platforms that use two different languages. Specifically, PROVEtech:TA that uses WinWrap Basic as the language of test scripts, and Linux that uses C as the language of test scripts. The overview of the translator is given in Section 4.2. The overall structure of the code translation is displayed in Fig. 4.1. Additionally, the conversion of a workspace from CANoe to PROVEtech:TA has been created as well. Since the graphical user interfaces of PROVEtech:TA and CANoe are very different from each other, it is not possible to make one-to-one conversions of the graphical user controls of the given rest-bus simulation. So, it may be necessary to spend some time arranging the graphical controls manually after the conversion. The workspace conversion is described in Section 4.3. XSLT language was used for graphics conversion. Additionaly, XSLT was used for the feature of setting certain values in XML configuration file for PROVEtech:TA. The usage of the whole program is described in Section 4.4.

4.1 Lexer and Parser

The lexer and parser has been developed with Python Lex-Yacc (PLY) package. PLY is a pure Python implementation of the known unix-based tools Lex and Yacc, both written in C. PLY relies on the way in which traditional Lex and Yacc tools work. Both of these modules, `lex.py` and `yacc.py` are found in Python package called `ply`. From now on, names Lex and Yacc refer to `lex.py` and `yacc.py` modules. The Lex module is used for carrying out the lexical analysis. This means, it breaks input text into a collection of tokens specified by a collection of regular expression rules. The Yacc module, on the other hand, is in charge of parsing a code. It is used to recognize language syntax that has been specified in the form of context-free grammar. The Yacc module uses left-to-right, rightmost derivation (LR) parsing. It generates its parsing

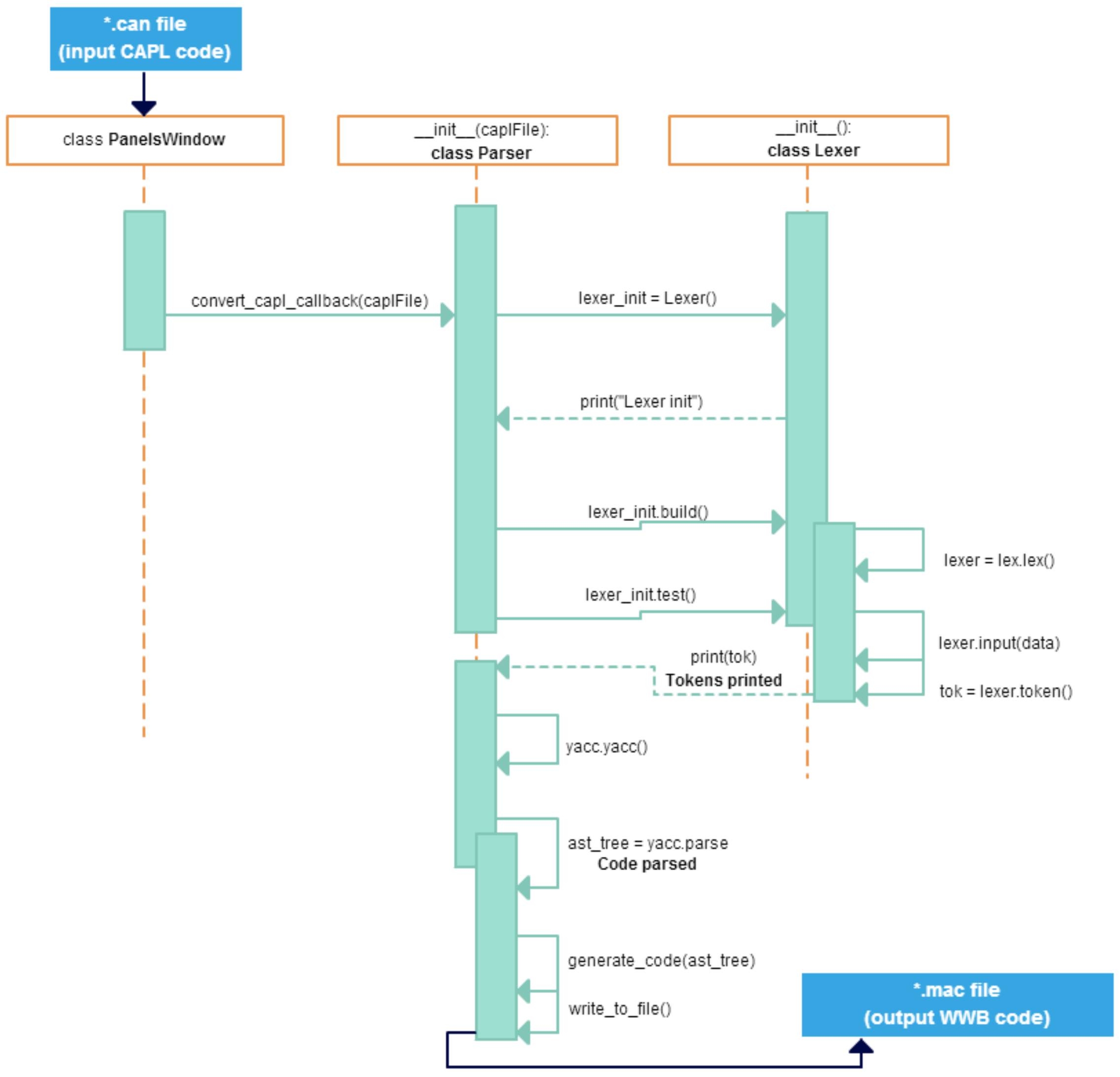


Figure 4.1: An UML sequence diagram of the code translation from CAPL to WWB

tables using Look-Ahead LR (LALR) algorithm.

The Lex and Yacc modules are meant to work together. Specifically, Lex provides an external interface in the form of the `token()` function that returns a next valid token on the input stream. Yacc calls this repeatedly to retrieve tokens and invoke grammar rules. The output of Yacc in this project is an Abstract Syntax Tree (AST). Building the AST for this project is described in Section 4.1.2.2. The implementation of Lex is further described in Section 4.1.1. The implementation of Yacc is then described in Section 4.1.2. The biggest difference between the Yacc module in Python and the Yacc program in Unix is that Unix Yacc converts grammar specification to a C code whereas Python Yacc interprets the specification directly. This means that there are no extra source files nor a special compiler step, e.g. running Yacc to generate Python code for the compiler. Since the generation of parsing tables is relatively expensive, PLY caches the results and saves them to a file. If no changes are detected in the input source, the tables are read from the cache. Otherwise, they are regenerated. [11]

4.1.1 Lexical Analysis

As mentioned above, the Lex module (`lex.py`) is used to tokenize an input string. This means, it breaks the input string into chunks called tokens. Tokens are given the identifiers which are used by the parser. The tokenizing breaks input into separate tokens. An example of a simple input expression and a tokenized output may look as follows:

```
x = 1 + 2;
('ID','x'), ('EQ','='),
('DEC_NUM','1'), ('PLUS','+'), ('DEC_NUM','2')
```

The identification of individual tokens is done by defining regular expression rules. Further, it is described how the definition of regular expressions was done during this project.

Beforehand, it is mentioned that the class `Lexer` is defined in the file `lexer.py`. As the first step, a list of `tokens`, which defines all possible token names, must be provided. The list of tokens was divided into several “sub-categories” where tokens are defined according to [12]. These categories are:

- arithmetic operators – +, −, ∗, /, %, ++, −−
- assignment operators – =, + =, <<=, & =, · · ·
- boolean operators – !, ||, &&
- bitwise operators – , &, |,^,<<, >>
- relation operators – ==, ! =,>, >=, <, <=
- miscellaneous operators – ., ?, :
- others

The sub-category `others`, from the categories listed above, contains all other token names that specify, for example a white space, parentheses, an identifier, a comment, etc. The following code is an example of defined tokens:

```
tokens = [
        # arithmetic operators
        'PLUS',
        'MINUS',
        'TIMES',
        ...
        # others
        'WS',
        'LCBR',
        'RCBR',
        'ID',
        'COMMENT',
        ...
        ]
```

This token list is then appended by a list that specifies reserved words and a list that specifies different number formats, i.e. `int`, `float` and `hex`. The reserved words handle all data types from CAPL, as well as keywords of control statements such as `if`, `else`, etc.

```
reserved_words = {
'char' : 'CHAR',
'byte' : 'BYTE',
'int' : 'INT',
        ...
'void' : 'VOID',
'if' : 'IF',
'else' : 'ELSE',
        ...
'return' : 'RETURN',
'default' : 'DEFAULT',
'this' : 'THIS',
}

ved_numbers = {
       'dec_num' : 'DEC_NUM',
       'float_num' : 'FLOAT_NUM',
       'hex_num' : 'HEX_NUM',
       }

           tokens = ['PLUS','MINUS,...] + list(reserved_words.
              values()) + list(reserved_numbers.values())
```

Each token is specified by writing a regular expression rule. Each of these rules are defined by declarations with special prefix `t_` to indicate that they define a token. The regular expressions are defined by a string with prefix `r`. The simple regular expressions are specified first for the tokens. The simple regular expressions include operator signs, char, and so on. More about the regular expressions rules can be found in [13].

```
t_PLUS = r'\+'
t_MINUS = r'\-'
...
t_LSHIFT_EQ = r'\<\<\='
...
```

Tokens with slightly more complicated expressions are listed, together with examples of them, in Table 4.1. Moreover, the token `CAPLEVENT_word` must be mentioned separately. It handles the keywords of CAPL events declarations. More can be found in Section 4.1.1.1. As the last, tokens that are defined as series of more complex regular expressions are explained in Section 4.1.1.2.

When using the lexer, data are first fed into `input` method. After that, `token` is called repeatedly to produce tokens. The following code shows the implementation of

token rule	regular expression	example
t_NUM	`r'[0-9]'`	123
t_STRING	`r'\"(\\.|[^\\"])(\\.|[^\\"])+\"'`	”hello world”
t_ID	`r'[a-zA-Z_][a-zA-Z0-9_-]*'`	x09_c
t_CHARC	`r'\"(.)\"'`	”a”
t_KEY	`r'\'(.)\`■	’b’
t_COMMENT	`r'/\*([^\*]|\*[^*/])*\*/'`	/* comment */

Table 4.1: Definition of slightly complicated token rules

using `input` and `token`.

```
def test(self,filename):
        data = open(filename).read()
        self.lexer.input(data)
        while True:
            tok = self.lexer.token()            # take next token
            if not tok: break                   # EOF reached
            print(tok)
```

4.1.1.1 CAPL Events Token

Since a program developed in CAPL may be organized around event procedures, a token with a regular expression that handles the keywords of this events must be introduced. The token `CAPLEVENT_word` currently supports the following general and CAN specific events:

> `preStart`, `start`, `stopMeasurement`, `busOff`, `timer`, `key`, `message`, `errorActive`, `errorPassive`, `warningLimit`, `errorFrame`, `envVar`, `sysVar`, `preStop`

The rest of events is not currently supported because they are outside the scope of this thesis assignment. It can however be easily implemented by either modifying `t_CAPLEVENT_word` token rule or by creating a new token with a specific token rule.

4.1.1.2 Token Rules as Functions

If some kind of action needs to be performed or if tokens shall be build from series of complex regular expression rules, a token rule can be specified as a function. In case of a complex regular expression (example in Table 4.1), a variable with this regular expression is initialized beforehand. The `@TOKEN` decorator is then used with either a token rule or a variable representing the complex regular expression as its parameter. This decorator then attaches to a function as the parameter `t`. Generally, the function always takes a single argument `t` which is an instance of `LexToken`. It has the attributes `type` – token type, `value` – actual text matched, `lineno` – current line number and `lexpos` – position of the token relative to the beginning of an input.

First, the following particular function introduces a rule to match an identifier and to do a special name lookup.

```
@TOKEN(t_ID)
t_RESERVED(self,t):
    t.type = self.reserved_words.get(t.value, 'ID')
    return t
```

This function identifies the token rule `t_ID` and assigns a value of an entry from the reserved words set.

Complex regular expressions (i.e. built from series of regular expressions) are handled by a function in the form `t_FunctionName(self,t)`. These regular expressions include – different formats of digits (`float` and `hex`), array brackets (e.g. `[10][10]`), the declaration of CAPL events (`on` keyword, e.g. `on message`) and specific CAPL header and footer comments that circumscribe either CAPL events or functions (e.g. `/*@@message:msg:*/`, `/*@@end */`). The following example shows the function for detecting token `CAPLEVENT`. This token is built from series of complex regular expressions that are defined in the decorator `on_event_declar`.

```
on_event_declar = r'(on' + t_WS + r')+(' + t_CAPLEVENT_word + r')+'

@TOKEN(on_event_declar)
t_CAPLEVENT(self,t):
return t
```

Another functions for handling another complex regular expressions are written in the same manner.

4.1.2 Parsing

The Yacc module (`yacc.py`) is used to parse language syntax. The name ”yacc” stands originally for ”Yet Another Compiler Compiler”. The syntax is specified in terms of Backus-Naur Form (BNF) grammar. An example of such a unambiguous grammar might look as follows:

```
expression  : expression + term
            | expression - term
            | term

term        : term * factor
            | term / factor
            | factor

factor      : NUMBER
            | ( expression )
```

In the grammar, symbols such as `NUMBER`, $+$, $-$, are known as terminals. They correspond to raw input tokens. Identifiers such as `term` or `expression` are known as non-terminals. They refer to grammar rules consisting of terminals and other rules. This grammar is defined in docstrings which are then inserted into a Python code. As an example, the simple arithmetic expression from the previously described grammar is shown:

```
def p_expression_plus(p):
        'expression : expression PLUS term'
        p[0] = p[1] + p[3]
```

The expression grammar given in the previous example has been written in order to eliminate ambiguity. However, generally it is very difficult to write grammar with all ambiguity eliminated. The dealing with ambiguous grammar is described in Section 4.1.2.1.

Yacc uses the LR parsing. LR parsing can be called shift-reduce parsing as well. It is the bottom up technique that tries to recognize the right hand side of grammar rules first. During LR parsing, symbols are shift onto a stack. After that, it is looked up on this stack and the next input token, and it is searched for patterns that match one of the grammar rules. When a valid right hand side is found on the top of the stack, it is usually reduced and the grammar symbols are replaced by the grammar symbol on the left hand side. To understand this principle, the parsing of $1+2$ expression is given according to unambiguous grammar mentioned at the beginning of this section.

```
Step   Symbol stack    Input tokens   Action
----   ------------    ------------   ------
1                      1 + 2          Shift 1
2      1                 + 2          Reduce factor : NUMBER
3      factor            + 2          Reduce term : factor
4      term              + 2          Reduce expression : term
5      expression        + 2          Shift +
6      expression +        2          Shift 2
7      expression + 2                 Reduce factor : NUMBER
8      expression + factor            Reduce term : factor
9      expression + term              Reduce expression :
                                      expression + term
10     expression                     Reduce expression
11                                    Done.
```

In the project specific implementation, the `Parser` class is implemented in the file `parserPy.py`. Each grammar rule is then defined by a separate function. The function accepts an argument `p`. Under the index `p[i]`, it contains the corresponding value of a symbol in the given rule[1]. The function `p_error` catches the syntax errors, i.e. any undefined rules in the grammar context. If any syntax error is caught, the output containing the corresponding value of a symbol and the corresponding line number will be printed to the console. [11]

4.1.2.1 Ambiguous Grammar

As mentioned above, sometimes it is very difficult to define the unambiguous grammar. A developer would very often use such expression – `expression : expression PLUS expression` instead of dividing it into other "sub-rules". The grammar defined in this way is ambiguous since the parses may have two options at the same time – either shift

[1]the left hand side is under the index `p[0]`

on a stack or reduce. By default, all shift/reduce conflicts are resolved in favor of shifting. This may, for example, introduce problems during parsing arithmetic expressions. To resolve ambiguity, the Yacc module allows assigning a precedence level to individual tokens. This is done by adding the following type of code:

```
precedence = (
   ('left', 'PLUS', 'MINUS'),
   ('left', 'TIMES', 'DIVIDE'),
   ...
   )
```

4.1.2.2 Building Abstract Syntax Tree

The Yacc module provides no special functions for constructing an AST. The Python build-in library can be used in some cases. However, it turned out that this library is not very suitable for parsing the CAPL language because it contains nodes that are not present in CAPL, or on the other hand, it lacks nodes needed for building the AST for the CAPL code. For the construction, the tree structure has been created by defining `Node` class.

```
class Node:
    def __init__(self,type,children=None,leaf=None):
         self.type = type
         if children:
              self.children = children
         else:
              self.children = [ ]
         self.leaf = leaf

    def __repr__(self):
        return "{type: %s, children: %s, leaf: %s}" % (self.type,
           self.children,self.leaf)
```

The description of the AST for this project will be accompanied by displaying some parts of the AST. The appropriate description will follow in such a manner, that a reader can get acquainted how the AST is built and what syntax to use in order to ease a translation to WinWrap Basic or C when creating an input file. The terminals in the AST can be recognized by upper-case letters. Even though, it is assumed that a user will built the AST for a CAPL code that has already been compiled in some CAPL editor, by understanding the AST a user can easily identify errors in CAPL syntax that would lead to a malfunction of the parser.

By understanding the AST, a user can easily identify a syntax for which the parser does function properly.

First of all, the whole `program` is divided into `code_fragments`. Each `code_fragment` represents a node, that can be of four different types.

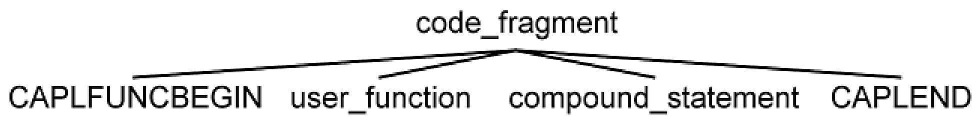


Figure 4.2: Node `Function_UD`

Listing 4.1: Example of Node: Function_UD

```
/*@@caplFunc:function1(int x): */        // CAPLFUNCBEGIN
void function1(int x)                  // user_function
{
   ...                                 // compound_statement
}
/*@@end */                               // CAPLEND
```

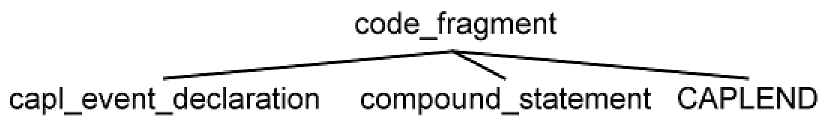


Figure 4.3: Node `CAPL_event`

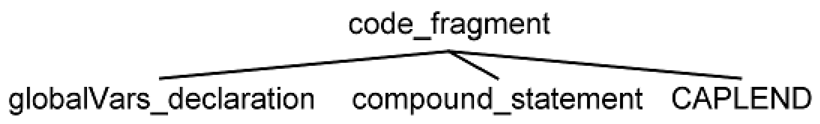


Figure 4.4: Node `GlobalVars_decl`

The terminal `CAPLEND` is a CAPL specific comment that appears at the end of every section, specifically `/*@@end */`.

The Fig. 4.2 shows the structure of a fragment for declaring a user-specific function. The terminal `CAPLFUNCBEGIN` is a CAPL specific comment which must also include the name of a user-defined function. It is followed by a user-function declaration and a compound statement. Example is shown in Listing 4.1.

The Fig. 4.3 shows the structure of a fragment for declaring CAPL events. `Capl_event_declaration` includes both, the CAPL specific comment and declaration of an event. There are only two differences from user-defined functions. First, the CAPL specific comments are different in content. Second, the CAPL events do not take any parameters. The table 4.2 shows the supported CAPL events.

Fig. 4.4 shows the structure of a fragment for declaring global variables. This is exactly the same as declaring CAPL events with the difference in comment and the declaration itself. The comment with declaration goes as `/*@@var:*/ variables{...}`.

specific comment	event
preStart:PreStart:	on preStart
startStart:Start:	on start
stop:StopMeasurement:	on stopMeasurement
busOff:BusOff:	on busOff
timer:timerName:	on timer timerName
msg:messageName:	on message messageName
errorActive:ErrorActive:	on errorActive
errorPassive:ErrorPassive:	on errorPassive
warningLimit:WarningLimit:	on warningLimit
errorFrameErrorFrame:ErrorFrame:	on errorFrame
envVar:variableName:	on envVar variableName
sysVar:variableName:	on sysVar variableName

Table 4.2: Supported CAPL events

The last code fragment are comments. The node `COMMENT` is either an one-line or multi-line comment like in C language.

As can be seen, `compound_statement` is the key part of every fragment. The `compound_statement` is bordered with curly braces. Inside curly braces are `block_items`. The `block_item` can be of five different types:

`declaration`, `assignment`, `statement`, `case_statement`, `comment`.

4.1.2.2.1 Declaration & Assignment

First of all, it is important to say that `assignment` is basically the same as `declaration`. The only difference is that the `declaration` includes the type of a variable, type of an array, type of a function, etc. The `declaration` is divided into two parts. First the declaration of messages is represented separately by node `Decl-MSG`. This node includes messages name and ID, e.g. `message 0x001 msg`. The rest of declarations is represented by node `Declaration`. The structure of node `Declaration` is displayed in Fig. 4.5. There are two leaves, `declaration_type` and `declaration_list`. The `declaration_type` can be one of following types:

`byte`, `int`, `word`, `dword`, `long`, `float`, `double`, `timer`, `mstimer`, `char`, `void`.

As can be seen, `declaration_list` includes one more single declarations separated by a comma[2].
The `declaration_single` can have several forms. The structure of these forms

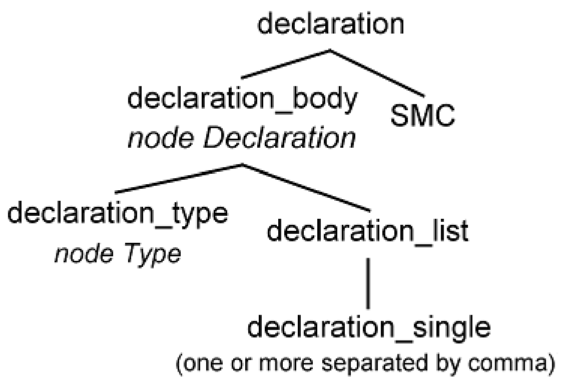


Figure 4.5: Node `Declaration`

[2]e.g. x, y, z

is explained piece by piece starting from the basic ones. Every possible structure also contains corresponding examples.

The simplest possible declaration is not displayed in a figure. However, the simplest possible declaration consists only from `entry`. The `entry` may be one of the following:

- node `ID` – e.g. `x`
- node `This` – keyword `this`
- node `ThisDot` – e.g. `this.dlc`
- node `Key` – e.g. `'a'`

The next structure of declaration, which is also not displayed, represents node `Array`, and contains only `array_brackets` above the `entry`, e.g. `x[ ]`, `y[5][5]`. Subsequently, another structures of declarations (node `Assign`, node `Assign_OP`, node `Assign_array`) already contain leaves `expression`. These structures represent assignments. The structures are shown in Fig. 4.6. It is apparent that the structures always contains `entry`

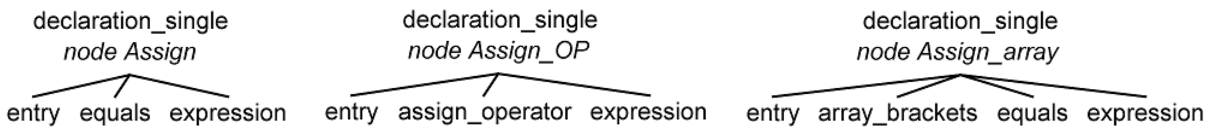


Figure 4.6: Different forms of `declaration_single` – assignments

followed either by `equals` or `assign_operator` sign, or by `array_brackets` with subsequent `equals` or `assign_operator` sign. The `assign_operator` can be one of the following:

$$+=,\ -=,\ *=,\ /=,\ \%=,\ <<=,\ >>=,\ |=,\ \&=,\hat{}=.$$

Furthermore, the `equals` sign can be followed by a so called `initializer_array` in case of an array declaration. In other cases, the `equals` or the `assign_operator` sign is followed by an `expression`. The `expression` can be either `logical_expression` or `conditional_expression` (node `Cond_EXPR`).

The `initializer_array` is either a string, a list of strings, or a list of numbers. The following example shows all possible cases:

```
{"ABCDE"}
{"ABCDE","FGHIJ","KLMNO"}
{ 1,2,3,4,5 }
{ {1,2,3},
  {4,5,6},
  {7,8,9}
}
```

As will be further explained, the parser gradually proceeds through as follows – `logical_expression`/`conditional_expression` → `binary_epression` → `unary_expression`.

The `logical_expression` is described beforehand. It may bind recursively two expressions by either a logical AND (&&) or a logical OR (||). If they are bound by a logical operator, they represent the node `Logic_EXPR`. The structure of `logical_expression` is following:

- `binary_expression`
- `logical_expression` operator `logical_expression`
- (`logical_expression`) operator (`logical_expression`)

Next, the `binary_expressions` may either stand for `unary_expression`, or may bind recursively two expressions by either a relational operator or a bitwise operator. Similarly as in the previous case, if they are bound together they represent the node `Expression`. The `unary_expression` can have the following forms:

- `single_expression`
- `value_expression`
- `capl_function_body`

The `single_expression` stands for a unary operation, i.e. the increment or decrement of a variable (postfix or prefix), a variable's complement or logical NOT. Behind the term `value_expressions` may be either a variable, a variable with array brackets[3], a constant (character or number), or a so called `message_signal`. It is important to say that a number can be either in decimal or hexadecimal format. The `message_signal` represents the message name paired together with the signal name by a double colon, e.g. message_control::signal_button. The `capl_function_body` represents the node `CAPL_fcn`. It is simply any function's name with or without parameters inside parentheses (as shown in Fig. 4.7).

To make it clear, listing 4.2 shows examples of possible expressions.

Listing 4.2: Examples of expressions

```
x + y && z << 1  // logical_expression
x + y, z << 1  // binary_expression
x++, y++, ++x, --x, ~x, !x  // single_expression
x, arr[], 999, 0x1FF // value_expression
function1(), function2(x,y) // capl_function_body
```

Next, the `conditional_expression` is described. It it does not step right down towards the `binary_expression` and so forth. It has the structure shown in Fig. 4.8. The example is following:

```
(x < y) ? z : y
```

[3]e.g. var[ ]

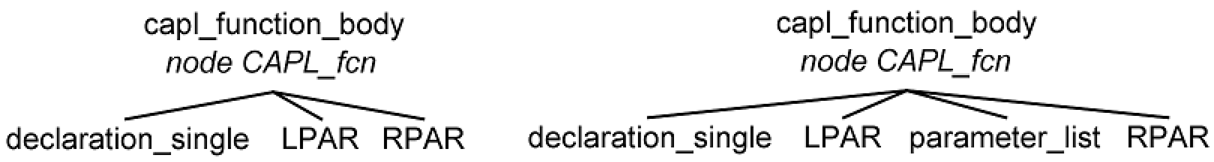


Figure 4.7: Node `CAPL_fcn`

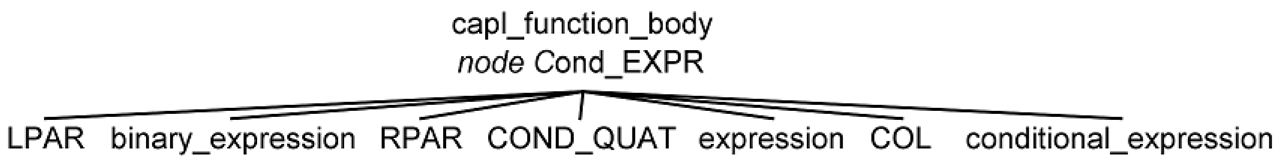


Figure 4.8: Node `Cond_EXPR`

4.1.2.2.2 Statement & Case Statement

The `statement` contains the following type of statements:

> `capl_function`, `compound_statement`, `if_statement`, `while_statement`, `do_while_statement`, `for_statement`, `switch_statement`, `jump_statement`

All of these statements are equivalent to the statements in C language, except the `capl_function`. Therefore, they do not have to be extensively described. However, a short overview is presented. The `capl_function` has already been previously described. It corresponds to the expression `capl_function_body`. It is a name of a function either with or without parameters ended by a semicolon, e.g. `function(x,y);`.

The `compound_statement` has already been completely described as well. It may contain declarations, assignments or other statements.

At all other control statements, control statements must be placed inside curly braces. Otherwise, the parser will produce an error and the parsing process will crash. Below is an example of a working and a non-working case:

```
if (speed < 100) { cruise_speed = 100; } // working
if (speed < 100) cruise_speed = 100; // non-working
```

The `if_statement` has two possible forms. First option is the statement without `else`, the other contains an `else` statement. To make it clear, the structure of the `if_statement` with `else` is shown in Fig. 4.9. That picture presents the idea how all control statements are parsed. Similarly, `while_statement` and `switch_statement` consist of the control keyword followed by a control statement inside parentheses and the `compound_statement`. The `case_statement` is parsed apart from the control statements since it has a different structure. As in C language, `do_while_statement` places the control statement at the end terminated by the semicolon. The `for_statement` must include all three parts of the statement. It is not possible to implement For loop in a manner such as `for(; number != 0 ;) { ... }`. The first part, i.e. the initialization of a variable both may and may not include the type of variable. However, it is recommended to use the variable type due to easier translation to other languages.

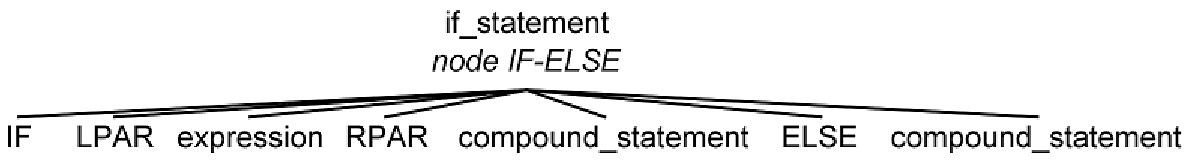


Figure 4.9: Node `IF-ELSE`

4.2 Translator

After the CAPL code is parsed, it can be translated to other languages. How the translator works for translation to WinWrap Basic is described in Section 4.2.1. The translation to the C language would be carried out similarly. The complete translation to the C language has not been developed, because for the creation of a full-fledged rest-bus simulation not only translation of test scripts from one language to another is sufficient. A simulation engine, which procures the rest-bus simulation based on an input DBC/ARXML file, is needed. Even though, the complete translation to C has not been developed, adding this feature to the translator part should be straightforward because CAPL is very similar to C. Nevertheless, the concept of translating events that react on received messages is proposed in Section 4.2.2.

4.2.1 Translation to WinWrap Basic

An AST serves as an intermediate representation of the parsed code. The converter gradually proceeds through the created AST during the translation. If a particular tree has any subtrees, the code is first recursively generated for each of the subtrees. The function `generate_code` takes the tree/subtree as an argument. The WinWrap Basic (WWB) code is then generated for every node based on the node's type and looking at node's children and leaves. The translation process is described in Section 4.2.1.1. Since the WWB scripting language is quite far different from the CAPL scripting language, the complete one-to-one translation from CAPL to WWB cannot be created. Moreover, some functions, such as timers, are not currently supported even by the available libraries in PROVEtech:TA. The main differences and the code unsupported by WWB are mentioned in Section 3.5.3. The examples of translated code together with the build AST are included in Section 4.5.

4.2.1.1 Translation Process

The following subsections describe how particular nodes of an AST are translated.

4.2.1.1.1 node GlobalVars_decl

Many CAPL programs start with the section containing declaration of global variables. The node `GlobarVars_decl` covers these global variables. The code from this section is put inside the main method in a WWB script[4]. The pieces of code inside this section can be declarations (node `Declaration`), declarations of messages (node `Decl-MSG)` and comments (node `COMMENT`). The example of declaring global variables is shown in the example 4.5.1.

4.2.1.1.2 node Declaration

The node `Declaration` is translated by the function `generate_declaration`. The input for this function is a subtree of a particular declaration. Due to the reason that the variable type in WWB always starts with an uppercase letter[5], the function first looks for the variable type in the tree's leaves, and subsequently changes the first letter to the uppercase. After that, the function looks at the tree's children. Unless there is a subtree of another node found, the declaration is generated. Moreover, multiple declarations may be written in CAPL per one line. Then each of the declarations is translated separately. For example:

```
// CAPL
        int x, y, z;
```

```
' WWB '
        Dim x as Int
        Dim y as Int
        Dim z as Int
```

On the other hand, a subtree of the following node types may be found (as explained in Section 4.1.2.2.1):

`Array`, `Assign`, `Assign_Array`

An array must be declared with some dimensions. Therefore the function `generate_array` is used in order to separate array brackets and obtain the numbers of relevant dimensions. The function `generate_array` has the following input parameters:

variable name, array brackets (e.g. `[10][10]`), variable type

The declared array may look as follows:

```
// CAPL
        int x[10][10];
```

```
' WWB '
        Dim x(10,10) as Int
```

[4]i.e. right after `Sub Main`

[5]WWB is not case-sensitive though, thus the types do not have to start with an uppercase letter

When the node has the type `Assign`, the assignment is generated after a line of the declaration (i.e. as if no assignment would follow) has been put on the previous line. The function `generate_code` is called recursively with the relevant subtree as the input parameter. The generation of an assignment is described in Section 4.2.1.1.3.

The node `Assign_Array` works on the same principle. It first generates code for the declaration of an array, then the assignment of values into array's positions. The assignment into the array's positions is described in Section 4.2.1.1.4. Declarations with an assignment can be seen in the examples 4.5.1 and 4.5.2. The example 4.5.3 contains a simple declaration. The example 4.5.4 shows the declaration of arrays.

4.2.1.1.3 Node Assign

The function `generate_assignment` is called after the node `Assign` has been encountered. Most importantly, the leaves of the subtree belonging to this node contain either values (i.e. number or characters) or nodes that are assigned to a variable. The node that can be assigned may be either of the type `Array`, `Expression` or `Logic_EXPR`. The node `Array` means assigning a value from an array to a variable. For example:

```
// CAPL
x = arr[i][j];
```

```
' WWB '
x = arr(i,j)
```

If the node `Expression` is assigned to a variable, the translator continues again recursively to the function `generate_code` with the relevant subtree as the input parameter. It should be clear from Section 4.1.2.2 that a call of a function is one of the subtrees in the node `Expression`. The generation of expressions is described in Section 4.2.1.1.5. The function generation is then described in Section 4.2.1.1.6. Assignments of simple variables is shown in examples 4.5.1 and 4.5.2. The example 4.5.3 shows the assignment of an expression, whereas the example 4.5.5 shows the assignment of a CAPL function.

4.2.1.1.4 Node Assign_Array

WinWrap Basic do not enable assigning all values into array at one step as in CAPL. Therefore the values need to be assigned step by step. That means, one assignment per one line of the generated code. For example:

```
// CAPL
int x[2][2] = { {1,10}, {99,2} }
```

```
' WWB '
Dim x(2,2) as Int
x(0,0) = 1
x(0,1) = 10
x(1,0) = 99
x(1,1) = 2
```

An example of the node `Assign_Array` can be seen in the example 4.5.4.

4.2.1.1.5 Nodes Expression & Logic_EXPR

During the translation of an expression, an operator is found beforehand. The operator may be both unary or binary. The operator is the first child of the subtree belonging to the node `Expression`. The other children of the subtree are then literals, variables, or other expressions located in the particular expression. Unless one of the children is a function, then the expression consisting of these children and the leaf is generated. If the expression consists of a function, the function `generate_function` (described in Section 4.2.1.1.6) is called in order to generate the code. Even though parsing of an expression consisting of other expressions (in other words more than binary expression) is possible, the translation currently works only for the binary expressions. More complicated translation currently will not cause the translator to crash, but will produce an inappropriate output. Therefore a user should keep this in mind when writing more complicated expressions. They should be split apart on separate lines. Simple expressions inside control statements are shown in the examples 4.5.2 and 4.5.3. The example 4.5.6 shows more complicated expressions.

A logical expression binds together two unary or binary expressions by a logical operator, AND or OR. As in the case of node `Expression`, an operator is the leaf of the subtree belonging to the node `Logic_EXPR`. An example of a logical expression is shown in the example 4.5.6.

4.2.1.1.6 Nodes Function_UD & CAPL_fcn

The function `generate_function` is called with the relevant subtree of nodes `Function_UD` or `CAPL_fcn` as the input parameter. When any occurrence of a function is translated from CAPL into WWB, it may or may not include a type of a function. Naturally, a function has no type when a piece of code that calls a particular function occurs. That is, a function in an assignment or an expression would contain no type.

On the other hand, the declaration of a function as the procedure must include its type. The type can be found as the first child of the subtree. If a function is of the type `void`, or in other words has no return type, it is generated as an according WWB procedure called `Sub`. For other cases a function is generated as follows[6]

```
// CAPL
int function1(double x) { ... }
```

```
' WWB '
Function Function1(x as Double) as Int
...
End Function
```

The example 4.5.4 shows how the node `Function_UD` is translated.

The generation of node `CAPL_fcn` is specific. Under this type, there are certain functions that have corresponding equivalents in libraries used in PROVEtech:TA. A name of a function is retrieved from the leaf of the subtree. Then different kind of translation

[6]function's first letter is made upperletter according to WWB

is carried out based on the function's name. For illustration, few frequently used functions in rest-bus simulations were chosen. Other functions can be easily implemented. However, many functions supported by Vector CANoe (i.e. CAPL language) are not supported by PROVEtech:TA and their translation is outside the scope of this project. The selected functions and their description can be found below:

- CAPL syntax: `getSignal(signalName)`
  WWB syntax: `System.GetSignal signalName`
  description: gets the value of a signal
  parameters: `signalName` – the name of a signal to be polled

- CAPL syntax: `getFirstCANdbName(buffer, size)`
  WWB syntax: `System.GetDatabase buffer`
  description: finds out the name of the first assigned database parameters: `buffer` – buffer in which the database name is written, `size` – size of the buffer (not used in WWB)

- CAPL syntax: `ILSetSignal(signalName, value)`
  WWB syntax: `System.SetSignal signalName`
  description: sets the transferred signal to the provided physical value parameters: `signalName` – the name of a signal to be set, `value` – the physical value to which the signal should be set

- CAPL syntax: `output(messageName)`
  WWB syntax: `Send messageName`
  description: outputs a message from the program block; for WWB, it must be checked whether Channel, ID and Data were initialized parameters: `messageName` – the name of a message to be sent

- CAPL syntax: `write(string), ...`
  WWB syntax: `Debug.Print " ... "`
  description: outputs a text to the console
  parameters: `string` – string to be output; CAPL uses C-style output such as `"%d"` for the output of numbers, WWB connects pieces of string simply by `+` sign[7]

The examples 4.5.5 and 4.5.6 contain translations of nodes `CAPL_fcn`.

4.2.1.1.7 Control Statements

All control statements[8] except the `For` and `Switch` statements are translated in the same manner. A control keyword is placed at the beginning of a section. The leaf

[7]e.g. "hello World = "+x
[8]`If`, `If-Else`, `While`, `Do-While`

of the relevant subtree contains a control expression which is then recursively generated by calling the function `generate_code`. The children of the subtree represent the code inside a compound statement. This code is again generated by the function `generate_code`. The control statement ends with a keyword belonging to the particular control statement, e.g. `If ... End If`.

The `Switch` is translated in a similar way as statements described above. However, the children of the subtree contain several cases that may be selected during switching. Due to this fact, the converter walks iteratively through the children. For every child, the code belonging to the particular case is then generated.

Since the `For` statement is far different in CAPL and WWB, only the translation of simple For loops has been implemented. The word simple means that For loop written in CAPL should include the number till which the loop should be iterated and the size of iteration step. The code inside the statement cannot be generated if this condition is not satisfied. Example of the `For` statement may look as follows:

```
// CAPL
for(i = 1; i < 10; i++) { ... }
```

```
' WWB '
For i = 1 to 10 Step 1
        ...
Next i
```

Considering the jump statements, only `Return` is translated because WWB does not include any other jump functionality. Other jump statements will not be translated. The examples of control statements are shown in the examples 4.5.2 and 4.5.3.

4.2.1.1.8 node CAPL_event

As described in Section 4.4.3, CAPL event triggers the code inside the compound statement after a specific `on` event. When such an event is found, the keyword `on` is separated and the converter looks for the name of the event. After the name is found, the procedure is generated as if a function has been spot in the CAPL code. The name of an event only effects the name of the particular `Sub`. If one of the following is found – `on key`, `on message`, `on timer`, the name of a belonging key, message or timer appears in the name of `Sub`. Examples of the generated code may look as follows:

```
// CAPL
on Start { ... }
on message msg { ... }
```

```
' WWB '
Sub On_Start
        ...
End Sub
Sub On_message_msg
        ...
End Sub
```

How the node `CAPL_event` is translated, is shown in the example 4.5.6.

4.2.2 Translation to C

Translation to the C language is carried out in the same manner as the translation to WWB (described in Section 4.2.1). Due to the fact that CAPL has almost the same syntax and semantics as C, whole translation process is not described. However, CAPL includes a feature of reacting on a received message. This event is called `on message`. When the particular message is received, the relevant procedure is started. The main focus of this section lays in introducing a feature of translating the reaction on received messages to the C language. The SocketCAN framework, together with the libev library are used for this purpose. SocketCAN is the framework for CAN bus under Linux. Is has been designed to allow socket communication similarly as possible to the TCP/IP protocols [14]. The methods used for communication on the bus are described in Section 4.2.2.1. The `libev` is a high-performance event loop with many features [15]. The most important feature for this project are I/O watchers. The concept of implementing this library which reacts to a received message and triggers a relevant event is described in Section 4.2.2.2.

From the programming point of view, the CAPL and SocketCAN/libev reaction on messages has showed not to be the one-to-one equivalent. CAPL has an assigned database with IDs and names of all messages. The `on message` simply waits for a particular name of message and triggers the relevant procedure. By using the SocketCAN however, it is, first of all, checked whether the IDs match the IDs from the database. In case an exact equivalent is needed, the database with message names would probably need to be linked to the C program. From the rest-bus simulation point of view, PROVEtech:TA and Vector CANoe use the actual drivers by vendors' of CAN to USB converters to connect to the Windows operating system. The usage of SocketCAN however, seems as an interesting alternative for devices with the Embedded Linux platform. The drawback is that the usage of SocketCAN and libev represents solution only for the CAN bus. Commercial software tools most frequently contain possibility of creating a rest-bus simulation even for other buses such as LIN or FlexRay.

4.2.2.1 Usage of SocketCAN

This section describes the file `socketCan.c` implementing the communication on the CAN network. The whole while can be found in the attached CD, or online at `github.com/mikulleo/RestbusSim-Converter`.

Likewise in implementing the TCP/IP communication, a socket needs to be opened first for communicating over the CAN network. This is done in the function `open_port`. This function takes a port name as the input parameter. The socket is opened by the function `socket`. After that, the port name is copied to the interface structure by calling `strcpy`, and a file descriptor is set to be non-blocking by `fcntl`. To determine the interface index, the function `ioctl` is used. The relevant data are then assigned to `sockaddr_can`, the structure for CAN. Finally, a socket is bound to the CAN interface by calling the function `bind`.

The function `read_port` handles received frames on the network. The input parameter of this function is a file descriptor. Reading CAN frames for a bound CAN socket consists of reading the structure `can_frame`. To read a received frame, the function `read` is called. On success, the number of read bytes is returned. The function `read_port` returns the ID of a received frame. The idea here is to process this ID[9], and based on a recognized message further call a right handler by a `libev` library callback.

To write CAN frames on a socket bound to the CAN interface, the function `send_frame` is used. This function takes a file descriptor and structure of a CAN frame as the input parameters. The `write` call is located inside this function to send frames over the network. The number of sent bytes is returned in case of success.

4.2.2.2 Usage of libev

The concept of using the `libev` library for creating events analogous to CAPL's `on message` events is introduced. The code is included in the file `msgEvents.c`, and can be found at Attachments. This file includes a code prepared for automatic filling of the relevant `on message` events. Moreover, an example of shifting to car's reverse gear has been created in order to demonstrate the functionality of `libev`. This example is described in Section 4.2.2.2.1.

There are two watchers. First, a timer watcher is initialized by calling `ev_timer_init`, and started by `ev_timer_start`. A relevant callback is called every time the timer expires. Another watcher is an I/O watcher. This watcher is initialized and started by calling `ev_io_init` and `ev_io_start`. The I/O checks whether reading would not block the process. Hence, when a file descriptor becomes readable the `recvmsg_cb` is called. An ID of a received message is obtained by calling `read_port`. Furthermore, the switch statement selects and triggers the procedure relevant to the received message.

This switch statement is automatically generated during the translation process from CAPL to C. Nevertheless, function `convert_hexToID` is called beforehand. This function has been created manually in order to assign correct IDs to messages used during a simulation. When the `on_message` event is found in the CAPL code, the switch statement is created with a relevant case. Nevertheless, if the switch has already been created, only another case is added. The possible improvement of creating switch statements would be to pre-process a generated AST, write down all on message events, and then generate the switch statement. The relevant procedure for the received messages is included in an automatically translated code from CAPL to C. Therefore, this file must be included in the `msgEvents.c` file. To make it equivalent to the CAPL events, a database should be available, so the message name could be retrieved.

4.2.2.2.1 Example – Reverse Gear

The example can be found if the file `sent_rcv_libev.c`. First of all, a message is sent periodically over a CAN bus in this example. Consequently, messages are read back while arbitrating if the reverse gear has been set successfully based of received data. There are two watchers. First, a timer watcher triggers periodically a callback `sentmsg_cb`. This callback sends the data for setting the reverse gear by calling

[9]+ possibly compare it to messages' names in a database

send_frame. The function delay is called. It ensures that an I/O watcher is started after one second. This watcher calls a callback recvmsg_cb. This callback ensures that the data are read every specified period of time by calling read_port. Furthermore, the message is printed depending whether bits has been set correctly.

4.3 Graphics Conversion

The conversion of the whole graphical user interface (GUI) is a very complex problem. Graphical objects in Vector CANoe differ from those in PROVEtech:TA. Moreover, many created rest-bus simulations use custom objects and bitmaps. During this project, the conversion of simple objects used for the particular rest-bus simulation created in this project has been developed. CANoe uses the file format XVP for storing the GUI. Every created panel is located in a separate file. PROVEtech:TA uses AOF files. An AOF file consists of several tags inside brackets. These tags specify properties of individual pages. The whole GUI is located in one file, but workspace pages are separated by the tag [PageX], where X is the page number. The format XVP is an XML-based file format. So, EXtensible Stylesheet Language Transormation (XSLT) together with XML Path Language (XPath) can be used for the conversion. To be more precise, lxml toolkit is used. It is the Pythonic binding for the C libraries libxml2 and libxslt.

4.3.1 Implementation

Since several XVP files should be converted into one AOF, the GUI of created converter enables selecting multiple input XVP files which will be converted into one AOF file. Thus, XVP files need to be iteratively processed and converted into one single file. XSLT contains the wrapper type ElementTree. The ElementTree enables loading of an XML file as set of Element objects. These objects are designed to store hierarchical data structures such as XML. After a file has been selected, the transformation is run on XML code in order to get the ElementTree by calling the function etree.parse, where the file path is the only input parameter.

After that, the function tree.xpath is used several times during the conversion process. This function performs a XPath query for nodes of the ElementTree. First of all number of objects is found by counting number of Object tags. The ID is assigned to every object when iteratively exploring every object's properties.

Inside the Object tag, only its type is important. The type must be found for every object. For example:

```
<Object Type="Vector.CANalyzer.Panels.Design.CheckBoxControl, ... "
    >
```

Obviously, the whole type property must be split in order to separate the actual type, such as CheckBoxControl. In this project objects CheckBoxControl and ButttonControl are used. Nonetheless, the enum type in the code is prepared for the conversion of other CAPL default controls. Afterwards, `Property` tags are processed. Some properties' meaning is obvious. Those are `Name`, `Size`, and `Location`. The tag `Text` includes the text that is displayed on the control. The very important property is `SymbolConfiguration`. This property includes several values separated by the semicolon. First two numbers are not important for the conversion at all. After them, the values go in following order – a database name[10]; a node name[11]; a message name; a signal name; default value of the signal; name of a DBC file. There are two lists created:

- `controls_list` – stores the ID of an object, a belonging node, message, signal and DBC file
- `design_list` – stores the objects' position, size and label

These two lists are sufficient for the generation of an AOF file.

Objects of controls are prefixed by the text `BOOLCONTROL=Signalschlüssel:`. After this prefix, all properties of an object are written in one line separated by the semicolon. First, the whole signal "path" is listed, i.e. the direction together with the name of node, message and signal. During this project, the controls on the workspace deal only with sent signals during the rest-bus simulation. Therefore the direction `TX` is set implicitly. The example below shows a possible implementation:

```
BOOLCONTROL=Signalschlüssel:TX.Control.Control_Actuator.OnKey; ...
```

As next, the name, position and size are listed. The position and size need to be recalculated. CANoe considers the position and size values in pixels. On the other hand, PROVEtech:TA uses its own scale. One horizontal step in the position, as well as one piece of the button width equals 49 pixels. One vertical step and one piece of the button height equals 25 pixels. Due to this reason, the position and size must be recalculated before placing it in the AOF file.

For the position, the following calculation is performed:

$$x_{AOF} \doteq \frac{x_{XVP}}{h} - L$$
$$y_{AOF} \doteq \frac{y_{XVP}}{v},$$

where x_{AOF} and y_{AOF} are positions in the AOF file, x_{XVP} and y_{XVP} are positions in the XVP file, dividers h and v determine how large will be the horizontal and vertical space between individual controls, and L moves all controls to the left side. The constants h, v and L may be largely modified for other projects.

For the size, the width of a control is determined based on the width of its label. The height is then only converted from pixels to PROVEtech:TA scale. This project uses the following equations:

$$w_{AOF} \doteq \frac{len(name)}{5.5}$$
$$h_{AOF} \doteq \frac{h_{XVP} * 1.5}{25},$$

[10]CANdb, i.e. includes all messages and signal used during the rest-bus simulation

[11]one node equals one ECU

where w_{AOF} and h_{AOF} are sizes in the AOF file, h_{XVP} is the height in the XVP file, and $len(name)$ is the length of a label.

All the attributes after `YSize` in the AOF file for an object can be pre-set. These include, for example the color of a button, a text alignment, etc. As mentioned at the beginning of this section, the conversion of GUI is a very complex problem. The GUI conversion for this project has been successful, however it is not very neat. Nevertheless, it provides a user with the generated workspace for PROVEtech:TA which can be very easily modified. It is completely up to the user which changes will take place, but the modification will most probably include changing the button sizes or positions.

4.4 Usage

This section describes how the converter should be used. The Section 4.4.2 describes the general usage of the program. That is, what steps should be made after the program is started. The Section 4.4.3 describes how the CAPL code should be written in order to eliminate errors during parsing and translation to WWB.

4.4.1 Installation

The program is written in Python 3.4. It is necessary to have several Python modules available during a compilation. The modules can be added by using the `pip` installer. These modules are:

`io`, `string`, `mmap`, `os`, `tkinter`, `lex`, `yacc`

The `lex` and `yacc` modules are included in Ply package[12]. For running the tool, `tk-interApp.py` needs to be executed. After this file is executed, the GUI shows up. Subsequently, a user navigates himself through the user interface as described in the following section.

4.4.2 GUI

After the program is started, the window showed in Fig. 4.10a opens up. This window is used for writing all necessary values into tags inside the rest-bus XML configuration file for PROVEtech:RE. Without setting this XML file, the rest-bus simulation will not be functional. The windows includes the following lines:

- *P:RE config file – XML*: the absolute path of the PROVEtech:RE specific XML configuration file
- *DBC/ARXML file*: the absolute path of the DBC or ARXML file
- *RBS file – XML*: the absolute path of the so called RBS file where all messages and signals for the rest-bus simulation are defined[13]

[12]available at: `http://www.dabeaz.com/ply/ply-3.6.tar.gz`
[13]generated by RBSConfig tool

- *Port Name – CAN channel*: the name of a specific CAN channel used during the rest-bus simulation, e.g. `CAN1`
- *Bit Rate*: the bit rate for the CAN channel specified in *Port Name – CAN channel*

The bit rate used during this project is 250 000 kb/s because the bus is a high speed CAN. The example of all set values is in Fig. 4.10b. When the button *WRITE TO P:RE XML* is pressed, all values set in the fields mentioned above will be written to the file cited in *P:RE config file – XML*.

When the button *GUI & Code Conversion* is pressed, the window showed in Fig. 4.11a opens up. In this window, the conversion of either GUI or the CAPL code is triggered. There are two fields that need to be filled in (*Select* button may be used), specifically:

- *XVP file – GUI*: the absolute path of the XVP file(s), i.e. of the CANoe GUI
- *CAN file – CAPL*: the absolute path of the CAPL file, i.e. the code

Three buttons at the bottom serve as the initalizators of either the GUI or the CAPL code conversion. Whenever a particular button is pressed, the belonging action will follow. The example of a filled in window is in Fig. 4.11b. An output generated file is called `generatedWorkspace.aof`.

4.4.3 Test Language Restrictions

A user should be aware of certain properties of CAPL language that could make the parsing and consequent translation to WWB working improperly. First of all, the CAPL specific comments for every code section must be included. For example:

```
/*@@var: */
variables { ... }
/*@@end */

/*@@caplFunc:speedTest(float speed, float clock): */
float speedTest(float speed, float clock) { ... }
/*@@end */

/*@@msg:message1: */
on message message1 { ... }
/*@@end */
```

Next, even though the parsing of expressions consisting of several binary expressions works fine, the translation to WWB has been successfully created only for single binary expressions. Due to this reason, more complicated expressions should be split apart into the binary expression on separate lines. Following example shows both, the right and the wrong usages:

```
// WRONG:
x = y + z - 10;
```

```
// RIGHT:
x = y + z;
x -= 10;
```

The binary expressions may be bound together to a logical expression. If one of the expressions on the side is inside parentheses, then the second one must be placed inside the parentheses as well. Below there are two possible implementations:

```
if (x != 0 && y > 0) { ... }
if ((x != 0) && (y > 0)) { ... }
```

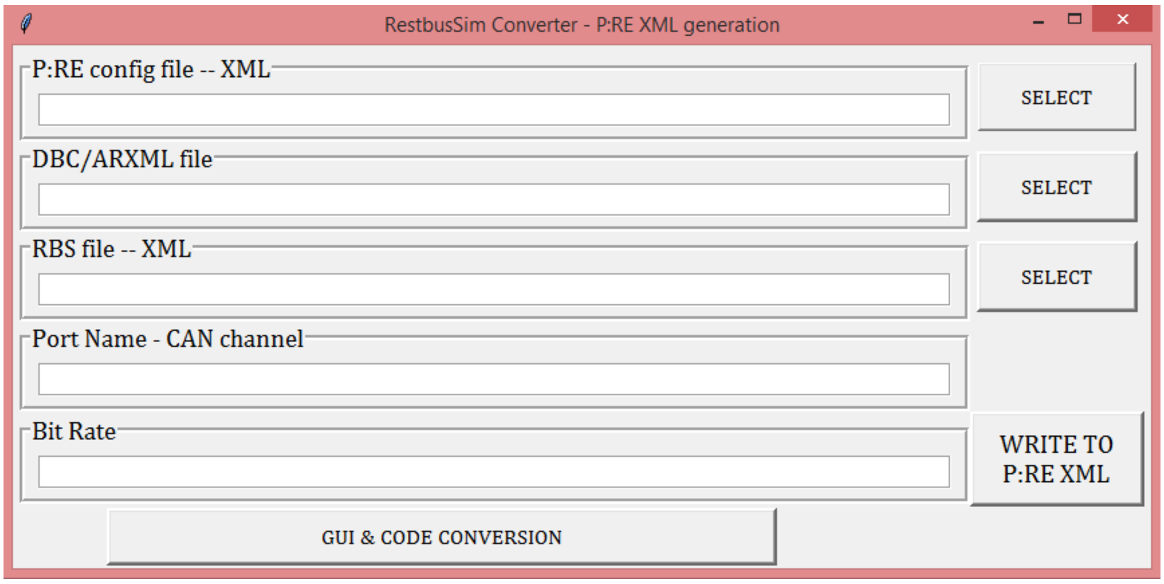
Since the implementation of for loops is different in CAPL from those in WWB, only the translation of simple for loops works. Under the word simple for loop is meant, that the initialization variable must be included, the value till which the loop will iterate, and the iteration step are included. The for loop can be implemented as follows:

```
for(int i; i < 10; i++) { ... }
```

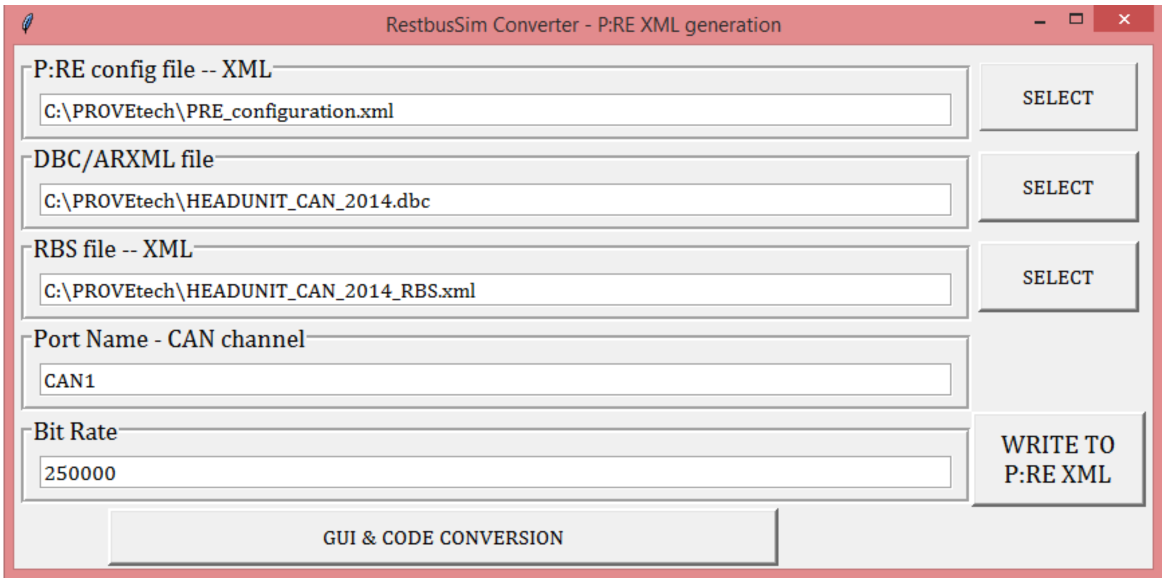
Moreover, when using a control statement, i.e. for, while, etc., the code must be placed inside the curly braces. The implementation of control statements with no curly braces is not supported.

Another unsupported feature include the possibility of an expression inside array brackets, such as `arr[++]`.

To avoid a crash during any translation process, unsupported translations do not raise an error. However, a user should be aware of these test language restrictions, and consequently make manual changes in a translated output file.



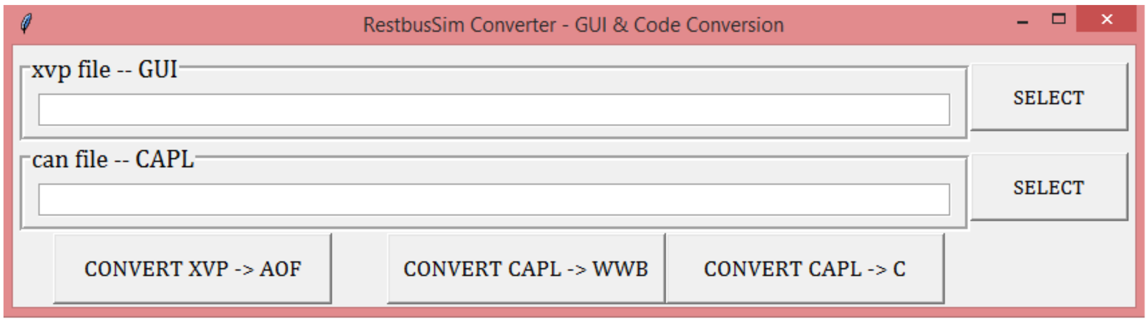
(a) blank



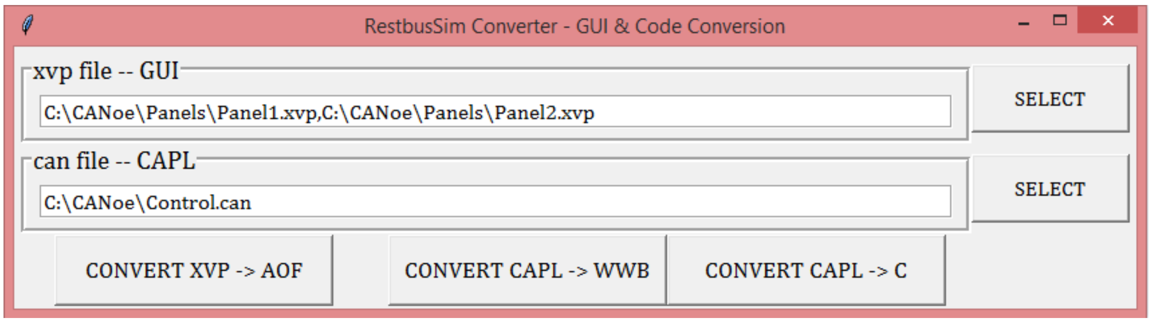
P:RE config file -- XML: C:\PROVEtech\PRE_configuration.xml
DBC/ARXML file: C:\PROVEtech\HEADUNIT_CAN_2014.dbc
RBS file -- XML: C:\PROVEtech\HEADUNIT_CAN_2014_RBS.xml
Port Name - CAN channel: CAN1
Bit Rate: 250000

(b) filled in

Figure 4.10: P:RE XML generation window



(a) blank



C:\CANoe\Panels\Panel1.xvp,C:\CANoe\Panels\Panel2.xvp

C:\CANoe\Control.can

(b) filled in

Figure 4.11: GUI & Code Conversion window

4.5 Examples

The example 4.5.1 shows the declaration of global variables. The message declaration can be seen there. The example 4.5.2 shows the declaration of the user-defined function. The processing of If statements is displayed here. The example 4.5.3 shows the same user-defined function, however with usage of For statement. Also the processing of an expression inside the For statement is shown. The example 4.5.4 shows the processing or an array declaration and the assignment to arrays. The example 4.5.5 shows the processing of the CAPL event `on envVar` with CAPL specific functions, and their translation to WWB equivalent. The example 4.5.6 shows the processing of the CAPL event `on start` followed by If statement containing expressions and function in the input parameter.

4.5.1 Example 1

```
/*@@var:*/
variables
{
        char letter_a = "a";
        int j, k = 2;
        message 0x101 msg;

        /* comment */
}
/*@@end */
```

```
Sub Main
        Dim letter_a As String
        letter_a = "a"
        Dim j As Integer
        Dim k As Integer
        k = 2
        Dim msg as New CanMsg
        msg.Id = &H101
        ' /* comment */ '
End Sub
```

```
generate_code(tree)
  -> generate_declaration(tree.child)              --- char letter_a = "a";
     generate_declaration(tree.child)
        -> generate_code(tree.child.child)
        -> generate_assignment(tree.child.child) --- int j, k = 2;
     generate_message_declaration(tree.child)     --- message 0x101 msg;
```

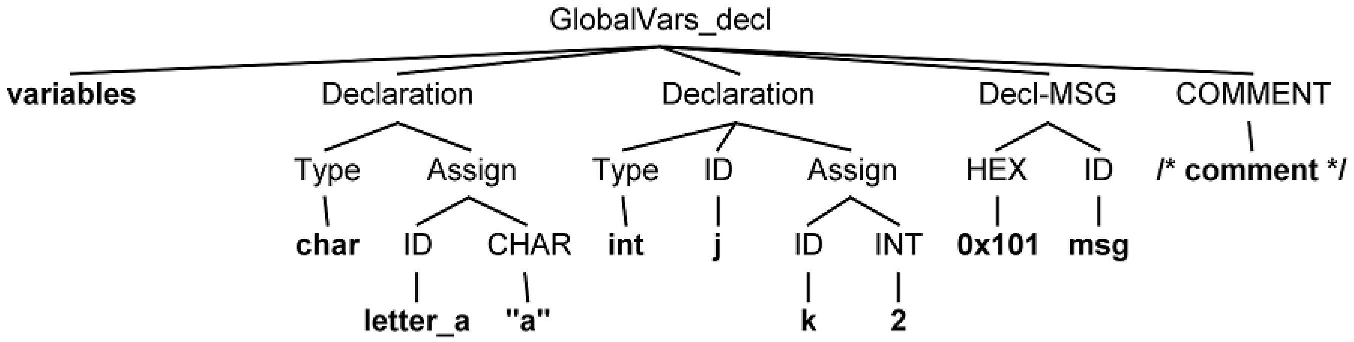


Figure 4.12: AST – global variables

4.5.2 Example 2

```
/*@@caplFunc:speedTest(float speed): */
float speedTest(float speed)
{
        float cruising_speed = 70;

        if (speed >= cruising_speed) {
                speed = cruising_speed;
        }
}
/*@@end */
```

```
Function SpeedTest(speed as Decimal)
        Dim cruising_speed As Decimal
        cruising_speed = 70

        If      speed >= cruising_speed Then
                speed = cruising_speed
        End If
End Function
```

```
generate_code(tree)
  -> generate_function(tree.child)                       --- float speedTest(float speed)
       -> generate_code(tree.child.child)
       -> generate_declaration(tree.child.child)
          -> generate_code(tree.child.child.child)
          -> generate_assignment(tree.child.child.child) --- float cruising_speed = 70;
          generate_code(tree.child.child)
            -> generate_code(tree.child.child.leaf)      --- speed >= cruising_speed
            -> generate_code(tree.child.child.child)     --- speed = cruising_speed
```

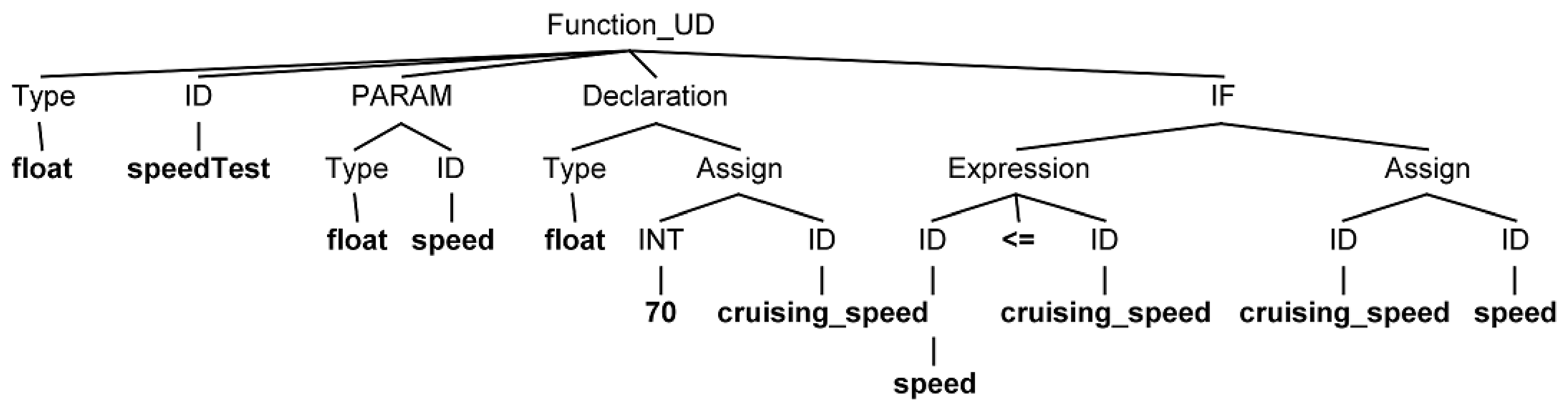


Figure 4.13: AST – the function, If statement

4.5.3 Example 3

```
/*@@caplFunc:speedTest(float speed): */
float speedTest(float speed)
{
        for(int i; i < 10; i++) {
                speed = speed++;
        }

        return speed;
}
/*@@end */
```

```
Function SpeedTest(speed as Decimal)
' If iteration variable not declared ---> declare by Dim! '
        For i = 0 To 9 Step 1
                speed = speed + 1
        Next i
        Return speed
End Function
```

```
generate_code(tree)
  -> generate_function(tree.child)                              --- float speedTest(float speed)
       -> generate_code(tree.child.child)                       --- for(int i; i < 10; i++)
            -> generate_code(tree.child.child.child)
            -> generate_assignment(tree.child.child.child)      --- speed =
               ->  generate_code(tree.child.child.child.leaf) --- speed = speed++;
           generate_code(tree.child.child)     return speed
```

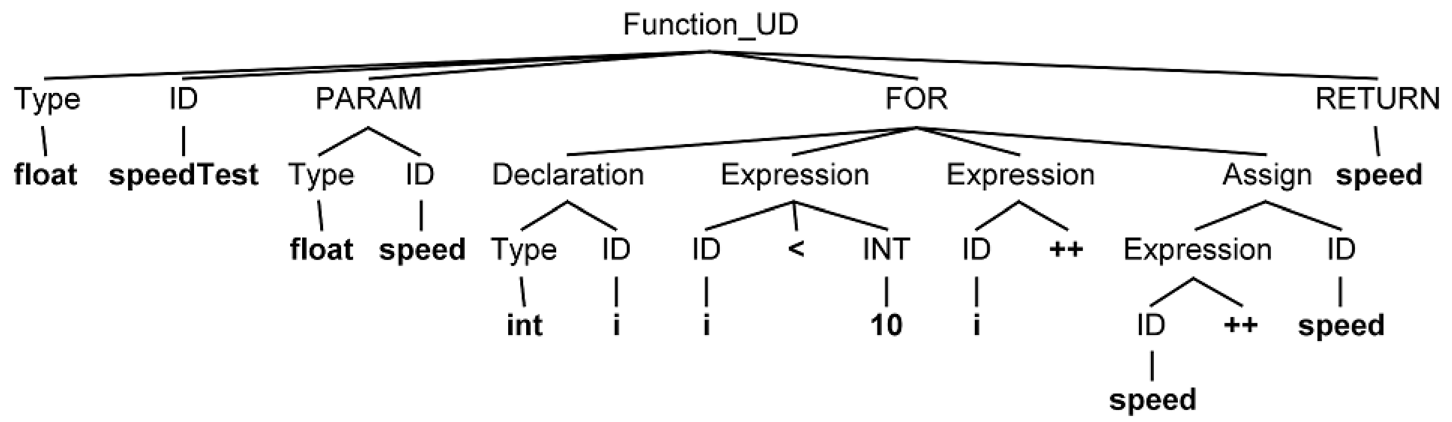


Figure 4.14: AST – function, For statement

4.5.4 Example 4

```
/*@@caplFunc:myFunc(): */
void myFunc()
{
        int sample_data[4] = {100,300,500,600};
        double M[2][2];
        M[x][x] = -3.14;
}
/*@@end */
```

```
Sub MyFunc()
        Dim sample_data(3) As Integer
        sample_data(0) = 100
        sample_data(1) = 300
        sample_data(2) = 500
        sample_data(3) = 600
        Dim M(1,1) As Double
        M(x,x) = -3.14
End Sub
```

```
generate_code(tree)
  -> generate_function(tree.child)                   --- void myFunc()
       -> generate_code(tree.child.child)
       -> generate_declaration(tree.child.child)     --- int sample_data[4] = {100,300,500,600};
            -> generate_array
          generate_declaration(tree.child.child)     --- double M[2][2];
            -> generate_array
          generate_assignment(tree.child.child)      --- M[x][x] = -3.14;
            -> generate_array
```

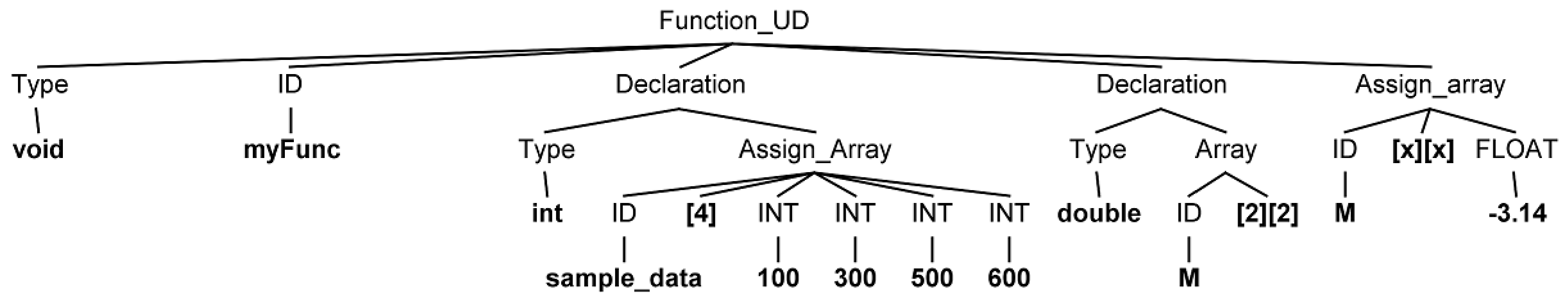


Figure 4.15: AST – function, arrays

4.5.5 Example 5

```
/*@@envVar:initialize:*/
on envVar initialize
{
    ILSetSignal( Ctrl_C_Stat1_AR::ReturnKey_Psd_UB , 1);
    x = getSignal(Ctrl_C_Stat1_AR::ReturnKey_Psd_UB);
}
/*@@end*/
```

```
Sub On_EnvVar
        System.SetSignal("TX.CTRL_C.Ctrl_C_Stat1_AR.ReturnKey_Psd_UB", 1)
        x = System.GetSignal("TX.CTRL_C.Ctrl_C_Stat1_AR.ReturnKey_Psd_UB")
End Sub
```

```
generate_code(tree)                                  --- on envVar initialize
  -> generate_function(tree.child)                   --- ILSetSignal( Ctrl_C_Stat1_AR::ReturnKey_Psd_UB ,
     1);
     generate_assignment(tree.child)
        -> generate_code(tree.child.leaf)
           generate_function(tree.child.leaf) --- x = getSignal(Ctrl_C_Stat1_AR::ReturnKey_Psd_UB);
```

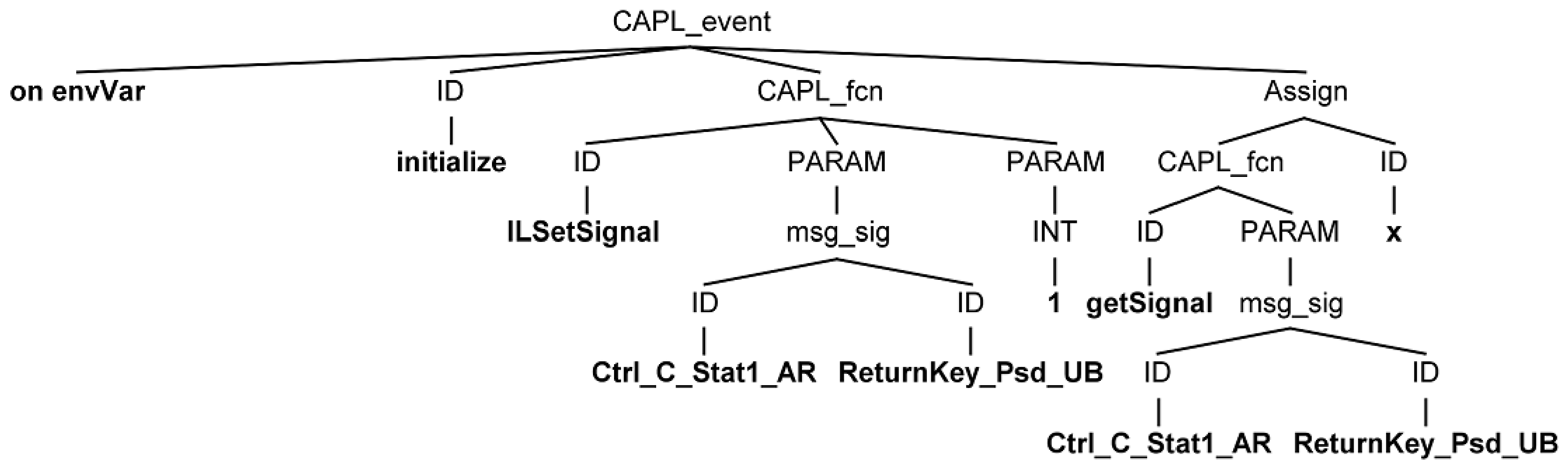


Figure 4.16: AST – CAPL event, specific functions

4.5.6 Example 6

```
/*@@startStart:start: */
on start
{
        if( readHandle != 0 && fileGetString(timeBuffer,elcount(timeBuffer),readHandle) != 0)
        {
                setTimer(cyclicTimer,100);
        }
        else
        {
                write("Data file cannot be opened for read access.");
        }
}
/*@@end */
```

```
Sub On_start()
```

```
        If      readHandle <> 0 AndAlso fileGetString(timeBuffer,elcount(timeBuffer),readHandle) <>
            0 Then
                setTimer(cyclicTimer,100)
        Else
                Debug.Print "Data file cannot be opened for read access."

        End If
End Sub
```

```
generate_code(tree)             on start
  -> generate_code(tree.child)
       -> generate_code(tree.child.leaf)                --- if ( ... && ... )
            -> generate_code(tree.child.leaf.child[0])  --- readHandle != 0
               generate_code(tree.child.leaf.child[1])
                 -> generate_function(tree.child.leaf.child[1]) --- fileGetString(timeBuffer, ...)
                    != 0
       -> generate_code(tree.child.child[0])            --- setTimer(cyclicTimer,100);
          generate_code(tree.child.child[1])            --- write("Data ... ");
```

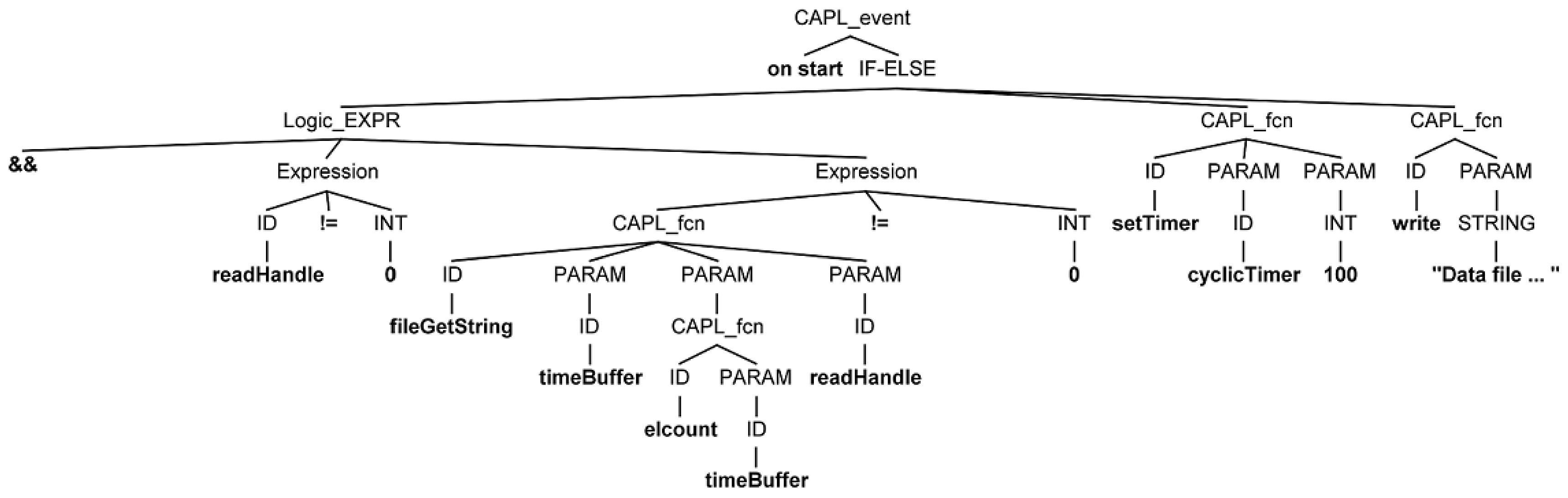


Figure 4.17: AST – CAPL event, logical expression

Chapter 5

Conclusion

At the early stages, basic testing of Vector CANoe and PROVEtech:TA by MBtech was carried out on already created rest-bus simulations in order to get familiar with these tools. During the next step, the rest-bus simulation for the head unit has been successfully made functional for both, CANoe and PROVEtech:TA. The rest-bus simulation has been created for both formats of configuration files, i.e. DBC and ARXML. After having created both rest-bus simulations successfully, a tool for converting the rest-bus simulation from CANoe to PROVEtech:TA was developed. The automated test scripts to the C language has been created as well.

The tool for converting the rest-bus simulation from CANoe to PROVEtech:TA has been developed successfully – it has been proved directly at the MBtech company. It allows the conversion of testing scripts from CAPL to WinWrap Basic as well the conversion of basic GUI. Additionally, it supports automatic configuration of several relevant settings in PROVEtech:TA specific configuration files. Because of the fact that WinWrap Basic together with PROVEtech:TA libraries is quite different from CAPL, the conversion of testing scripts is not one-to-one equivalent and so the translation of some pieces of code is not possible (e.g. complicated For loops, or timers). Nevertheless, the conversion works sufficiently for many test cases which are described in this thesis.

When translating automated test scripts from CAPL to WWB, it is assumed that a user works with a CAPL code that has already been compiled in a CAPL editor. The developed tool does not serve as a compiler. If a syntax error is detected during a translation process, the parser will raise an error and the program will crash. Furthermore, if a user uses a CAPL input file that includes unsupported syntax by the translator, the program will finish the translation. An output file must be manually modified because it will contain translation errors as the consequence. The enhanced functionality of translating unsupported syntax is planned to be added in future versions.

The feature of translating a CAPL code to the C language has been proposed as a concept. This functionality could not be verified for the head unit used during this thesis for several reasons as already mentioned in the previous text. The main reason was that the ECU was not available for usage outside the department at the MBtech company, and the impossibility to use own laptop (i.e. laptop with Linux platform) inside the department. Nonetheless, the proposed concept may be used as a basis for further development. Nevertheless, translation of automated test scripts to C works for many cases.

To summarize, the head unit can be controlled from CANoe and PROVEtech:TA by using the simulation in the same manner as there were control buttons physically available. Furthermore, the conversion process from CANoe to PROVEtech:TA works for all test cases required by MBtech.

The usage of open-source software tools has not shown to speed up the process of creating particular rest-bus simulations. Due to this reason, the open-source tools were not used even though the overview is given in the document. The open-source tools may serve mostly for additional analyses of the configuration files.

Appendix A

Content of the Attached CD

- The thesis in .pdf format
- The source of the developed tool with a functional example

Bibliography

[1] *PROVEtech:TA – Operating Instructions*, 2nd ed., MBtech Group, Sindelfingen, Germany, 2014.

[2] Vector CANoe – product information. [Online]. Available: http://www.vector.com/pi_canoe_en

[3] QTronic – Test Weaver. [Online]. Available: http://www.qtronic.com/en/weaver.html

[4] MaTeLo – model-based testing. [Online]. Available: http://www.all4tec.net/Matelo/model-based-testing.html

[5] S. Bender, V. Pannirsilvam, R. Khoo, P. L. Hidalgo, M. Tschochner, P. Sheth, S. Osswald, D. Gleyzes, H. W. Ng, and M. Lienkamp, "Concept of an electric taxi for tropical megacities," in *Proceedings 3rd Conference on Future Automotive Technology*, 2014.

[6] Provetech:TA – test automation. [Online]. Available: https://www.mbtech-group.com/eu-en/electronics_solutions/tools_equipment/provetechta_test_automation

[7] *DBC File Format Documentation*, 1st ed., Vector Informatik, Feb. 2007.

[8] *Model Persistence Rules for XML*, 2nd ed., AUTOSAR, Nov. 2011.

[9] *PROVEtech:RE – Manual*, 2nd ed., MBtech Group, Sindelfingen, Germany, 2014.

[10] *WinWrap Basic Language*, online help documentation, Polar Engineering, 2014. [Online]. Available: https://www.winwrap.com/web2/basic#!/ref/WWB-doc0001.htm

[11] D. M. Beazley. Ply (python lex-yacc). [Online]. Available: http://www.dabeaz.com/ply/ply.html

[12] *Programming with CAPL*, Vector CANtech, Novi, MI, USA, Dec. 2004.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, 2nd ed. Boston, MA, USA: Pearson Education, 2007.

[14] Readme file for the Controller Area Network Protocol Family (aka SocketCAN). [Online]. Available: https://www.kernel.org/doc/Documentation/networking/can.txt

[15] libev - a high performance full-featured event loop written in C. [Online]. Available: http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod

[16] *CAPL Functions*, 1st ed., Vector Informatik, Stuttgart, Germany, 2012.

[17] *CANoe – Manual*, 7th ed., Vector Informatik, Stuttgart, Germany, 2010.

[18] O. Kulatý, "Message authentication for can bus and autosar software architecture," Master's Thesis, Czech Technical University in Prague, Feb. 2015.

[19] M. Johnson, "Intermediate representation," Stanford University, Handout, Jul. 2008.