MŰEGYETEM 1782

SZAKDOLGOZAT-FELADAT

Sándi Tamás (SMNQCY)
szigorló villamosmérnök hallgató részére

Script alapú járműbusz-szimuláció

Napjaink prémium kategóriás személyautóinak működésében közel száz elektronikus vezérlőegység (ECU) játszik szerepet. Ezek között a kommunikáció szabványos autóipari protokollokon zajlik (CAN, FlexRay, LIN). Az így kapott összetett elosztott rendszer fejlesztésének elengedhetetlen részét képezi a vezérlőegységek kimerítő tesztelése. A tesztelés során a tesztelendő alkatrész által elvárt *kommunikációs jeleket szimuláljuk*. A tesztszekvenciák előállítására rendelkezésre áll a vállalatnál egy *saját fejlesztésű kommunikációs eszköz* (Gateway). Az eszköz számos szolgáltatást nyújt a felhasználó felé (CAN, FlexRay és LIN kommunikációs csatornák, valamint GPIO és A/D portok).

A hallgató feladata egy olyan *Eclipse alapú szoftvereszköz* elkészítése, melynek segítségével a felhasználó egy erre a célra definiált *szkriptnyelvben* fogalmazhatja meg a Gateway által szimulálandó bonyolult kommunikációs szekvenciákat. Az eszköznek képesnek kell lennie továbbá az elkészített szkript alapján a *Gatewayen futtatható kód generálására,* valamint a hardverre való letöltése után annak *futtatására* is. A hallgató feladatának a következőkre kell kiterjednie:

- *Gyűjtse össze a gyakori használati eseteket* (use case analízis), majd ezek alapján *definiáljon egy kényelmesen használható domainspecifikus nyelvet* a felmerülő igények kielégítésére. A szkriptnek lehetőséget kell nyújtania eseményvezérelt szekvenciák megfogalmazására is (pl.: üzenetek megérkezésére, digitális vagy analóg bemenetek megváltozására való reagálás). A nyelv definiálására, valamint a későbbiek során annak feldolgozására az *Eclipse* alapú *Xtext keretrendszert* használja. Adja meg a definiált nyelv *meta-modelljét* is.
- *Vizsgálja meg* a szkript által megfogalmazott szekvenciák Gateway-en való *végrehajtásának lehetséges megoldási módjait*: (i) a szkript alapján egy köztes bájtkód generálása, majd a hardveren való végrehajtása egy értelmező segítségével, (ii) a szkript alapján C/C++ kód generálása, lefordítása, majd letöltése a hardverre. *Válassza ki* ezek, valamint az egyéb felmerülő megoldási javaslatok közül *az optimálist* (memóriaigény, futásidő, időzítési pontosság).
- *Tervezze meg* és *implementálja* azt a *beágyazott* (C/C++ nyelvű) *keretrendszert*, mely képes a választott megoldást a Gateway alacsonyszintű *drivereire támaszkodva* támogatni, valamint *készítse el* a választott megoldást megvalósító *kódgenerátort* is. *Igazolja a helyes működést* megfelelő *tesztszekvenciákkal*.

Tanszéki konzulens: Dr. Sujbert László, docens
Külső konzulens: Faragó Dániel (ThyssenKrupp Presta Hungary Kft.)

Budapest, 2017. március 10.

……………………
Dr. Dabóczi Tamás
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Sándi Tamás

SCRIPT BASED VEHICLE BUS SIMULATION

BELSŐ KONZULENS

Dr. Sujbert László

BME MIT

KÜLSŐ KONZULENS

Faragó Dániel

ThyssenKrupp Presta Hungary kft.

BUDAPEST, 2017

Contents

HALLGATÓI NYILATKOZAT

Alulírott **Sándi Tamás**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. 05. 17.

...…………………………………………….
Sándi Tamás

Összefoglaló

Az autóipar napjaink egyik leggyorsabban fejlődő iparága. Az iparág növekedésével az autóipari teknológiák robbanásszerű fejlődésnek indultak. Az elektronikai eszközök népszerüsödésével, elkerülhetetlen, hogy az elektronikai komponensek egyre nagyobb szerepet játsszanak az újabb és újabb autók tervezésénél.

A mai okos autókban számtalan biztonsági és kényelmi funkció van. Egyre több mechanikai komponenst cserélnek le az elektronikai megfelelőjükkel. Ezen funkciókat elektronikus vezérlőegységek (Electronic control unit – ECU) biztosítják az autó különböző pontjaiban. Ezek az egységek szabványos autóipari kommunikációs protokollokon (tipikusan CAN, FlexRay, Lin és Ethernet protokollokon) keresztül kommunikálnak egymással, egy elosztott hálózati rendszert megvalósítva ezzel.

A vezérlőegységek komplexitásának növekedésével, egyre gyakoribbá válik, hogy az egy hálózatba kerülő ECU-kat egymástól függetlenül fejlesztik. Azonban a vezérlőegységek fejlesztéséhez szükség van a környezetükkel való kommunikációhoz. Ezt gyakran úgynevezett *Restbus simulation*-nel biztosítják az ECU számára. A módszer lényege, hogy egy külső teszt eszköz segítségével emulálják a hiányzó kommunikációs hálózat viselkedését.

A feladatom egy olyan szkript nyelv megtervezése és implementálása amelynek segítségével effektíven és könnyedén lehet Restbus-hoz hasonló teszt eseteket leírni. A nyelvnek biztosítania kell a kommunikációt vezérlőegységekkel CAN-en és FlexRay-en keresztül. A nyelvnek képesnek kell lennie komplex kommunikációs viselkedést leírnia magas szintű szkript parancsok segítségével, ezzel elrejtve a teszt mérnök elöl az implementációs részleteket.

A végső célom egy olyan asztali alkalmazás létrehozása, ami segíti a felhasználót szkriptek létrehozásában, fordításában és letöltésében a teszt hardverre.

Abstract

The automotive industry is among the fastest growing industries nowadays. Owing to the huge amount of capital being invested in the industry annually, automotive technologies are being developed at an accelerated rate. With the exploding growth of electronics, it is inevitable that electric components play an increasingly critical role in the development of new cars.

Smart cars today have a multitude of safety and comfort functions, often replacing mechanical parts with their corresponding electric versions. Functionality of these features are centered in separate electronic control units (ECU) throughout the car. These units communicate with each other through standard automotive network communication protocols (typically CAN, FlexRay, LIN and Ethernet), realizing a distributed control system.

As the complexity of individual ECUs increase, it is more economical and practical to design them independently of each other (by different development teams or even companies). These components demand substantial interaction with its environment during development phase. Said interaction is frequently provided by Restbus (remaining bus) simulation, where a designated testing unit emulates the required behaviour of the ECUs' environment.

My task is the design and implementation of a flexible script language with the intention to efficiently describe and create extensive test cases determined by use case analysis. The language should provide interfaces for CAN and FlexRay communications for ECU testing. The main benefit is the ability to emulate complex communication behaviours through high level script commands, hiding software specific details from the test engineer.

My ultimate objective is the creation of a desktop application, to faciliate the creation and downloading of individual scripts in a custom-made IDE[1].

[1] Integrated Development Environment

1 Introduction

In the following chapter, I am going to give a comprehensive description about the different aspects of my thesis.

1.1 Significance and motivation

The work described in this thesis was done at ThyssenKrupp Presta Hungary kft. The company is a technology leader in the field of steering systems and a major innovative partner of the automotive industry.

My task is to create a user friendly script language with IDE support, capable of describing required communication behaviours (e.g. restbus simulation tests) to help ECU development at the company. The language will be run on our so-called Gateway hardware. The Gateway has an existing software structure operating on an RTOS and offers advanced drivers to communicate through CAN and FlexRay protocols. The software offers GPIO functionalities as well. It comes with a desktop API implemented in the Java programming language, making it possible to configure and use it from a desktop environment.

The language shall be designed to offer ways to effectively and concisely describe communication behaviours required for testing. A code generator module parses and transforms the target script into an embedded software configuration in the form of C++ source code. This is compiled and downloaded to use with the Gateway's base software framework. The Gateway acts as an independent testing unit after flashing and requires no configuration or connection to a desktop computer. The hardware can then be put directly into a car or used in a continuous environment (where tests run autonomously for long periods of time).

The language takes inspiration from the CAPL language used by Vector tools. CAPL is a script language created to configure hardwares supplied by Vector [8]. Since there are a limited amount of vector tools at the company, the GatewayScript is aiming to replace the role Vector tools play at testing departments.

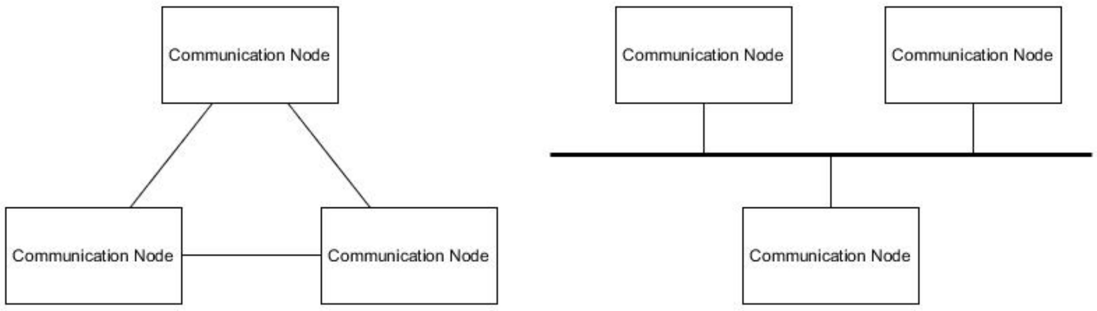
1.2 Communication protocols in the automotive industry

Since communication will be a central role of the language features, I will give a brief description about the communication protocols used in the industry today.

1.2.1 Overview

As the role of electronic components grows at an accelerated rate in todays cars as does their heavy reliance grow on intercomponent communications. In the earlier years of electronization data exchange between components were accomplished with dedicated cables between desired ECUs. This method of cable infrastructure is called point to point communication.

As the increased amount of information that had to be exchanged grew, the cabling necessary to faciliate point to point communication became unacceptably large and complicated. The solution that ultimately completely replaced point to point communication was the use of so-called communication buses. A bus is a single communication route that has multiple components connected to it, thus information traveling on the bus is shared between all participants.



1-1. Image - Point to Point versus Bus communication

Characteristics that are present and are subjects for comparison between all bus based protocols:

Information filtering

Since data traveling along the bus is visible to all participants, it is necessary to establish some kind of system that helps each individual component separate the relevant and the useless information. This is normally accomplished by encoding extra addressing information into the data. We differentiate between sender-selective and reciever-selective addressing. When using sender-selective addressing the sender specifies the recipients of the message. The more conventional strategy – reciever-selective – brands each message with a unique identifier based on the information it carries. Recievers can then individually decide if they want to process the given message based on the ID.

Data protection

Ensuring the authenticity of information is crucial during communication, especially in safety critical systems. Factors that influence data safety can be reflections that arise at the end of cables and electromagnetic noises caused by nearby electronic components (electric motors for example), that can result in capacitive and inductive coupling. Increased protection can be gained with shielding, or by using differential signal transmission. This is done through unshielded twisted pair cables (UTP) where information is encoded in the difference between voltage levels on the cables.

Bus access

All participants on a single bus share a singular input/output to the bus (meaning at any given time components write to the same resource, and all read operations result in the same data). This demands the presence of a rule set that controls data traffic along the bus, to prevent concurrent transmissions, and manage transmissions in general.

1.2.2 Control Area Network (CAN)

Development of the CAN protocol started in 1983 at Robert Bosch GmbH. It’s a protocol specially designed for automotive usage [4]. It was standardized by the ISO (International Standardization Organization) in 1993 and has seen several revisions since then. CAN remains the most used protocol to this day, and is mainly used in the drive and chassis areas for handling the operation of the car.

CAN uses an unshielded twisted two-wire (UTP) line as a physical medium for symmetrical signal transmission, and has bus termination resistors to prevent reflection. It has a maximum data rate of 1 Mbit/s. The dominant bus level (a typical differential voltage of 2V on the wires) corresponds to a logical „0”, and the recessive bus level (a differential voltage of 0V) corresponds to logical „1”. When different nodes attempt to write different logic values to the bus, the dominant bus level alway overrides the recessive level. This means the logic „0” always enjoys priority above the logic value „1”.

CAN is an event-driven protocol that uses receiver-selective addressing, meaning all nodes have access to the bus at all times. Messages are equipped with a 11 bit (29 bit in case of extended IDs) long identifier. CAN guarantees non destructive data transmission with the CSMA/CA (Carrier Sense Multiple Acces with Collision Avoidance) method. When simultaneous transmission occurs, all transmitting nodes begin transmitting the identifier field. When difference is detected between the bit written and read back from the bus by a node, it detects the collision and terminates transmission. Difference occurs when a node sends a dominant bit, whereas a recessive bit was sent by another. This is called bitwise bus arbitration, and it implicitly assigns a priority to messages by way of their identifiers (the smaller the ID is the higher its priority).

1.2.3 FlexRay

FlexRay was developed by the FlexRay consortium made up by several of the largest automotive manufacturers. It was designed to be used in safety- and time-critical applications, where it is necessary to be both faster and more reliable than what the CAN protocol is able to offer [3].

The protocol provides a redundant channel of transmission to increase safety. Both communication channels can operate with a maximum datarate of 10 Mbit/s. The second channel can be used as independent communication channel as an alternative to boost the datarate up to 20 Mbit/s.

FlexRay is a time-triggered communication architecture, meaning nodes are only allowed to transfer data in their dedicated time slots. A FlexRay communication is periodical, made up by a fixed number of communication cycles. Each cycle is made up by a static segment, network idle time (NIT) segment, and optionally a dynamic time segment and symbol window.

The NIT segment is upheld for synchronization of local clocks, no communication occurs during this time. Each cycle length is specified by its number of macroticks assigned to individual segments. Macroticks are composed of microticks, the smallest time unit of local clocks. Differences in hardware may result in microticks of different lengths, hence macroticks can be made up by a different number of microticks on different nodes.

The static segment is implemented with the TDMA (Time Division Multiple Access) method, assuring deterministic transmission of data. Static segments are organized into a number of static slots, with a maximum number of 1023 slots. Static messages can be assigned to the individual slots, which are transmitted by their nodes in their alloted time segment. A counter shared between all nodes is employed to keep track of the schedule. The counter’s value corresponds directly to the current slot ID availible for transmission. A minimum of two slots assigned to FlexRay nodes in the static segment must be upheld to generate the global time base.

Since the TDMA method is unsuited for sporadic and asynchronous communication, FlexRay allows for transmission of data in an event-driven manner, with the use of the dynamic segment. The dynamic segment is based on the FTDMA (Flexible Time Division Multiple Access) method. To preserve the determenistic nature of the static segment, the dynamic segment is always of the same length. It is made up by a fixed number of minislots, to which dynamic messages can be assigned to. Similarly to the static segment a separate shared counter is used for the dynamic segment. In the case a minislot has no message assigned to it, the counter is incremented without transmission. Cases may arise where not all messages can fit into a dynamic segment, in which case they get delayed a whole cycle. It is the responsibility of the system engineer to avoid such complications.

1.2.4 Standardization

It is imperative that common technology standards be set for developing independent products that will ultimately be part of the same distributed system. It also helps manufacturers set requirements that are familiar to developers and are easy to review. These technologies are described by ISO (International Standardization Organization) standards.

1.2.5 AUTOSAR

AUTOSAR (Automotive Open System Architecture) is a software architecture standard for developing ECU software. Its aim is to structure software functionalities into independent software components. Software components communicate with each other and with the hardware through standardized interfaces. Since the Gateway will be communicating with ECUs running with AUTOSAR based softwares, it is important to conform to the communication interface set by the standard. The detailed description about the rest of the architecture is outside the scope of this thesis.

Frame Structures

AUTOSAR defines how data is transmitted through communication protocols by using frame structures [1]. Software components in the application layer deal with external informations in the form of signals. Signal represent real-world information (e.g. temperature or speed). A transformation can be specified between the raw data carried by the signal and the application relevant information by assigning *computation methods* to signals. Linear compumethods have an offset and a factor, Text table compumethods assign strings to specific values and Rational function methods have a polynom to calculate the information. Some of the advantages of Compumethods are readability, and the ability to transfer non-integer values as integer values to avoid the need to deal with floating point numbers.

Groups of signals are wrapped into PDUs (protocol data units). Information in AUTOSAR travels through layers in the form of PDUs. PDUs in addition to its carried data, can contain protocol relevant information in its PCI segment (protocol control information). PDUs travel upward through layers by processing the PCI and forwarding the data segment (called service data segments, SDUs), and downwards by wrapping the SDUs with PCIs.

PDUs are transmitted between ECUs by assigning them to frames. Messages (the protocol a given message uses is irrelevant) transmit singular frames as their payloads. Frames abstract the transmitted data from the transmitting resource, the messages.

1.3 Task description

My work is separated into four individual parts. A brief description of each fundamental part is given below.

1.3.1 Research

First, it is necessary to have a solid background in language theory and language design to create a well structured language. The chosen technologies and frameworks for implementation have to be learnt to be able to use them correctly and efficiently.

1.3.2 Use case analysis

The second part requires the gathering and research of common use cases and required features with the aim to design a well defined DSL with a simple learning curve. Knowledge of standard automotive network communication protocols used in ECU development is crucial in correctly defining the language features. The DSL borrows both semantic and syntactic features from the Vector CAPL language, since it is an established tool with a similar functionality. Due to this design choice, users of CAPL should be able to easily switch to the GatewayScript (GWS) testing framework

1.3.3 Language design

The third part is the design and implementation of the GWS language coupled with a code generator with the purpose to transform code written in our DSL into a working application. Xtext is the tool used to achieve this. Xtext is a framework for development of programming languages and domain-specific languages. It offers an extensive infrastructure including parser, linker, typchecker, compiler as well as editing support for the Eclipse IDE.

1.3.4 Embedded software development

The final part consists of the embedded software running on the gateway hardware. The software can be visualized as a layered structure with three central layers. The lowermost layer incorporates the existing interface drivers and RTOS functionalities. The middle layer is the static part of the test software module written in C++. It includes the interface to the C part of the source code, C++ specific utility classes, and the common parts of testing software uniform through all configurations. The top layer is the script specific, generated test configuration. Xtext provides the necessary tools for the parsing of the script and generation of the embedded source code. Compilation and downloading to hardware will be integrated into an Eclipse based application created to use the Gateway from a PC.

1.4 Chapters overview

A brief description of the following thesis chapters are given below.

The second chapter gives a detailed description about language design by providing a theoretical background. The branch of science dealing with this is called programming language theory. After a general introduction formal definitions are introduced. The chapter ends with details regarding the implementation of languages.

In the third chapter I demonstrate both the frameworks used to create the whole language infrastructure, and the existing frameworks from the gateway upon which the generated code will run. In the first part I present the technologies used for language design, namely EMF, Xtext and Xtend. The second part describes both the existing Gateway software and its desktop framework.

In the fourth chapter I go over my design choices and give a detailed commentary about the implementation. I consider and compare several possible solutions to the conversion of the language into code executable by the embedded system. I present the GWS and illustrate its grammar meta-model. The different layers of the language implementations are explained and demonstrated with examples.

The fifth chapter contains a complete documentation of the language and its features. It serves as a learning material for future users. It details its language structure, keywords and use cases. It ends with two examples showcasing offer features, one of which was created for a real Electric Power Steering unit developed at the company.

The final chapter concludes my work and details possible features to be implemented in the language in the futures.

2 Language design

2.1 Domain specific languages

Language theory – also known as programming language theory or PLT – is the general study of programming languages. It is a well established branch of computer science dealing with the design, implementation, analysis and classification of languages. I will approach the matter from the subject of designing and implementing so called Domain Specific Languages or DSLs [6].

2.1.1 Language types

Programming languages can be split into two major categories.

General purpose languages

General purpose languages or GPLs are equipped to deal with problems ranging across all application domains, but generally takes longer to develop solutions, due to its tools being more general. They are a tool for programmers to instruct computers. They are turing complete meaning it can solve any problem a Turing machine can.

Popular general purpose languages today are for example C, Python and JavaScript. The reason for having multple GPLs is that their offered features and tools are optimized for different problems. In a sense these are also *domains*, but much larger than the ones targeted by domain specific languages. For example C offers pointers and bitwise operators to manipulate data directly in the memory. This makes it appropriate for embedded environments where low level programming is necessary. In Python the language has many built-in containers and utility functions making it ideal for solving small scale script tasks. In a sense this makes all languages domain specific. Smaller domains with more specific tasks drive people to design more specialized languages, leading to DSLs.

Domain specific languages

Domain specific languages are specialized to a particular application domain. A domain is an area of expertise or collection of problems. When discussing a particular domain of problems, subjects outside of it are irrelevant. The languages' built-in abstractions are aligned with the abstractions used by the domain. Their syntax is customarily tailored to express these abstractions effectively and concisely. Textual notations might be replaced by tables, symbols, graphical notations or a combination of the above. Having been highly specialized to a particular goal, DSLs usually lack the tools to solve problems outside their domain, and thus their use is limited to their target domains.

Popular DSLs include for example HTML for specifying the appearance of websites, Matlab for solving engineering and scientific problems and MySql for writing database queries for accessing databases. Optimalization comes in the form of execution as well as syntactic appearance, e.g. Matlab is equipped with highly efficient ways to operate on matrices.

2.1.2 Benefits of using DSLs

Once a DSL is built, development in the domain becomes significantly more effective. Concepts can be expressed concisely in the language of the domain, while hiding away implementational details. Libraries are similar in this manner, since they also encapsulate functionality relevant to a single domain. However, DSLs can rely on domain specific semantic and syntax checking, while libraries are committed to their native languages' features. Since a DSL works with domain specific concepts, semantic analysises can have straightforward high level implementations, also making more meaningful error messages available.

A good DSL guides its users to effective code by language structure. Since implementation is of no concern to the users, the execution engine is free to generate optimized code by removing overheads resulting from using abstractions.

Possibly the biggest benefit of DSLs is that non-programmer domain experts can develop programs independently from software developers, further increasing productivity.

2.2 Formal definitions

In the following section I introduce formal definitons for programs, languages and domains.

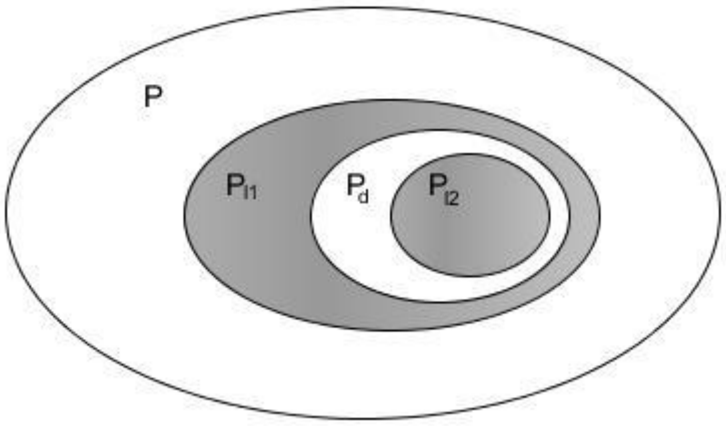
When speaking about programs we mean algorithms or computations that are executable on a Turing machine. Let P be the set of all conceivable programs. A program p in P is a conceptual representation of a computation. A language l defines a syntactic notational structure for expressing and encoding programs. A program p expressed in the language l is denoted as p_l. A program can be expressed in multiple languages, and sometimes multiple ways in the same language (a factorial can be computed with loops or recursion in many GPLs). A transformation T can be defined between languages l_1 and l_2, which maps programs from their l_1 representation to their l_2 representation, i.e. $T(p_{l_1}) = p_{l_2}$. A language may not be able to express the whole of P. Let P_l be the subset of P that can be expressed in l. Some languages might be better at expressing given subsets of P, evidenced by being for example more concise, or analyzable.

A domain D can be defined as a subset of P denoted as P_d. There are two general approaches for defining a given domain.

In the inductive (bottom-up) approach we analyze existing software – a subset of P – aimed at the same set of problems. By inspecting common characteristics and similarities we can outline our domain P_d. It is noted that the language used to express these problems are irrelevant. In the special case where the language *is* uniform – a specific language P_l –, domain specific patterns may arise. This makes the design process simpler, since abstractions can be clearly identified.

The deductive (top-down) approach views the domain as a body of knowledge outside the realm of software. The typical motivation for this approach is in order to provide (maybe previously non-existent) software support for the given field. Defining a domain this way requires thorough understanding of the area to be contained.

With these definitions we can now formally define Domain Specific Languages. A DSL denoted as l_d is a language specialized at expressing problems that encompasses the target domain P_d. It has abstractions similar to the target domain, and hides the irrelevant details from the user.



2-1. Image – Language under- and over-approximation

Since neither the domain definition and the language design is an exact process, it is up to the language designer to approximate a language that more or less covers the desired domain. Since even the domain interpretation is subjective to a certain extent, an exact coverage is generally impossible to achieve. It is best to aim for an under- or over-approximation first, and later evolve the language in iterative steps as the *language purpose* becomes progressively clearer.

2.2.1 Language purpose

Designing a language around a domain is not a straightforward process. There can be multiple language designed for the same domain, differing in the abstractions they make use of. Determining how to choose which abstractions to shape into the grammar depends on the *language purpose*. The way a language portrays a domain should be decided based on a specific purpose. Having a well defined purpose influences all stages of the language design process, and hence should be carefully planned out beforehand.

2.2.2 Language structure

So far we have defined languages as tools to express and encode programs. The following chapter elaborates upon the way languages accomplish their tasks. Programs are represented in two ways, both of which are part of a given language: the concrete syntax and the abstract syntax.

Concrete syntax is the interface (the concrete notation) with which the user interacts with the language. The notation can be textual, symbolic, graphical, or any combination thereof. Programs are created, edited and persisted in their concrete syntax forms.

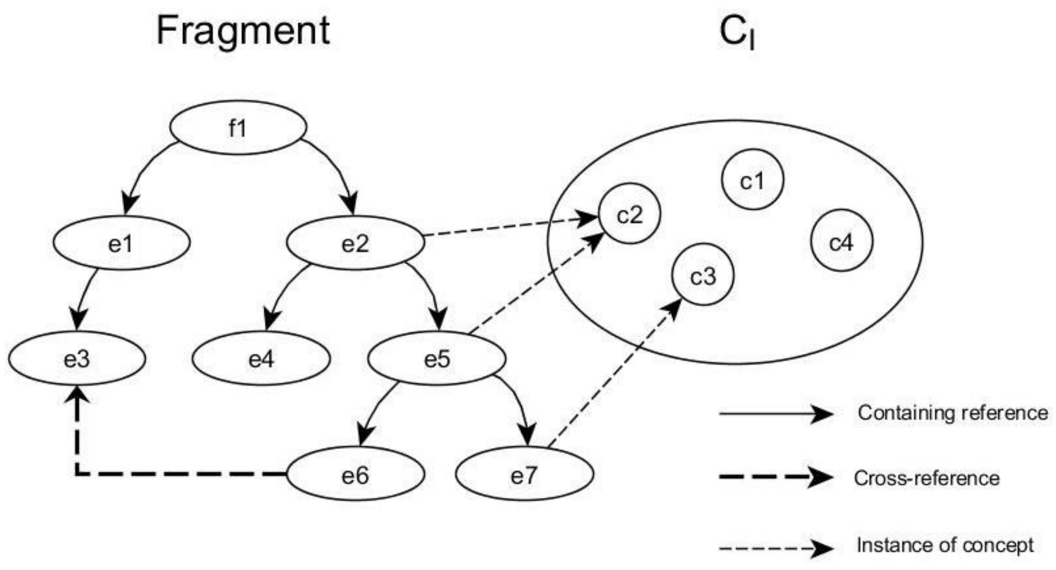
The *abstract syntax* is the semantic representation of a program. It does not contain notational details like whitespaces, commas and keywords. Abstract syntax is used for analysis and further processing of the program.

A language can be defined in this manner by specifying its concrete syntax, abstract syntax and a mapping between the two often called a *grammar*.

A language is constructed with either of the two folloing types of editors. *Parser based editors* lets users edit the concrete syntax. The abstract syntax is then derived with the help of the grammar rules. *Projectional editors* on the other hand gives users access to the abstract syntax. Altough still performed through the concrete syntax, it directly manipulates the abstract syntax. The concrete syntax is a projection, no notational parsing takes place. Ultimately the concrete syntax is used to edit and visualize the abstract syntax.

Abstract syntaxes are usually stored as trees, otherwise called as *Abstract syntax trees*, or *ASTs*. The individual tree nodes are elements of the language, which are instances of so-called *language concepts*, or simply concepts. By concepts we refer to both the syntactic and semantic meaning of the element. Nesting in the concrete syntax results in a parent child relation in the AST. Cross references between the branches are possible. A program can be made up of multiple ASTs called *fragments* connected by cross-references, resolved during linking phase.

A language l is made up of a set of concepts C_l. E_f is the set of program elements in a fragment f. Each e element of E_f is an instance of a concept c of C_l of the language l.



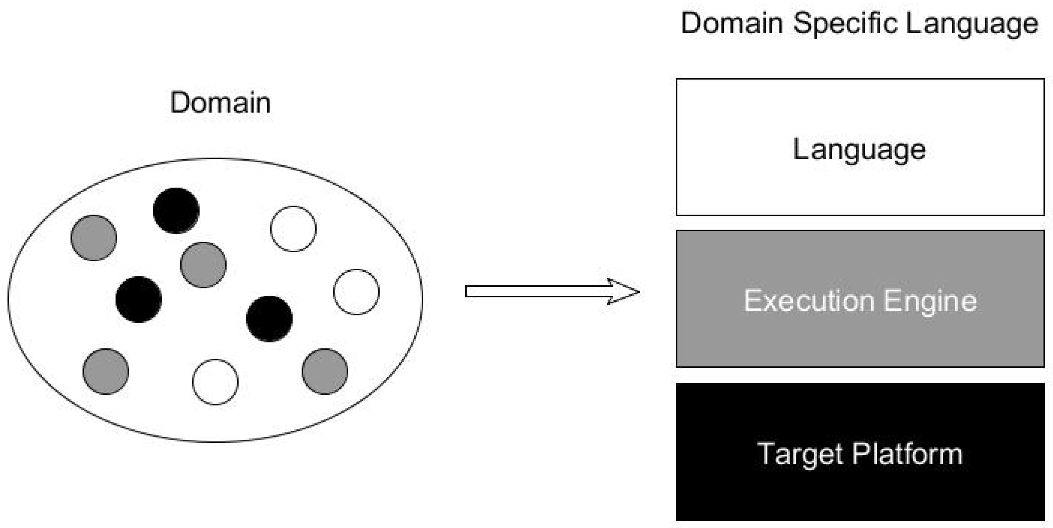
2-2. Image – Fragments, Elements and Concepts

2.3 Implementation

2.3.1 Separation of concerns

Every DSL has to be separated into three individual parts (based on the concerns populating the domain), taking on different roles when executing a program defined in the language. Concerns are specific elements that make up individual problems.

A DSL has three ways of dealing with these concerns. The user interacts with the *language* by creating programs. The *execution engine* transforms the program into a form runnable on the target *platform***.**



2-3. Image – Separation of concerns

Variable concerns (white circles) that regularly differ in applications can be specified in the language by the user, these usually being the abstractions or concepts found in the domain.

Derivable concerns (grey circles) can be implicitly derived from a given program. These are handled by the execution engine. Execution engines can remove the overhead that comes with expressing abstractions and generate efficient code.

Fixed domain concerns uniform through all programs are placed in the platform.

2.3.2 Interpretation versus Translation

Two main strategies exist for building an execution engine: translation and interpretation. In the case of translation, DSL code is first transformed into the language of an existing execution engine, which then generates the runnable object for the target platform. Interpretation means we generate the code directly for the target platform.

Translation is usually achieved by way of a suitable GPL. These come with optimized compilers or virtual machines that can execute code very effectively. If GPLs are not suitable to process the semantics of the language (its abstractions differ greatly), using an interpreter might be justified. However, designing compilers and interpreters require great proficiency and it is a field of expertise in itself.

2.3.3 Parsing

The language parser is the entity responsible for transforming plain text (the concrete syntax) into a processable form (the abstract syntax), as well as for taking up roles such as verifying syntactic correctness. In modern IDEs this is frequently done in real-time, keeping the abstract syntactic structure up to date with the textual representation. This makes it possible to provide live feedback to the user (e.g. outline, validation, semantic errors).

Parsing can be broken down into several phases.

Lexical analysis

Lexical analysis is the process of breaking down the program into a string of tokens. This is performed by the lexical analyzer, or simply, lexer. A token is an atomic element of a language. Types of tokens include syntactic keyword, (e.g. class in C++, or the arithmetic operators +, -, etc.), identifiers (e.g. a concrete C++ class name), or symbols (an instance of that class name). Lexers are usually implemented by using regular expressions.

Syntactic analysis

We must also determine if our string of tokens form valid sentence in our language. The validity of a given statement can be determined with the help of the so-called language grammar. A grammar is a rule set that describes the general syntactic structure of the sentences that compromise the language. This process is called syntactic analysis, or parsing. Parsing requires complex algorithms capable of dealing with recursions and solving ambiguities in the statements.

Semantic analysis

After the validity of our statements are established, parsers transform the concrete syntax into an AST for further processing.

Semantic checks about the validity of certain aspects of the program that are unavailible for checking during previous phases can be performed. These are language specific, but typical properties to check during semantic analysis can be type checking and scoping. Type checking ensures that assignments and various operations are performed with the correct types. Scoping deals with the lifetime and visibility of variables (managing a symbol table can help when dealing with scoping). For example variables should not be used before initialization, or access to variables are restricted when declared in a function.

Execution

When we determined that our statements are completely valid in our language, it can be run through an interpreter or compiler to be transformed into machine readable form.

2.3.4 Grammars

Different parser implementations are intended for different language grammars. Grammars specify the formal definitions for the concrete syntax, as well as the mapping between the concrete syntax and the abstract syntax. Failing to use the correct parser for a given language may result in unnecessarily convoluted grammars, or the language may even turn out to be impossible to parse.

Grammars are sets of grammar or *production rules* that define how valid sentences look like, or from a different point of view they implicitly declare all the valid sentences of a language.

Grammars can be classified into *context-free* and *context-sensitive* grammars. Languages using context-free grammars can be processed character by character without any awareness of the environment (the environment being everything not currently being processed). Context-sensitive grammars on the other hand require information found in the context of the parsed sentence, thus requiring the parser to be aware of a lot more than the actual syntax, making it more complicated.

Context-free subclasses

A particularly important subclass of context free grammars are LL(k) grammars, since ANTLR only supports generating parsers for these grammar classes. The first L in LL(k) stands for left-to-right scanning, and the second L stands for leftmost derivation. Both L letters can be replaced by Rs to mean right-to-left scanning and rightmost derivation, respectively. The constant k determines how many tokens the parser will look ahead to decide which production rule to use. A special case LL(*) means the parser can look ahead any number of tokens to find the correct parser rules. Higher k constants mean more possible syntactic forms, but typically reduces performance.

2.3.5 Programming paradigms

Programming paradigms describe convential behavioural structures employed by most programming language. Choosing a behaviour similar to the domains natural behaviour can greatly enchance the clarity of the language. Adopting an existing, or a combination of existing paradigms when designing a DSL can be hugely beneficial for a multitude of reasons. Firstly it saves effort that has to be invested in creating a coherent well structured behaviour. Secondly, existing paradigms have well established advantages and drawback, and can it be easier to tweak them to the language at hand, due to having personal experience with using them. Most programmers are also familiar with them, making the learning curve easier. Maybe most importantly it can make generating code considerably simpler for a language with the same behavioural structure, since its behavioural abstractions are closely aligned.

The following programming paradigms are widely used by todays programming languages.

Imperative

Imperative languages use a sequence of statements to influence the state of the program. It is the oldest known paradigm and the one most natural to computers. Both procedural (e.g. step by step algorithms) and object oriented programming are imperative, their main difference being in the structural organization of the statements. Due to their straightforward nature it is generally easier to understand imperative programs than their alternatives.

Functional

Functional programming uses functions as its base unit of abstraction to structure programs. A functions return value is based purely on its arguments. Calls with the same arguments result in the same results at all times, meaning functional programs are inherently stateless. Its main utilization is for computations, since it cannot affect its environment (being stateless, it can not alter the state of the environment either). Arithmetic expressions are often resolved in a functional manner.

Declarative

Declarative programs use rules and constraints instead of specifying a sequence of instructions. The rules define the logic of computation but not how to perform the computation. An evaluation engine is responsible for finding solutions in ways defined by the declarations. HTML is a famous declarative language, since it has no control flow. It describles the visuals of a webpage, but does not specify how to achieve that look.

Event-based

In event-based programming behaviour is triggered by events. Events may come from an external source or triggered indirectly by other events. It is often used when interaction with an external source (e.g. real world) is necessary. Changes to state can be easily expressed and processed by events.

Dataflow

In dataflow, variables refer other variables as dependencies along with a mapping between the two. A variable updates itself when one of its dependencies change, hence an external input change causes all variables to update that depend on it (either directly or indirectly). Dataflows are often visualized as blockdiagrams, since they convey information about the dependencies effectively. Hardware description languages (HDLs) often use dataflow because its parallel nature closely resembles the operation of digital hardwares.

2.3.6 IDE support

In ideal DSLs, IDE (integrated development environment) support should always be provided to the users. Programmers are much more inclined (and their work is made much more easier) to use a language if they are not required to code in a pure text editor.

Having an IDE as a framework offers multiple benefits:

- *Syntax highlighting* gives the programmer better understanding of the structure of the code through visual effects (e.g. colouring).
- *Auto complete* helps with automatically inserting cross referenced variables or existing functions for the user.
- *Error markers* point out syntactic (or possibly semantic) mistakes while editing the code, providing feedback while coding.
- *Quickfixes* offer typical solutions to arising errors.

- *Reference navigation* allows the user to move through the code through references.
- *Background parsing* provides real-time feedback about semantic information
- *Refactoring* lets users edit program elements and all their references simultaneously
- *Outline* of the current state of the AST can be provided to the user in a separate window with the help of background parsing
- *Custom features* can be supported, like visualization or graphical editing

3 Frameworks

3.1 Eclipse technologies for language design

Eclipse is an open-source integrated development environment (IDE). It is the most popular Java based IDE, due to having a plug-in (plug-ins are components that build up an Eclipse applicaiton) system allowing Eclipse users to highly customize the IDE, or create their own applications. Many softwares have been created over the years ranging from adapting Eclipse to be used for developing applications in different languages to various frameworks for a great number of application domains. The following technologies were used in conjunction to develop my DSL.

3.1.1 Eclipse Modeling Framework

EMF is a modeling framework for creating structured data models with code generating facilities [2]. It lets users specify a meta model described in XMI by way of a graphical or tree based editor and generates a set of Java classes derived from the model along with a set of adapter classes that enable viewing and command-based editing.

At EMFs core is the Ecore model which serves as the meta meta model for the framework. A meta model is essentially a template for creating model instances (a model to a meta model is like an object to a class), meaning the Ecore acts as the template for creating meta models.

Ecore consists of the following core concepts:

- EClass: a class containing an arbitrary number of EAttributes and EReferences.
- EAttribute: an attribute with a name and a EDataType.
- EReference: a reference to an EClass. Can be either containing or non-containing (meaning either a has-a or a refers-to-a relationship).
- EDataType: represent the type of an EAttribute (e.g. EInt, EString).

3.1.2 Xtext

DSL development is a time consuming and complex task, sometimes making it an undesirable endeavor. However there are various technologies available today that offer developers complete frameworks for quick development of DSLs, Xtext being such a technology [10]. It is an Eclipse based product, which is preferred at the company, since most existing softwares rely on Eclipse technologies. Xtext offers a complete toolkit that generates the whole infrastructure to create a complete custom made language. Xtext itself is a DSL that allows its users to define languages using a powerful grammar language. It automatically generates an ANTLR based parser. ANTLR (ANother Tool for Language Recognition) is a parser generator for reading, processing, executing, or translating structured text or binary files. It also generates a customizable IDE support complete with the most essential features. Xtext allows users to generate their own code from the AST, with any JVM[2] based language.

Xtext Grammar

Xtexts grammar language was designed for description of textual languages. The users describe the concrete syntax along with a mapping to the abstract syntax. To store the abstract syntax, an EMF meta-model is generated, along with the grammar specific parser. As such, languages created with Xtext are *parser based editors*. The parser transforms plain text into a semantic model (abstract syntax tree) according to the grammar, which can in turn be freely processed by the user. Understanding how the Ecore meta-model is built from a grammar is crucial in writing an effective language.

Terminal Rules

Terminal rules define data types for grammar. Terminals are handled as tokens by the parser, and are the concrete syntax representations of variables types in the abstract syntax. They usually return existing EDataTypes from the Ecore meta-model. Terminals are declared using regular expressions. The following terminal rule defines a syntax for the type int:

```
Terminal INT returns ecore::EInt:
  (’0’..’9’)+;
```

[2] Java virtual machine

Parser Rules

Parser rules process a sequence a terminals, and produce a tree of terminal and non-terminal tokens. While terminal rules define the valid tokens, parser rules also define the valid sentences of the language. Feeding a sequence of tokens into a set of parser rules produces parse trees, represented as EMF models in Xtext. Parser rules are represented as EClasses in the grammar meta-model. Parser rules, similar to EClasses can contain or reference other parser rules (EClasses) or terminal (EDataTypes).

The name of the EClass can be explicity declared or implicity inferred from the rule's name. Assignments add features to the EClass, its type inferred from the assigned value.

```
Person:
   ’Person’ name = ID;
```

The above code creates an EClass named Person, and creates an EAttribute name with the type EString (the terminal ID returns an EString). A valid sentence in this language would be ’Person John’, creating a Person object with the name John. There are three assignment operators:

- Simple assignment =, used on features to contain singular elements.
- Add operator +=, for storing multiple values in a list type feature.
- Boolean assignment ?=, returns true if a given token is consumed, otherwise returns false.

Features can also contain other parser rules, in which case the containing reference points to another EClass.

Cross references (non containing references) to existing EObjects can be assigned to features by square brackets. Cross references use the ID attribute to identify possible references.

A single parser rule can parse multiple sentences with different concrete syntax by employing the ’|’ logical or operator in the rule. We modify our example rule to the following:

```
Person:
   ’Person’ name = ID (’{’
   ((’child:’ new_children += Person) |
   (’child:’existing_children+= [Person]))+
   ’}’)?
```

The above changes the grammar to allow other people to be optionally assigned to a person in the feature children. The following cardinality operators can be used when defining rules:

- + operator: one or more
- * operator: zero or more
- ? operator: zero or one

A valid program with the modified rule would be:

```
Person Martha

Person Thomas {
        child: Person Bruce
        child: Martha
}
```

Actions

Simple actions can explicitly set the EClass of the EObject returned by the parser. With simple actions a single rule can create multiple types of EObjects without using delegation to other rules, improving the clarity of the grammar.

```
Person:
     ({Adult} ’adult’ name = ID) |
     ({Child} ’child’ name = ID);
```

The parser creates the correct object of either type Adult or type Child depending on the token it encounters (either ’adult’ or ’child’).

Assigned actions can assign the current object to a feature of a newly created object. This is primarily used when *left-factoring* grammars. Xtext relies on the ANTRL technology when generating parsers. ANTLR implements its generated parsers with LL(*) algorithm which does not allow left-recursive grammars.

```
Expression:
     (Expression ’+’ Expression) | INT;
```

The above code creates a parse tree for creating simple addition trees. When it encounters a ’+’ operator it calls itself on either side of the operator, until it happens upon an integer, in which case it ends the recursion with that terminal value. This implementation is left-recursive, since the first rule it encounters (on the left) is itself, and hence can not be parsed by LL(k) parsers. *Left-recursion* is commonly solved by *left-factoring* the grammar. We introduce new rules to eliminate the recursion element of the parser rule.

```
Add returns Expression:
     Primary ({left = current} ’+’ right = Primary)*;

Primary returns Expression:
     Value = INT;
```

The above left-factored grammar rule enters the Add rule when encountering an Expression (Expression is the abstract base class of both Add and Primary). It calls the primary and returns with an INT type if the current token is a terminal token. Otherwise, an *assigned action* happens. It creates a new Add EObject, and assigns the current Expression (which can only be Primary to it), and the other operand Primary to the feature right. This can go on indefinitely (as signified by the * operator), meaning everytime the parser encounters a ’+’ token, it assigns the current tree to the feature left, and assigns the right hand side operand to the feature right.

Unassigned rule call

Unassigned rule calls delegate to other rules without instantiating an EObject for the AST. The rule call will act as an abstract base class for the delegated rules in the grammar model.

```
Animal:
     Mammal | Bird;
```

Enum rules

Enum rules create user-defined enumeration types, which can be assigned to features in the grammar model.

```
Enum Direction:
  NORTH = ’north’ | EAST = ’east’ | SOUTH = ’south’ | WEST = ’west’;
```

Validation

Xtext provides custom abstract syntax tree validation in the form of constraints. Constraints are checked against the AST and evaluated to be either true or false. Constraints can be added separately to each EClass of the language grammar, checked each time an EObject of the given EClass is created.

Scoping

Custom scoping can be provided for cross-references in the grammar. For example, users can introduce namespaces and restricted access to variables with self-defined scoping mechanisms.

3.1.3 Xtend

Xtend is a flexible and expressive dialect of Java that seeks to improve on many aspects of the Java language [9]. It compiles into readable Java code, but provides its users with several modern language features missing from Java (i.e. type inference, operator overloading, closures). Additionally it adds syntactic sugar (template expressions, extension and dispatch methods), and removes some of Java's syntactic noise. The following features make Xtend an exceptional tool for code generation.

Template Expressions

When it comes to code generation, Xtend offers an extremely beneficial feature: Template Expressions. Templates in Xtend allow for readable string concatenation. An example template expression for generating an empty C function with a custom name could look like this

```
/* Function declarations start with the def keyword.
 Return type in the function declaration is optional,
 it is inferred from the function definition */
def generateCfunc(String cFuncName, String cStatement) ’’’
        void «cFuncName»() {
             return // A semicolon is optional in Xtend
        }
’’’
```

Characters enclosed by the triple single quotes (’’’) are outputted as plain text by the Xtend function. Xtend expressions can be evaluated by encasing them into so-called guillemets («»). This also allows loops and if statements with special keywords.

The following Xtend function outputs C code where an array is initialized with its Xtend function arguments given the actual element is bigger than zero.

```
def initCArray(Integer[] initValues) ’’’
        int cArray[«initValues.size»] = {
        «FOR intValue : initValue»
             «IF intValue > 0»
                  «intValue»,
             «ENDIF»
        «ENDFOR»
        };
’’’
```

This feature makes Xtend an ideal language for directly processing ASTs into C++ source code.

Extension methods

Extensions methods make it possible to expand the interface of a class (it's collection of public methods) in a non-invasive way. This is especially valuable when handling generated data structures. Since generated classes offer no functional interface, programmers usually create helper classes to encapsulate methods that work with the generated structure. By creating extension classes instead, the data structure and its operations will be indistinguishable from the viewpoint of the client[3].

Any method in Xtend automatically becomes the extension method of its first parameter[4]. It can be called through that classes objects (given the method declaration is in an accessible scope). An example extension method definition with its subsequent call:

```
// This acts as an extension method
def printInt(Foo aFoo, Integer aInt) {
     println(aInt)
}

def testFunction() {
     Foo foo = new Foo()

     // Call printInt as an extension of Foo
     foo.printInt(5)
}
```

It is worth noting that extension methods can only access the public interface of a class, since in reality it is not part of the class definition. Extension classes are specially created to collect the extension methods of a given target class. Clients use extension classes by injecting it to their scope. Injection is an Eclipse-based concept, where users simply declare dependencies on classes without specifying their origin. A framework then resolves the dependency in the background autonomously.

[3] Code segments that use a given class are defined as its clients.

[4] Given that parameter is a class

Dispatch methods

Dispatch methods are created in sets, made apparent by each having the same name and the same number of arguments. When calling on a dispatch method set, the actual method invocation is determined at runtime based on polymorphic type check performed on the arguments. Coupled with extension methods, it supplies polymorphic behaviour to inheritance trees while adhering to encapsulation.

A dispatch is declared using the dispatch keyword. An example dispatch extension method on an inheritance tree in which Dog and Cat classes inherit from an abstract Animal base class.

```
def dispatch makeSound(Animal animal) {
     println(„Called on abstract base class, error”)
}

def dispatch makeSound(Cat animal) {
     println(„Meow meow”)
}

def dispatch makeSound(Dog animal) {
     println(„Bark bark”)
}

def testFunction() {
     Animal cat = new Cat()
     Animal dog = new Dog()

     /* Methods with zero arguments can be accessed as a property, and
    called without parenthesis */
     cat.makeSound // outputs „Meow meow”
     dog.makeSound // outputs „Bark bark”
}
```

3.2 Gateway

The „Gateway” is a hardware developed at our company for testing Electronic Control Units through communication interfaces, mainly CAN and FlexRay. It is equipped with the following hardware peripherals:

- 3 CAN channels
- 1 FlexRay channel
- 1 LIN channel
- GPIO and ADC ports
- Ethernet port

The hardware is equipped with two safety-critical TMS570 processors and an STM32 to act as an independent watchdog in case of runtime system failure. Since a FlexRay communication requires a minimum a two nodes for clock synchronization to operate, a separate processor is necessary to be able to cold start a FlexRay communication independently of other participating nodes. The GPIO/ADC port can be used to emulate the ignition signal, or check the voltage supply. The ethernet port is used to communicate with a PC when controlled by desktop applications.

3.2.1 Embedded software

The software provides the basis of functionality for both the desktop application (see 3.2.2) and the DSL. It uses FreeRTOS[5] as its operating system to introduce multi threaded processing.

Modules

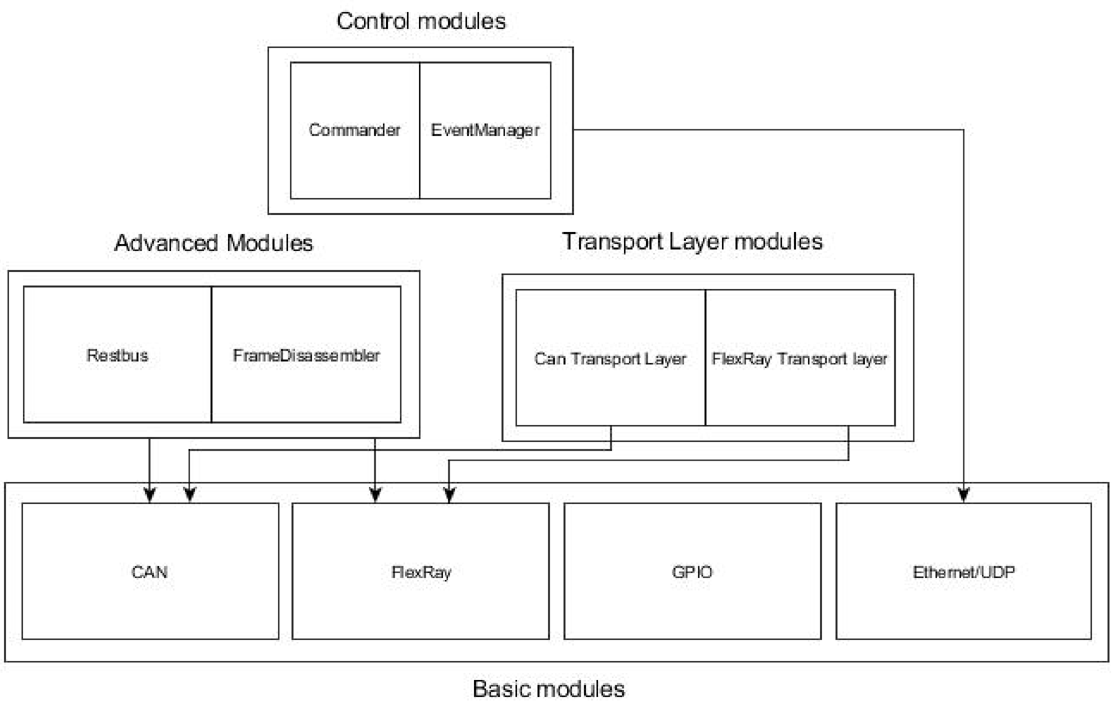
The software is organized into into modules according to funcionality. Each module has a distinct role in the system, usually encompassing a peripheral or a more advanced feature. The following notable modules can be found the software

- Can Driver: Clients can configure the three Can hardware channels, and create transmit and receive message objects on them. Transmit messages can transmit data on its assigned channel, while data received on Receive messages can be processed by events (see below).
- CanTransportLayer: Clients can communicate on the Can Transport Layer protocol defined by the AUTOSAR standard. Transport layers support transmission of data unable to fit into single messages.
- FlexRay Driver: Clients can configure FlexRay channels. An independent optional second channel can be configured to use the Gateway as a coldstart or sync node. No data transmit is allowed on the second channel. Transmit and Receive objects can be created analogously to the Can Driver.

[5] FreeRTOS is an open source real time operating system targeted at embedded environments.

- FlexRayTransportLayer: Clients can communicate on the FlexRay Transport Layer defined by the AUTOSAR standard.
- Restbus: Restbus provides a data structure for creating frame structures identical to the one specified by AUTOSAR. Clients can construct Frame, Pdu, Signal objects and map them to each other. A transmit resource (either CAN or FlexRay messages) and a time period is assigned to each frame. The restbus transmits all configured frames and its contained signals periodically. Users can influence the transmitted data by setting individual signal values. Signals represent real-world information (e.g. wheel speeds, steering angle). In this way, the Restbus hides the underlying communication interface and lets users deal with information relevant in the application layer.
- FrameDisassembler: Similarly to the Restbus, the FrameDisassembler expects a frame structure configuration. Instead of transmit resources, it expect Receive resources (CAN receive messages, FlexRay receive messages). Users interact with signals, but the values are read from incoming messages instead of setting them for transmission. The Disassembler allows the observation and processing of a communication from the application layer.
- GPIO Driver: Clients can configure pins to input/output. Configured pins can be read from or written to.
- EventManager: Listeners can register to events through the EventManager (see the section of events below). The Event manager is responsible for notifying all listeners when an event happens.
- Ethernet/UDP Driver: Handles communication with desktop applications.
- Commander: The commander interprets and executes the commands coming from a PC on the ethernet connection.

A clear hierarchy between modules can be observed, where more advanced features depend upon driver modules.



3-1. Image – Module hierarchies

Events

Events are a way for modules to distribute information to clients. Modules can register to events they want to listen to, and in the case of events the modules broadcast the event to all *listeners*. Registrations happen dynamically during runtime, to prevent static coupling to all listeners[6]. Listeners can register for the following events:

- Can Events
- FlexRay Event
- Can Transport Layer Event
- FlexRay Transport Layer Event

3.2.2 Java API

While the embedded software provides the means to configure and use the Gateway, The Java API is the user interface throgh which actual configuration happens. For every embedded module there is a corresponding Java package.

[6] This is also known as the Observer design pattern in the OOP community.

Connections

Communication between the Desktop and the Gateway is done through UDP[7]over Ethernet. Gateways can be connected to by way of an assigned Ip address. After a connection is established a dedicated alive keeper oversees to its lifetime.

Commander protocol

The commander serves as an adapter between the desktop and the embedded API. Individual function calls to the embedded modules are wrapped in *commands* at the desktop level. Commands store overhead information about their destination and identity. Commands received on the hardware get interpreted by the Commander module. A *command response* gives assurance that the command was executed, and if required, a return value or structure.

Java events

Users can register listeners implented in Java through Java interfaces. Events are sent through the connection and can be freely processed. The API offers most of the events availible in the embedded software for listening by implementing the desired interfaces.

3.3 Gateway2Hell

Gateway2Hell is a standalone Eclipse based application also known as an RCP[8] application. It offers GUI[9]s for Restbus simulation and Tracing.

Configuration

A communication bus, its participating nodes, and its defined messages, frames, pdus, signals and their mapping to each other are collectively called communication networks. There are established file formats to store these networks, for persistence and distribution. The most common are CanDB for CAN protocol and Fibex for FlexRay.

Configuration is done automatically by the application, based on gateway specific configuration files. The Eclipse Modeling Framework was used to create the

[7] User Datagram Protocol

[8] Rich Client Platform

[9] Graphical User Interface

configuration models. Two separate meta-models were created for reasons of modularization and reuse. The *Communication Configuration* (ComCfg) model is a network description file similar to CanDB or Fibex (it stores both CAN and FlexRay configurations), whereas *Gateway Configuration* (FbgwCfg) maps the abstract resourses found in a ComCfg to a concrete realization of the stored communication network to be used by the hardware.

Gateway2Hell offers parsers for both Fibex and CanDb for converting these into communication configurations. FieldbusGateway configurations can then be created from Comcfgs with a wizard. The wizard simply prompts the user to choose the frames (present in the referenced ComCfg) to be used by the Restbus or the Trace module. Based on the decision a transmit or receive message will be created from the abstract message in the ComCfg.

Restbus

Creates a GUI for changing the individual signals described in configuration files.

Trace

Users can trace the incoming and transmitted messages in a log window with various features to customize the view.

4 GatewayScript

In following chapter, I am going to give a detailed explanation about the design and implementation process that led to the completed language. The process can be broken down into the following stages:

4.1 Use case analysis

My task was to create a language with the ability to effectively and concisely describe automotive communications and assign dynamic behaviour to them. Since softwares already exist targeted at the same domain it was a relatively straightforward process to translate it into a list of essential features to be provided by the language.

CAPL[10] is a script language with a similar purpose created by Vector Informatik GmbH[11] to be run on their hardware products. Additionally our own existing software framework (Gateway2Hell) already provides a subset of the required functionality.

I collected several common applications (used at the company), that the language should be able to handle.

Restbus simulation

Restbus (remaning bus) simulation is when we provide a simulated communication environment to a target ECU. This is usually done by the sending of periodic messages. The simulating node might be required to process incoming messages from the ECU. The transmitted data is either set by the test engineer or computed from received messages.

Signal conversion

Signal conversion treats signals as its basic unit of operands. Operations are defined between signals, where the actual computation defined by the operations are performed on signal values. For example we can define a given signals value to be the sum of two other signal. Transmission of the signals can be handled by a Restbus.

[10] Communication Application Programming Language

[11] Vector develops software tools and components for networking of electronic systems

Network adapter

Adapters are used to connect incompatible communication networks. Signals might be in different messages, or missing. Adapters can transform and move signals to their correct message and position, or complement the communication with missig frames.

Diagnostics/Flashing

Diagnostic and flashing (the downloading of binary files to the ECU) protocols are customarily implemented over transport layer protocols.

Domain definition

I collected the following list of domain concepts deemed to be important in such a language based on CAPL, Gateway2Hell and the use case analysis:

- Transmit and Recieve data on CAN protocol
- Transmit and Recieve data on FlexRay protocol
- Communication configuration loading/persisting mechanism
- Give access to the GPIO/ADC driver
- Allow versatile low level data manipulation
- CRC and checksum support
- A language structure that allows periodic and triggered events
- Support for transport layers and optionally for diagnostic protocols

Gateway2Hell vs GWS

While Gate2Hell shares a lot of its core features with the DSL, one does not make the other obsolete, since both are targeted at differing situations. The DSL generates a standalone embedded software capable of running independently making it possible for it to operate autonomously without human supervision. On the other hand Gateway2Hell requires a constant live connection to a computer to function. In return it presents the communication in human readable form with the Trace feature. Trace makes it possible to both log and observe the communication in real time. The DSL is designed to be able to express much more complicated communication behaviours, which the desktop version is incapable of expressing. This makes the DSL the more versatile tool, while Gateway2Hell can be mainly used to verify the communication.

In the end, the IDE and the whole infrastructure for the DSL will be integrated into the desktop application. Users will use the Gateway2Hell to code, compile and download the scripts to the Gateway.

4.2 Planning phase

As previously noted in the *language purpose* section, mapping a domain to a language should be thought through in advance.

For behavioural structure I adapted a combination of event-based and procedural paradigms similar to what CAPL uses. Users define events with trigger conditions, and assign a sequence of statements to execute.

Events

There are three subtypes of *events*. Start events are triggered only once, at the start of the program. It can be used to initialize global resources and configuration. Timing events offer periodic triggering of events. In the case the period is zero seconds, execution happens once. This is a special case since it effectively takes up the role of a function allowing code reuse for other events. Communication events are triggered when receiving particular messages from the communication channels (CAN, FlexRay). It also provides the following code the information found in the message, most importantly the payload. It is also possible to listen to all events collectively. Events listening to specific messages enjoy priority when receiving, when a receive all event is present.

Statements

Statements can be separated into two interlocking components. An expression language modelled on the C language makes up the core part.

A list of features implemented in the expression language:

- Variable declarations and references
- Type support for numeric, array and boolean types
- Arithmetic and boolean operators

- If-else statements
- For, While, Do-While loops

The other component gives access to the low level Gateway driver modules. The full API of the modules are ported into the languages abstraction set. The currently implemented modules are the following:

- GPIO
- CAN
- Restbus
- FrameDisassembler

FlexRay Protocol and Transport Layers are omitted in the initial version of the language, and will later be included in subsequent iterations.

Configuration

Since a well defined communication descripting structure already exists, I decided to reuse the EMF meta-models (FbgwCfg and ComCfg) to load configurations onto the Gateway from the script, instead of defining a separate structure or letting the users configure dynamically at run-time.

Importing configurations shall be global, run-time independent and static, which is why they are declared independently of events. An arbitrary number of FieldbusgatewayConfigurations can be imported, with an option to add identifiers to each. This is to prevent name clashing by qualifying references to model elements with model identifiers.

Code generation vs Interpreting

When comparing the advantages and disadvantages of interpretation versus translation in terms of implementational benefits and complexity, it became abundantly clear that implementing an interpreter for the GatewayScript introduces overwhelmingly more complexity than its counterpart.

Allowing variables and loops in the language requires the interpreter to keep a symbol table and a stack. With code generation it is enough to translate these concepts into the textual representation for the chosen GPL (C++).

Symbol tables keep track of existing objects (in our case, only variables) and resolves references between them. It usually handles lifetime and accessibility (related to scopes) checks.

Stacks are traditionally used when dealing with nested scopes and function calls to manage the lifetime and visibility of local variables

Additionally since interpretation would have to be done on the gateway (we want to avoid runtime dependency to a desktop environment), an intermediate byte code would have to be defined to be downloaded to the gateway for execution. Interpretation itself takes up runtime resources, which are critical to an embedded system, especially in one that has to keep up with strict real time communication protocols.

Since it became evident early in the research that interpreting introduces significant complexities both to the development phase and the to the runtime behaviour, I decided to use translation into GPL (C++) code without further investigating the matter. Letting the compiler take care of the lowest level of code generation, significant optimialization can be achieved.

Separation of concerns

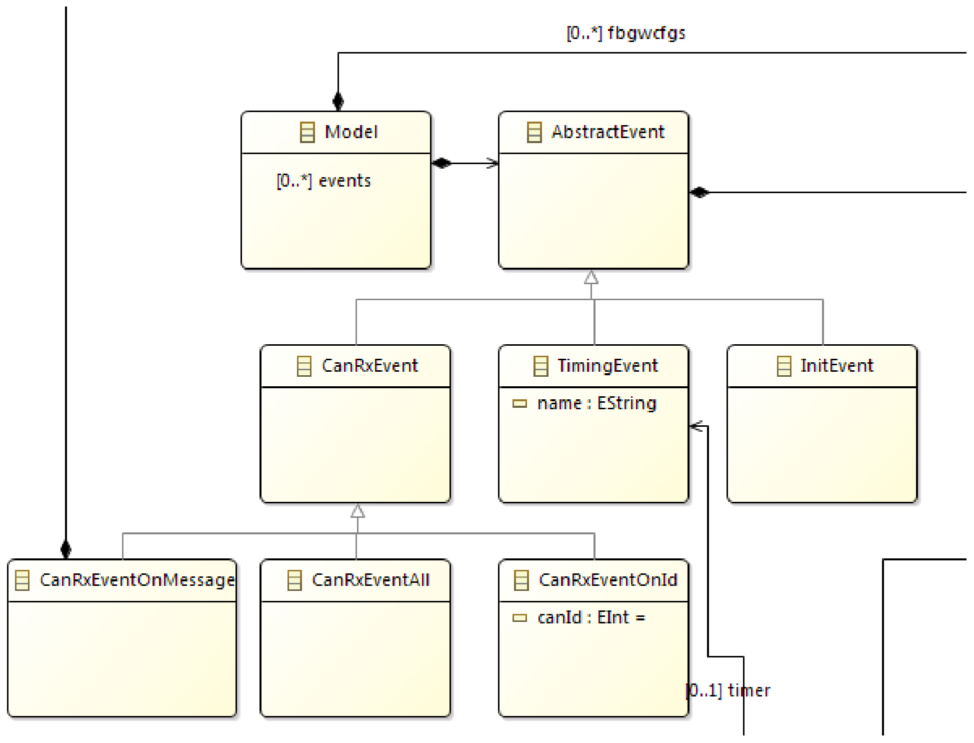
The three parts of a DSL implementation (talked about in section 2.1.2.), are the language, the execution engine and the target platform. In the case of the GatewayScript DSL Xtext will be used to create the language, Xtend for C/C++ code generation and the Gateway hardware acts as the target platform. Concerns, their type and implementations will be noted during the chapter.

4.3 Grammar

I am going to illustrate the GatewayScripts grammar by introducing the different parts of its meta-model inferred from the parser rules by Xtext. It should be noted that the semantic meaning of these grammar concepts are irrelevant from the viewpoint of the grammar. The represented functionality of these concepts are attached by the code generator and executed by the target platform.

The language meta-model can be separated into five distinct categories. These are events, gateway configuration references, statements, expressions and function calls.

Events



4-1. Image - Events

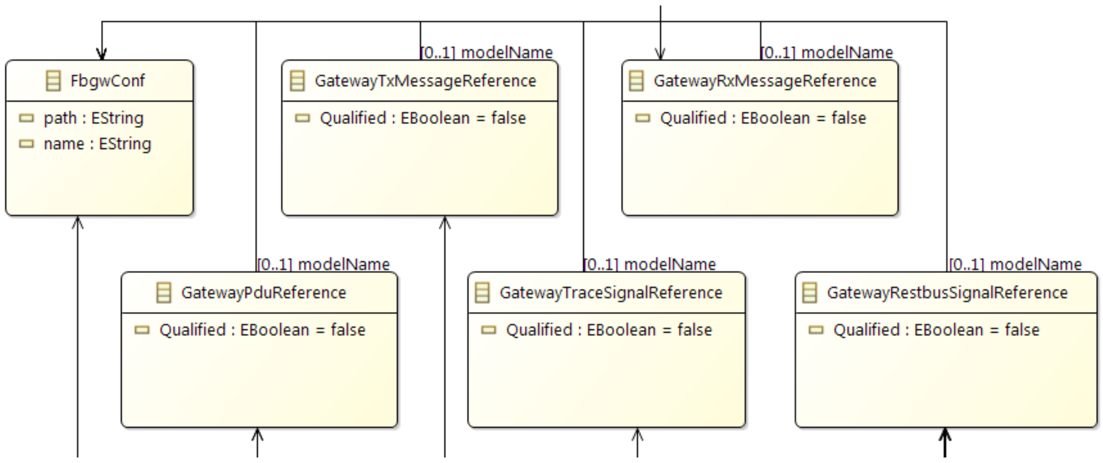
The root element of a program, *model* can contain an arbitrary number of FieldbusGateway configurations, and AbstractEvents. AbstractEvents have five derived classes, three of which are also derived from an abstract RxEvent class. Events are defined in the following way in Xtext:

```
AbstractEvent:
        InitEvent | TimingEvent | CanRxEvent;

InitEvent: {InitEvent}
        'On' 'start' '{'
             (lines += Statement)*
        '}';

TimingEvent:
    'On' 'timer' name=ID '{'
        (lines += Statement)*
    '}';

CanRxEvent:
        ('On''Message'{CanRxEventOnId} canId = INT |
        'On''Message'{CanRxEventOnMessage}     gatewayRxMessageReference     =
GatewayRxMessageReference |
        'On''Message''*'{CanRxEventAll})
        '{'
             (lines += Statement)*
        '}'
;
```

Depending on the keywords and arguments, the correct event type can be identified in all cases.

FieldbusGatewayReferences

References to the imported configurations are done indirectly through mapping classes.



FbgwConf
path : EString
name : EString
[0..1] modelName
GatewayTxMessageReference
Qualified : EBoolean = false
[0..1] modelName
GatewayRxMessageReference
Qualified : EBoolean = false
[0..1] modelName
GatewayPduReference
Qualified : EBoolean = false
[0..1] modelName
GatewayTraceSignalReference
Qualified : EBoolean = false
[0..1] modelName
GatewayRestbusSignalReference
Qualified : EBoolean = false

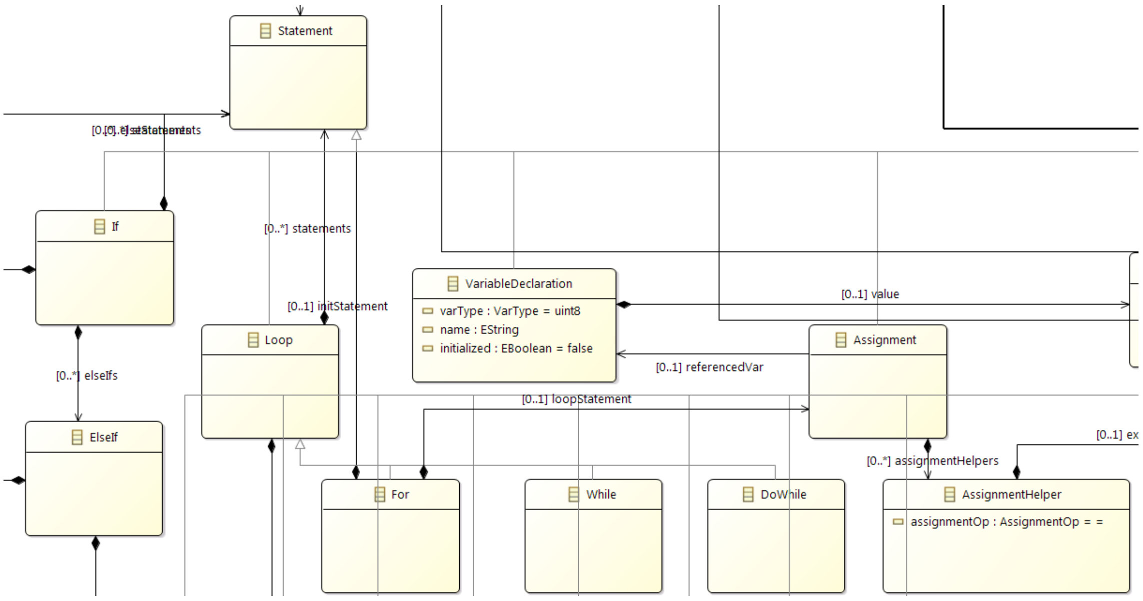
4-2. Image – Gateway references

These classes have hidden references to elements in the actual FbgwCfg Ecore model. Using separate classes for mapping means, that extra information can be added, like storing a qualified attribute, to allow for model specific scoping. The import rule and an example reference rule:

```
FbgwConf:
        'import' path=STRING 'as' name=ID
;

GatewayTxMessageReference:
        (modelName=[FbgwConf]     Qualified     ?='::')?     namedElement     =
[FbgwCfg::CanTransmitMessage]
;
```

GatewayTxMessageReferenceRules can cross-reference existing FbgwConfg instances with an obligatory scoping keyword ’::’, setting the Qualifier flag to true.

Statements



4-3. Image - Statements

Statements represent both individiual lines and complete ifs, and for/while/dowhile loops. The derived classes of Statement are If, Loop, VariableDeclaration, Assignment and Expression. With the exception of expressions, all other concepts’ syntax are hardcoded into the grammar.

```
Statement:
      FunctionCall';' | VariableDeclaration';' | Assignment';' | Loop | If
;

VariableDeclaration:
      (isStatic ?= 'static')? varType=VarType name=ID (initialized ?= '='
value = Expression)?
;
```

FunctionCalls return Expressions, so it is not part Ecore grammar (see the section about function calls). The VariableDeclaration rule has an optional isStatic flag, set if the keyword static is consumed. VarType is an enumeration containing the possible variable types. After the name of the variable, an optional expression can be added to initialize the variable.

Expressions

The ruleset handling the parsing of arithmetic and boolean expressions can be viewed as a subgrammar of GatewayScript. The expression grammar should be able to process expressions like:

```
!(true && ( 3 + 2 * 4 > 12))
```

This is done by a modified version of the left-factoring introduced in section about the Xtext grammars (section 3.1.2.). We adopt the structure of the standard left-factoring method, but introduce additional rules above it [5]. The lowermost four rules look like this:

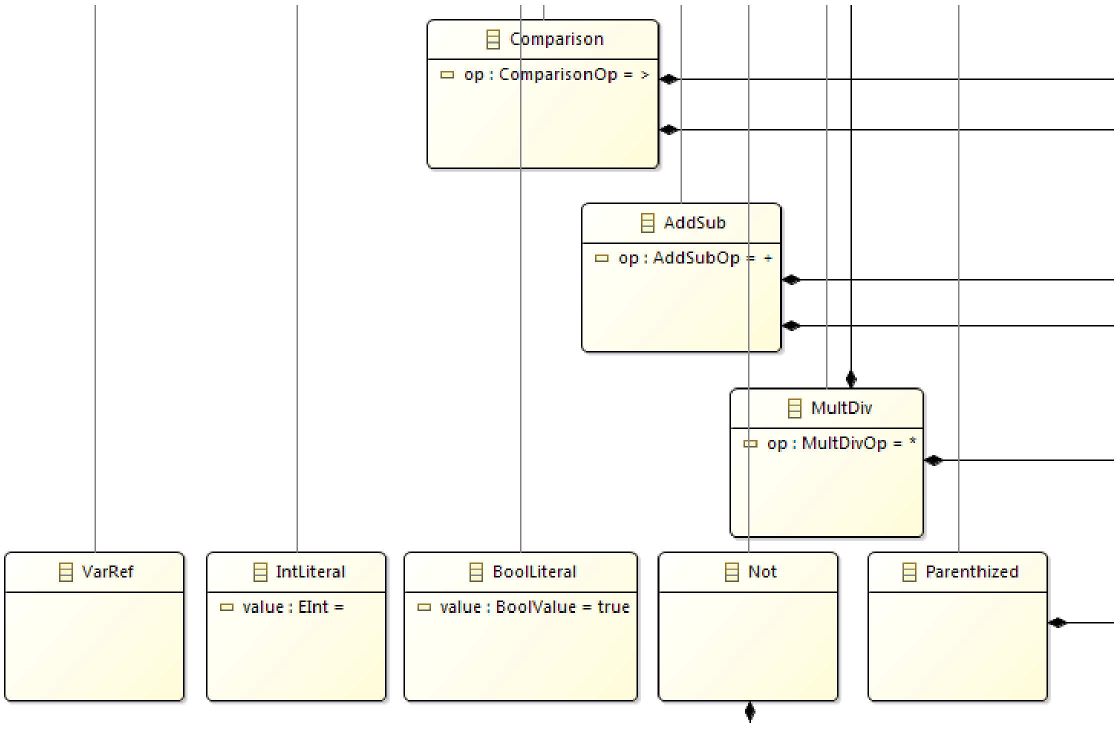
```
Comparison returns Expression:
     AddSub ({Comparison.left=current} op = ComparisonOp right=AddSub)?
;

AddSub returns Expression:
  MultDiv ({AddSub.left=current} op = AddSubOp right=MultDiv)*;

MultDiv returns Expression:
  Primary ({MultDiv.left=current} op = MultDivOp right=Primary)*;

Primary returns Expression:
     {Parenthized}'(' expr = Expression ')' |
     {Not} '!' expr = Expression |
     {BoolLiteral} value = BoolValue |
     {IntLiteral} value=INT |
     {VarRef} referencedVar = [VariableDeclaration] |
     FunctionCall
     ;
```

Structuring the rule like this, introduces operator precedence into the parsing process (e.g. we want to parse 3 + 4 * 5, so that the four and five are part of the multiplication). The later a rule is inserted the bigger its precedence will be, since all rules start by delegating to the one below it, the lowermost rules are tested first for consumption. If the expected operator or value is not present, it jumps back to the caller trying to consume the actual operator of that rule.

The complete list of operators and their relative precedent to each other, starting with the operator of the lowest priority:

- Logical or: ||
- Logican and: &&
- Equality checks: ==, !=
- Comparison operators: >, >=, <, <=
- Addition and substraction: +, -
- Multiplication, divison, and modulo: *, /, %
- Parenthesis, Negation (!), Literals, Variable references, Function calls

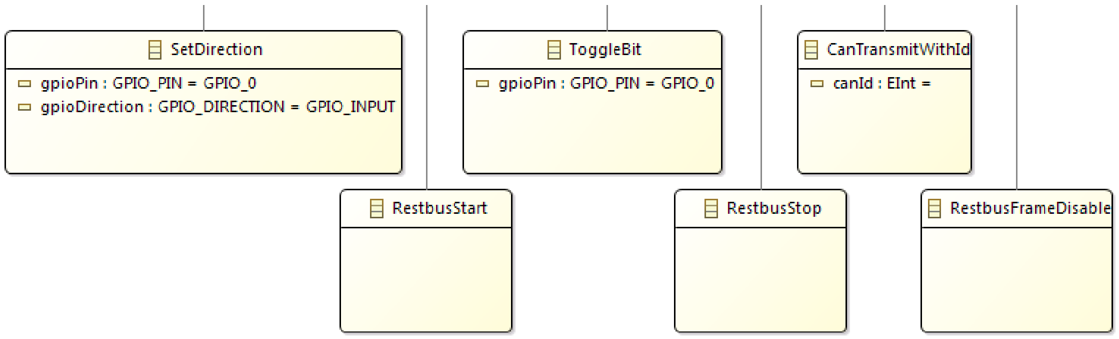
The meta model representation of the expression structure:



4-4. Image - Expressions

Function Calls

Function calls are the individual API calls to the gateway wrapped inside language concepts.


SetDirection
gpioPin : GPIO_PIN = GPIO_0
gpioDirection : GPIO_DIRECTION = GPIO_INPUT
ToggleBit
gpioPin : GPIO_PIN = GPIO_0
CanTransmitWithId
canId : EInt =
RestbusStart
RestbusStop
RestbusFrameDisable

4-5. **Image** - Function calls

Every function call is a derived class of the Expression base class, making it possible to embed calls into arithmetic and boolean expressions (so that their return values can be processed as part of an expression tree). Functions can also be called as standalone statements through the FunctionCall rule. Even though the rule returns an Expression the Statement rule delegates the FunctionCall rule specifically (the statement rule can be seen in the Statements section), to allow this behaviour (this is required since arithmetic expressions are not allowed as standalone statements).

```
FunctionCall returns Expression:
        'GPIO.' GpioModuleCall |
        'CAN.' CanModuleCall |
        'RESTBUS.' RestbusModuleCall |
        'TRACE.' TraceModuleCall |
        'Pdu' PduModuleCall |
        {TimerModuleFunction}   timer   =   [TimingEvent]   '.'   call   =
TimerModuleCall
        ;
GpioModuleCall returns Expression:
     'toggleBit('{ToggleBit} gpioPin = GPIO_PIN')' |
     'setDirection('{SetDirection}  gpioPin  =  GPIO_PIN','gpioDirection  =
GPIO_DIRECTION')'
;
```

TimerModule calles are treated differently, because the start and stop timer functions must be called on existing objects. Thus the call itself is moved to a feature, so the parent object can contain the cross-reference.

4.4 Validation and Scoping

Two types of validations are used with the GWS. The first is model validation performing checks regarding events. A few example constraints: only one Receive All and Start event is allowed, multiple Receive events can not listen to the same CAN ID.

The other, type checking is implemented for the expression language. The supported types are numeric and boolean, represented by an artificially created enum type Type. A type provider Xtend class assists in validation. ExpressionTypeProvider declares a getType dispatch method overloaded for all expression subclasses, returning their correct Type value. Additionally constraint checks are created for all subclasses. These constraints are static in nature, makig it easy to implement. For example, the < operator checks if both of its operands are numeric, and returns a boolean type when getType is called on it. One of its operands might be a multiplication which indeed returns the Type numeric, and in turn checks its own operands. This method results in a simple system for type checking, where subtypes provide their types to the containers, and checks contained expressions for type correctness against their own constraints. Consequently, in this approach complex expressions are validated recursively.

Special scoping is implemented for referencing configuration resources found in the imported Ecore models. Imported FbgwModels are loaded real-time by a background parser during editing. When attempting to create a cross-reference to a model element, the scoping mechanism automatically provides the valid possibilities to the user. It is possible to narrow down references to individual models by prefixing the reference with the model ID given in the import statement.

4.5 Execution Engine

Code generation is handled by Xtend template expressions and dispatch methods. Generation is organized into several classes with well defined responsibilities to keep a clear structure to the engine, and make introducing future features easier. When talking about class generation both the declaration and definition are included, meaning both the header and the cpp file is generated.

The following Xtend classes make up the execution engine:

GatewayScriptGenerator is the entry point of the generation process. It receives the AST for processing and distribution. It generates the derived action classes for the events, and the setup function instantiating events and actions.

GeneratorHelper generates the modules concerning events. Its output are the TimerModule and NotificationProxy classes.

FbgwConfigurator transforms the imported FbgwCfg models into code, by iterating through the tree model and processing the elements individually. It builds arrays and maps of the created objects to make later referencing possible.

FbgwConfiguratorHelper contains the stateless methods (methods requiring no access to member or external variables) used by the as static methods to keep the generating class easier to understand.

AbstractEventExtensions/ExpressionExtensions/StatementExtensions

Extension classes are created for base classes with many subtypes. They consist of overloaded extension dispatch methods for all derived classes. They abstract the generation of the extended classes from the generator classes, making it significantly simpler to process them. The whole generation of the imperative behaviour (the actions) are handled by the Statement and Expression extension classes, as well as Event generation used by the GatewayScriptGenerator.

Derivable concerns are computed during code generation. These include specifying the size of array and maps, filling them dynamically (which translates to static assignments from the viewpoint of the generated code, being seemingly hardcoded), computing the common period of the timermodule. Cross references are also resolved during generation by translating the String or ID references to array indices. This is done by storing the created object (e.g. CAN message) in the C++ code in arrays to a known index. Said index is then assigned to the language reference (String or ID) in an Xtend map. Later cross-references can refer back to the map to find the referenced C++ object.

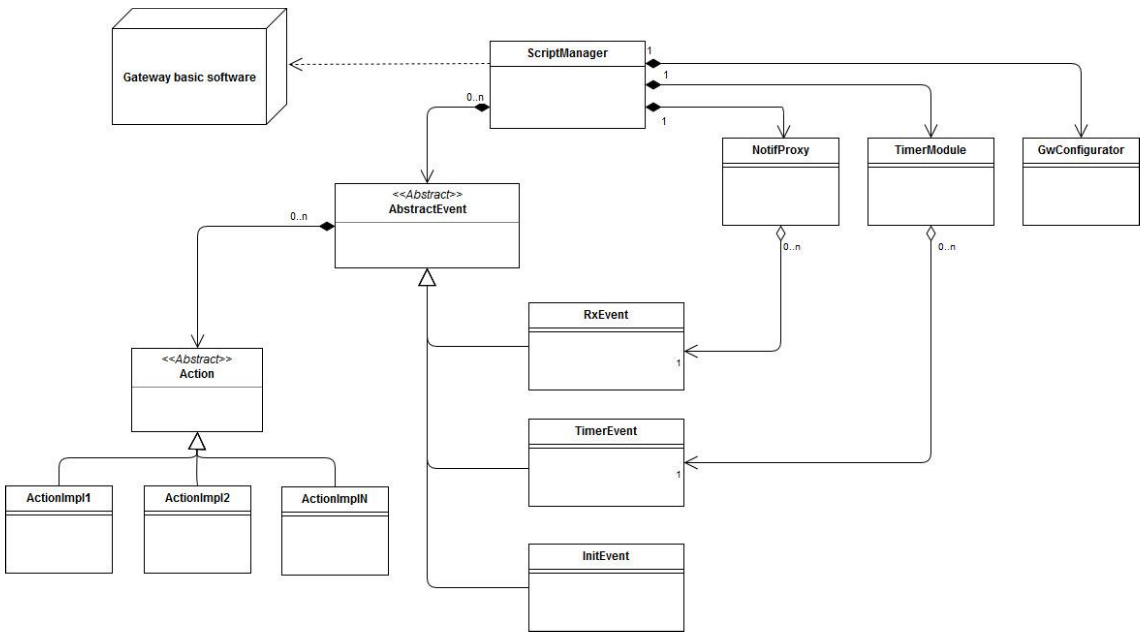
4.6 Target Platform

The target platform for the DSL is the embedded software, and can be split into three layers. The lowermost layer is made up by the the OS, driver modules and existing features providing functionality to the upper layers.

The middle layer is the foundation upon which the generated code can be executed. It acts as an adapter between the lowermost layers native C code and the uppermost layers generated C++ code, and as the intermediary communication interface for the other layers. It delegates requests from upward, forwards events and abstracts OS features into pure C++ concepts. Besides acting as the bridge between layers, the constant part of the C++ software is also found here, along with base types for generation and utility classes.

Utility classes include template arrays and maps to define type and size at compile time (making dynamic memory management obsolete), thread safe template variables for global resources (this is accomplished by using private FreeRTOS mutexes to protect write operations) and a mallocator class (to overload the new and delete operators provided by the compiler with our own mallocator algorithms written for the basic API.

These are the *fixed domain* concerns, talked about in the Separation of Concerns section. This layer defines a structural skeleton, for the generated code to adhere to. This predefined structure makes it possible for the upper layer to fit into the consistent parts.



4-6. Image – Target platform UML diagramm

There are four main classes to separate the core features of the language into independent units. The Gateway Configurator is responsible for configuring the gateway (generated accordingly to the imported EMF models). It also builds arrays and maps for the created object instances (e.g. CAN messages, frames, signals, etc.) for the run-time components to access.

For event handling, two separate classes are responsible for the Timing and Receive events, respectively. The timing module is provided with a core function by the middle layer that is periodically called by a FreeRTOS task. The task period is specified by the code generator, computed based on the (this is a derivable concern) Timing events created in the script (it finds the greatest common divisor). The core function keeps a counter to keep track of time. Timing event sstore their period in ticks. In each tick, each event is checked for its triggering conditions.

Receive event handling is managed by the NotificationProxy. The class registers itself as a CAN Event Listener to the CAN driver. Upon receiving a message, it delegates the information it carries to the correct event, and calls its statements. The class possesses a map, where CAN IDs are mapped to the concrete events. This makes delegation to the events simple, since events coming from the API carry this information.

The imperative behaviour of events are assigned by way of Actions. Action is an abstract base class with a single virtual function, perform. All events implement their own derived Actions, overriding the perform function with code corresponding to the statements assigned to the event in the script. Perform has multiple declarations – a functions declaration (sometimes called signature) is its combination of its return type and the number and type of its arguments, since different events may need to pass down arguments to the action (receive events pass down the payload found in the CAN message, whereas Timing events have no need of arguments).

All event types have an abstract base class as well, which has a pointer reference to an abstract action class. Derived events perform actions by calling the inherited Action pointers virtual perform function (separating behaviour from a class by hiding an implementation behind pointer is called the pimpl (pointer implementation) idiom [7], and is a well established concept). Structuring the relationship between event and action classes this way makes the implementation of events and actions fully independent to each other (meaning changes to one have no effect on the other).

Event and action creation and assignment to each other and to the correct modules happen in a separate setup function called once at initialization. Cross references between resources (e.g. references to the communication configuration resources from actions), are done through the second layer. The central class ScriptManager is implemented with the Singleton design pattern [7] making it possible to organize all resources behind a single reference, thus making it accessible to all objects. All modules found in the top layer can be accessed from the ScriptManager,

Singleton classes restricts object count to a single instance of the class, and makes this instance globally accessible. This is commonly done by making the constructor private (forbidding clients of constructing objects), and by offering a public static reference to the singleton object. Singletons are considered dangerous in multi threaded environments since access can be gained uniformly from all threads simultaneously.

5 Documentation

The following chapter is a standalone complete documentation of the GatewayScript. It details the present features along with instructions, guidelines and examples.

5.1 Statements

Statements are smallest element of the language, representing an action

Types

There are four supported types: byte, short, int and boolean. Byte is one byte in size, short is two bytes and an integer is four bytes. Boolean has two values: true and false.

Variables

Variables can be declared and optionally initialized in the following manner:

```
byte b;
short s = 42;
static int i;
boolean bo = true;
```

Adding the static keyword in front of a variables type, persists the variable's value between event triggerings. Static values are initialized on the first call of the event.

Created variables can be read and written to in statements and expressions.

```
i = s + 420;
```

Expressions

The language is capable of evaluating arithmetic and boolean expressions. Evaluated expressions infer a type from their contents, whereupon they can be used in the appropriate context.

```
int i = 3 + (4 * 5); // okay
int j = 3 < 4; // not okay, evaluated to be boolean
boolean b = 3 < 4; // okay
```

Comments

Comments can be written anywhere. There are two ways to write comments in the code:

```
// This is a line comment

/*
     This is a block comment
*/
```

5.2 Loops and Ifs

Loops

There are three kinds of loops, taken from the C language.

```
for(<assignment>; <boolExpression>; <statement>) {
     <statements>
}

while(<boolExpression>) {
     <statements>
}

do {

} while(<boolExpression>)
```

Ifs

If statements are executed if their boolean condition is evaluated to be true. An arbitrary number of else if statements can follow, and a single optonal else statement, executed when none of above the conditions were met

```
if(<boolExpression >) {
     <statements>
} else if(<boolExpression >) {
     <statements>
} else {
     <statements>
}
```

5.3 Configuration

Configurations stored as FieldbusGatewayConfigurations can be imported into the code with the following formula:

```
import <absolutepath\exampleCfg.fbgwcfg> as <modelId>
```

An arbitrary number of configuration files can be imported. To prevent name clashing, references can be restricted to a given models scope, identified by its *modelId.*

For example if there is a model imported as ModelA, and one as ModelB, and in both models there is a Frame called WheelFrame, the two frames can be accessed this way:

```
modelA::WheelFrame
modelB::WheelFrame
```

5.4 Events

Statements are organized by assigning them to events. When an event occurs its statements are executed in order. There are three types of events: Start, Timing, Message.

Events start with the *on* keyword followed by an event specific keyword and symbol. An event body follows, containing the statements enclosed by brackets.

Start event

The start event is triggered by the beginning of the program and executes only once. There can only be one start event per program.

```
On start {
        // Initial statements
}
```

Timer Event

Timer events are triggered periodically specified by a period assigned to them. Timer events are created with a name, and are inactive by default.

```
on timer exampleTimer {
        // Timer statements
}
```

The above code creates a timer object with the name *exampleTimer*. The following functions can be called by timer objects:

```
void start(uint16 aPeriod);
void start(uint16 aPeriod, uint16 aOffset);
void stop();
```

The start function starts the timer with the given period. Optionally an offset can be specified to delay the start. The default value of the offset is zero. The period and offset shall be specified in milliseconds. Subsequent starts to a timer will reset both the period and the offset. The stop function halts the timer, returning it to an inactive state.

For example our example timer can be started at the beginning of the program to be called once every ten milliseconds with a second delay;

```
On start {
        exampleTimer.start(10, 1000);
}
```

Message Event

Message events are triggered when their respective messages are recieved on CAN communication. CAN messages can be assigned to events in one of three ways.

Assignment to CAN ID: triggered when a CAN message with the given ID is received.

```
On message 123 {
     // Statements to execute when a message with the ID 123 is received
}
```

Assignment to CAN message: triggered when a CAN message referenced from the configuration model is received. Pressing ctrl+space after the message keyword will bring up all available Receive messages from the configuration.

```
On message ExampleCanMsg {
    /* Statements to execute when a message with the ID of ExampleCanMsg
    is received */
}
```

Assignment to all messages: it is possible to listen to all messages from a single event. There can only be one event listening to all messages per program. If a separate event is listening on a given ID, it will be prioritized over the listen to all event.

```
On message * {
// Statements to execute when a CAN message is received.
}
```

5.5 Built-in modules

Gateway functions are organized into and acessed through modules. A modules functions can be called by prefixing the function call with its respective modules name. Objects enclosed by < > are references from imported configurations. Pressing ctrl+space will bring up all available options.

Gpio

Calls to this module expect GPIO enums. The following enumerations exist in the module:

For representing the individual pins on the hardware.

GPIO_PIN values: GPIO_0, GPIO_1, GPIO_2, GPIO_3, GPIO_4, GPIO_5, GPIO_6, GPIO_7.

GPIO_DIRECTION values: GPIO_INPUT, GPIO_OUTPUT.

setDirection sets the specified pin to the specified direction.

```
void GPIO.setDirection(GPIO_PIN aPin, GPIO_DIRECTION aDirection);
```

toggleBit toggles the output value on the specified output pin.

```
void GPIO.toggleBit(GPIO_PIN aPin);
```

Can

Transmit transmits a CAN message with the specified ID. Can be called by can messages referenced from imported configurations.

```
void CAN.transmit(int aId);
void CAN.transmit(<CanMessage>);
```

Restbus

Disabling a restbus element will result in all contained elements being disabled, halting the transmission of all the affected signals.

RestbusStart

```
void RESTBUS.start();
```

RestbusStop

```
void RESTBUS.stop();
```

RestbusFrameDisable

```
void RESTBUS.frame(<Frame>).enable();
```

RestbusFrameEnable

```
void RESTBUS.frame(<Frame>).disable();
```

RestbusFrameTransmit

```
void RESTBUS.frame(<Frame>).transmit();
```

RestbusPduDisable

```
void RESTBUS.pdu(<Pdu>).disable();
```

RestbusPduEnable

```
void RESTBUS.pdu(<Pdu>).enable();
```

RestbusSignalDisable

```
void RESTBUS.signal(<Signal>).enable();
```

RestbusSignalEnable

```
void RESTBUS.signal(<Signal>).disable();
```

RestbusSignalSetValue

```
void RESTBUS.signal(<Signal>).setValue(int aValue);
```

Trace

TraceStart

```
void TRACE.start();
```

TraceStart

```
void TRACE.stop();
```

TraceGetSignalValue

```
int TRACE.signal(<Signal>).getValue();
```

5.6 Examples

Example 1

An example program demonstrating some features.

```
import "C:\\docs\\CanRxNode.fbgwcfg" as config

On timer OneSec {
     GPIO.toggleBit(GPIO_0);
     CAN.transmit(TpTxFlowControl);
}

On timer timer_500 {
     GPIO.toggleBit(GPIO_1);
     CAN.transmit(23);
}

On Message * {
     GPIO.toggleBit(GPIO_2);
```

```
		boolean timerFlag = false;

		if(timerFlag) {
			OneSec.start(1000);
			timer_500.start(500);
		} else {
			OneSec.stop();
			timer_500.stop();
		}

		timerFlag = !timerFlag;
	}

	On Message Cluster_GeneralInformation_N1_Frame_MSG {
		CAN.transmit(22);
	}

	On start {
		GPIO.setDirection(GPIO_0,GPIO_OUTPUT);
		GPIO.setDirection(GPIO_1,GPIO_OUTPUT);
		GPIO.setDirection(GPIO_2,GPIO_OUTPUT);
	}
```

Example 2

An actual program created to be a signal converter in a car where the Steering unit and the rest of the car communicated with two incompatible CAN databases. The gateway served as the connection between the two systems, and forwarded the relevant signals in the expected frames to the Steering unit. When a message containing relevant signals arrives, we read its signals' values and write it to the forwarded restbus signals' values.

```
import "C:\\Files\\EPAS_C1A.fbgwcfg" as EPAS_C1A
import "C:\\Files\\VEH_C1R.fbgwcfg" as VEH_C1R

// Rx from veh

On Message BRAKE_FWHEEL_R1_Frame_MSG {
		int FR;
		int FL;
		int speed;

		FR = TRACE.signal(WheelSpeed_F_R).getValue();
		FL = TRACE.signal(WheelSpeed_F_L).getValue();
		speed = TRACE.signal(VehSpeed).getValue();

		RESTBUS.signal(VehicleSpeed).setValue(FR);
		RESTBUS.signal(WheelSpeedFR).setValue(FL);
		RESTBUS.signal(WheelSpeedFL).setValue(speed);
}

On Message BRAKE_RWHEEL_R1_Frame_MSG {
		int RL;
		int RR;
```

```
        RL = TRACE.signal(WheelSpeed_R_L).getValue();
        RR = TRACE.signal(WheelSpeed_R_R).getValue();

        RESTBUS.signal(WheelSpeedRL).setValue(RL);
        RESTBUS.signal(WheelSpeedRR).setValue(RR);

}

On Message BRAKE_SWA_R1_Frame_MSG {
        int wheelAngle;

        wheelAngle = TRACE.signal(SteeringWheelAngle).getValue();

        // The read wheelAngle signal is larger than the one written to,
        // so we cut off the ends of the interval

        if(wheelAngle < 21966) {
             wheelAngle = 0;
        } else if (wheelAngle > 43566) {
             wheelAngle = 21600;
        } else {
             wheelAngle = wheelAngle - 21966;
        }

        RESTBUS.signal(SteeringWheelAngleRaw).setValue(wheelAngle);
        RESTBUS.signal(SteeringWheelAngleCorrected).setValue(wheelAngle);
}

On Message ECM_Control_RN1_Frame_MSG {

        // It is not necessary to declare separate variables
        RESTBUS.signal(PowerTrainStatus).setValue(
        TRACE.signal(EngineStatus).getValue()
        );
}

// Rx from epas
On Message DONGLE_EPS_A1_Frame_MSG {
        int dongle1;
        int dongle2;
        int dongle3;

        dongle1 = TRACE.signal(EPAS_C1A::DongleEPS1).getValue();
        dongle2 = TRACE.signal(EPAS_C1A::DongleEPS2).getValue();
        dongle3 = TRACE.signal(EPAS_C1A::DongleEPS3).getValue();

        RESTBUS.signal(VEH_C1R::DongleEPS1).setValue(dongle1);
        RESTBUS.signal(VEH_C1R::DongleEPS2).setValue(dongle2);
        RESTBUS.signal(VEH_C1R::DongleEPS3).setValue(dongle3);
}


On start {
        RESTBUS.start();
        TRACE.start();
        RESTBUS.signal(SwaSensorInternalStatus).setValue(5);
}
```

6 Summary

The language in its current state is equipped with an editor (with validation and auto-complete), and generates valid executable C++ code. Compilation and downloading has to be done manually by the user. In the future both the editor and the compilation and downloading process will be integrated into Gateway2Hell.

Debugging is currently not supported by the language. Since the generated software runs independently from the desktop, this is impossible to implement. An alternative solution would be to build a separate interpreter that transforms GatewayScript code into a Java-based software that emulates the gateway. Since Java code can be debugged, this eliminates the need to design a debugging module. In this scenario users can run and debug simulations in the IDE before downloading it to the hardware.

Future features for the language to implement: FlexRay communication protocol and transport layers. Diagnostic and flashing protocols can be implemented on top of the language by the users. Optionally these can later be moved into the supported language features.

The language with its currently implemented features can be used to create restbuses, signal converters and network adapters.

Thanks to GatewayScript, applications of these types can be developed vastly faster, and require no gateway specific knowledge. Projects at the company can freely create software, without support from the Gateway development team.

Bibliography

[1] AUTOSAR: *Layered Software Architecture*

[2] Eclipse Modeling Framework: *Documentation*, https://eclipse.org/modeling/emf/docs (revision 17:20, 10 May 2017)

[3] FlexRay Consortium: *FlexRay Communications System Protocol Specification Version 3.0.1.* https://svn.ipd.kit.edu/nlrp/public/FlexRay/FlexRay%E2%84%A2%20Protocol%20Specification%20Version%203.0.1.pdf (revision 18:14, 10 May 2017)

[4] Texas Instruments: *Introduction to the Controller Area Network (CAN)* http://www.ti.com/lit/an/sloa101a/sloa101a.pdf (revision 18:14, 10 May 2017)

[5] Lorenzo Bettini: *Implementing Domain Specific Languages with Xtext and Xtend – Second Edition*

[6] Markus Voelter, S. Benz, C. Dietrich, B.t Engelmann, M. Helander, L. Kats, E. Visser, G. Wachsmuth: *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*

[7] Scott Meyers: *Effective C++*

[8] Vector: *Programming with CAPL*

[9] Xtend: *Documentation*, http://www.eclipse.org/xtend/documentation/index.html (revision 18:11, 10 May 2017)

[10] Xtext: *Documentation*, https://eclipse.org/Xtext/documentation/index.html (revision 15:15, 10 May 2017)