Graduation Report
Project Avro ComMA
April 5, 2020

Robin Kuiper

Summary

During the graduation project for the HBO-ICT education, the student has worked on a project commissioned by Thales Group. Thales uses Apache Avro to specify the API they use. Based on this specification, empty methods are generated and filled out by Thales developers.

However, they want to specify more than just the API: they want to specify the behaviour as well. This behaviour specification would be used to validate whether the program behaves as is intended. Interface Description Languages (IDLs) were researched to find a candidate with such functionality, and the result was Component Modelling and Analysis (ComMA), a proprietary IDL developed by TNO and Philips. However, ComMA doesn't support all features Apache Avro does.

This project involved preparing the transition from an IDL Thales currently uses, Apache Avro, to the ComMA IDL. For every feature of Thales' current IDL Apache Avro that isn't supported by ComMA, a feature proposal has been created for ComMA. These proposals have been approved by Thales and a proof-of-concept implementation has been created for all proposals.

Contents

1 Change log

Version	Changes	Date
0.1	Initial Version	2019-11-15
0.2	Rewrote Planning Added Appendix Planning Added Appendix Proposals Added ComMA change proposals section	2019-12-09
0.3	Split ComMA explanation and problem explanation into two sections Added glossary Moved planning from appendix to in-line and added comparison to original planning Rewrote process into a coherent story Added template code changed appendix item Added ComMA availability piece Expanding on Franca usability explanation Added ComMA start-up guide Appendix	2020-01-06
0.4	Moved introduction back down, expanded it, and added a Summary Renamed research sections, added ComMA to Franca comparisons Renamed Execution to Evaluation Moved ComMA proposals from appendix to main body Added feedback section and expanded proposal section	2020-01-08
0.5	Added table comparing Apache Avro, ComMA and Franca Added decision matrix for choosing between ComMA and Franca Updated ComMA start-up guide Added implementation planning Added documentation for range and Type annotations code additions Updated planning	2020-02-21
0.6	Updated type annotations section to reflect latest work Processed feedback from version 0.4 and 0.5 Added signature annotation implementation Full document read through and general clean-up Added Xtext Section Added Franca code examples Added story points to feature planning	2020-03-18
0.7	Fixes based on feedback from company mentor: Mention that visual DSL is possible with Xtext Expand upon the background for Thales' motivation for this change Added implementation for immutable, dynamic array & templates	2020-03-22
0.8	Misc spelling and grammar fixes Specified in section 8.3 that only one proposal will be discussed Added range to planning in section 9.1 Specified in the research section why no other candidates were viable Expanded the summary a bit Split implementation specification into feature description, process and result added monthly split headers to section 11	2020-03-30
1.0	Spelling, grammar & formatting changes after review Added short and long implementation Updated planning Expanded summary	2020-04-05

2 Glossary

Apache Avro An open source IDL developed by Apache used to define a class signature. 1, 4, 6, 11, 12, 13, 14, 15, 17, 18, 26, 31, 32

AQL Acceleo Query Language. 25

Behaviour How an interface processes and responds to a signal.. 1, 6, 7, 8, 9, 12, 15

Behaviour validation Method of verifying whether the behaviour definition is adhered to. 6, 7, 9, 12, 13, 15, 17, 25, 31

Behaviour definition Specification of the state flow diagram for a class signature, and timing requirements. 6, 7, 8, 9, 12, 14, 15, 16, 31

Class signature A list of methods for a class, that can be used as an API. 7, 11, 12, 14, 15, 22, 23, 27, 28, 29, 30

ComMA Component Modelling and Analysis. 1, 4, 6, 7, 8, 9, 10, 11, 12, 14, 15, 17, 18, 20, 22, 23, 25, 26, 27, 28, 29, 30, 31, 32, 33

DSL Domain Specific Language. 18

Flow diagram A (literal) diagram used to describe the states and state transitions. *see* state flow, 12

Franca An open source IDL used to define a class signature and behaviour definition. 4, 15, 16, 17, 18, 33

Function A method with optional input and output parameters, used in the description of an class signature. 7, 14, 15, 17, 23

IDE Integrated Development Environment. 18, 20, 21

IDL Interface Description Language. 1, 6, 7, 11, 12, 13, 15, 17, 23, 31, 33

Interface The definition of the behaviour of a class signature. 6, 7, 12, 14, 22

State The state a program can be in. *see* behaviour definition, 7, 8, 12

State transition The method of changing from one state to the other in a flow diagram. 7, 8

State flow The way a program transitions through states. *see* flow diagram, 8, 12

Type Defined and named data type. Can be primitive data types, a collection of data types, enumerations or arrays. 7, 14, 15, 23, 24, 26, 27, 28, 29

Xtext An open source programming language development framework. 4, 18, 19, 20, 21, 29

3 Introduction

During the graduation project for the HBO-ICT education, the student was working on a project commissioned by Thales Group. This project involved working towards replacing the currently used Apache Avro IDL with the ComMA IDL, a proprietary IDL developed by TNO and Philips. ComMA's feature set partially overlaps with Apache Avro, an open source IDL. Apache Avro and ComMA support specifying message contents and functions. In addition to this ComMA supports behaviour definition, which can be used to define behaviour in an interface.

Thales currently uses a modified Apache Avro as an IDL, with custom annotations enabling support for embedded and legacy features. They want to use ComMA with its behaviour validation capabilities, but ComMA needs to be expanded to support all features required by Thales that the modified Apache Avro supports. The project was centred around modifying ComMA to support all features of Apache Avro, and all of Thales' modifications.

This document contains information pertaining to the project progression, the process documentation, and intermediate and final results. The document is structured in a mostly linear fashion, progressing along with the progression of the project.

One comment about the document structure: this document starts with an explanation of ComMA, of which some basic understanding is required to read the rest of the document.

4 What is ComMA

This section contains an in-depth explanation of one of the major components of this report. At first the abstract idea behind ComMA will be explained, followed by an example project. Finally ComMA's behaviour validation method is discussed, and the availability of ComMA for interested parties.

4.1 Purpose

Component Modelling and Analysis (ComMA)[5] has two major components: an Interface Description Language (IDL) used to describe message types, class signature and their behaviour definition, and a tool set used to validate the behaviour definition. ComMA can be used to define message contents in a .types file, and a class signature in a .signature file. Additionally, ComMA can be used for behaviour definition in a .interface file, which is called an interface. The interface contains various states, with each state representing a possible status of the system. Moving between states is called a state transition. state transitions are defined based on a function defined in the class signature, and triggered by that function begin called. Triggers for state transitions can have certain requirements to activate, these requirements are called state transition guards. state transition guards check whether requirements are met using variables defined in the interface. During a state transition, certain actions can also be defined, such as changing the values in variables or defining the response to the function call which triggered the state transition. Lastly an interface can also define time limits for the duration between receiving a function call and sending its response.

The tool set of ComMA has multiple functions as well:

- Compare a generated log for behaviour validation to expected behaviour defined in the interface,
- Generate a UML diagram which displays the state flow diagram defined in the interface,
- Generate empty C++ classes based on the class signature, and
- Generate documentation based on in-line documentation, similar to Javadoc[4].

ComMA is designed as a client-server model, with the class signature and interface applying to the server that a client communicates with. This design is relevant for two types of messages that ComMA supports: signals and notifications. Signals are messages that the clients send to the server, where the server doesn't respond. Notifications are messages that the server sends to a client, where the client doesn't respond.

4.2 Code example

To understand ComMA, an example case file was used: defining the class signature and interface for a vending machine (server), which communicates with a user-facing client (client). This vending machine has water, juice, and cola in its inventory, can be switched on and off, and the vending machine can be bought from and restocked.

Figure 1 shows a .types file that defines records and enums for the vending machine project. Records and Enums get named, as do their children. Records can have children that are other Records or Enums.

Figure 2 shows a .signature file that is used to define the class signature. The functions within the class signature are divided into three categories: commands, which get a response (return value), signals, which are received but get no reply, and notifications, which are sent, and the receiver doesn't respond. Signals al notifications are similar, but the inverse of each other: from the perspective of one entity in a network, signals are received and notifications

```
record Product {
    productName name
    int cost
}

enum result {
    DELIVERED
    NOT_ENOUGH_MONEY
    NOT_ENOUGH_SUPPLIES
}

enum switchOnResult {
    SWITCH_ON_OK
    LOAD_PRODUCTS_FIRST
}
```

Figure 1: "VendingMachine.types" - ComMA object and enum definition

```
import "VendingMachine.types"
signature IVendingMachine

commands
result orderProduct(productName prodName)

switchedOnResult switchOn

signals

switchOff

notifications

inventoryInfo(int items)

outOfOrder
```

Figure 2: "IVendingMachine.signature" - ComMA messages definition

are sent, and neither are replied to by the receiver. Functions can have an argument, which is defined after the function name within the parentheses. Functions can also have a return value, which is defined before the function name.

The ComMA .interface file is used to define the program's behaviour, which encompasses a state flow description, time and data constraints, and behaviour definition for state transitions. Figure 3 shows the start of the .interface file, where variables are defined and initial values are assigned. These variables are used by state transition guards and for transition behaviour. Figure 4 shows multiple state transition definitions. State transitions can be executed from all states and the defined initial state will be the state at the start of execution. When a 'switchOn' command is received in the initial state and there is stock in the inventory, the current state will transition to state 'Operational'. Figure 5 shows time and data constraints for transitions. It specifies that the time between receiving a call to 'orderProduct' and sending the reply for that call may not take longer than 10 milliseconds.

```
import "IVendingMachine.signature"
interface IVendingMachine version "0.3"
  variables
  int credit

  int colaSupply
  int juiceSupply
  int waterSupply

  init
  credit := 0

  colaSupply := 1
  juiceSupply := 1
  waterSupply := 2
```

Figure 3: "VendingMachineSpec.interface" - ComMA variable definition

```
17 machine vendingMachine {
18
19   in all states {
20     transition
21       do:
22         inventoryInfo(colaSupply + juiceSupply + waterSupply)
23   }
24
25   in all states except Initial {
26     transition trigger: switchOff
27       next state: Initial
28
29   initial state Initial {
30     transition trigger:
31       switchOn guard:
32         (colaSupply + juiceSupply + waterSupply) > 0
33       do:
34         reply(switchOnResult::SWITCH_ON_OK)
35         next state: Operational
```

Figure 4: "VendingMachineSpec.interface" - ComMA state & behaviour definition

```
130 timing constraints
131 TR1 in state Operational
132   command orderProduct -[.. 10.0ms] ->
133     reply to command orderProduct
134
135 data constraints
136 variables
137 int returnedCredit
138 DC1 command returnMoney;
139   reply(returnedCredit)
140     where returnedCredit >= 0
```

Figure 5: "VendingMachineSpec.interface" - ComMA constraints definition

4.3 Behaviour validation

ComMA monitoring isn't done whilst the program runs (known as 'online monitoring'), but it is done post-run (known as 'offline monitoring'). Offline monitoring is done using logs from the program. This requires the server to log all communications into a file. This file is later read to verify the behaviour.

When verifying the behaviour, ComMA generates multiple report files based on the log:

- Statistical overview of time and data constraints
- Behaviour analysis results:
  - List of constraint violations
  - Transition and state coverage percentages

Offline monitoring is done with hardly any overhead, since writing logs to a file can be done in the background. It also enabled new or updated behaviour validation for program runs that happened in the past. A downside to offline monitoring is that any errors that occur during the run cannot automatically be dealt with, but will need to be analysed after the program has finished executing to see what caused the error, and how to solve it.

4.4 Availability

ComMA is a proprietary solution created by TNO-ESI, a Dutch research lab that focuses on embedded systems, in collaboration with Philips. Philips contributed to the project by developing the C++ generator featured in ComMA, and C++ specific grammar additions.

Currently ComMA is not publicly available, for free or otherwise. The only organisations with access to ComMA are those with existing ties to TNO-ESI. TNO-ESI has expressed interest in open sourcing ComMA, but this is pending legal and technical review by both TNO-ESI and Philips.

TNO-ESI and Philips are willing to let Thales use ComMA, and participate in the development of ComMA so it fits Thales' needs.

```
protocol VendingMachine {
    record Product {
        productName name;
        int cost;
    }

    enum productName {
        WATER, COLA, JUICE
    }

    enum result {
        DELIVERED, NOT_ENOUGH_MONEY, NOT_ENOUGH_SUPPLIES
    }

    enum switchOnResult {
        SWITCH_ON_OK, LOAD_PRODUCTS_FIRST
    }

    result orderProduct(productName prodName);

    switchedOnResult switchOn();
}
```

Figure 6: "VendingMachine.avdl" - Apache Avro IDL object, enum, and message definition

5 Project definition

This section explains the problem and objective definition, the end result and the quality requirements of the project. The problem and objective definition contain a description of the problem experienced by the client, and the intended result that will be used to solve the problem. Finally, the scope of the project is described.

5.1 Current situation

Thales provides sensor solutions which require server clusters for data processing. Inter-server message structures are currently formatted according to the Apache Avro IDL. This message structure defines messages and their contents.

The same case study is used as in section 4 to explain the way Apache Avro IDL works. A vending machine class signature is defined, which can be communicated with by a user-facing client. Figure 6 shows an .avdl file, defining records (objects), enums, and messages with parameters and return values. The message definitions can use the defined records and enums. Notable is the fact that, in contrast to ComMA, Apache Avro defines all these aspects in a single file. Using an .avdl file, template files for any supported programming language can be generated, which eases implementation and prevents minor errors. Apache Avro also uses annotations for some basic features, but doesn't limit annotation usage. Thanks to this Thales uses custom annotations to support legacy and embedded features.

One of the reasons Thales uses Apache Avro is to simplify the development process. A benefit of Apache Avro is that it manages the APIs of all the components and keeps the versions of the APIs used in sync. For example, a system of micro services need precisely defined APIs to communicate. In classic development structure, a technical document is created specifying the API. Once this is created, it's up to the developers to implement the API according to this document. When the API needs to be changed, the technical document is updated and the developers are tasked with updating each process.

This method has a lot of room for error. For example, a developer could forget to implement an API point, or incorrectly read the documentation and implement something not in accordance with the specification. If a version update of the API is released, each service needs to be manually verified whether it's still in accordance with the specification.

When using Apache Avro, these potential issues are mitigated. The API for every service is still defined as it is in the classical scenario, but implementation phases occur differently. The class signature templates are generated, and the developers are only responsible for the body of the generated class signature. This prevents issues with missing or incorrect APIs. When a

new version is released, the Apache Avro generator can generate the new templates, and the tooling can identify which implementation doesn't comply with the specification and needs to be updated.

5.2 Problem

Apache Avro helping with implementing APIs and managing the different versions isn't enough. An API simply defines the input and output of a method. What the method does can still differ, without Apache Avro tooling warning the developers. A more complex definition can be an acceptable trade-off, if this provides a more specific API by also defining the behaviour. The downside that comes with dealing with a more complex definition can be managed by having just a small team responsible for the development and upkeep of this API specification. The upside presented by a more specific API can prevent large-scale trouble when implementing the proposed API with a large team of developers.

To enable this, Thales wants to be able to define, monitor, and verify system behaviour. Two examples of system behaviour are the state flow and the system call timings. Based on the system behaviour definition, Thales wants to be able to deduce whether the functioning system adheres to pre-specified requirements, also known as behaviour validation. These requirements are:

- correctly following the flow diagram that specifies what state a system must be in at that moment, and
- time limits that specify how long one or multiple calls may take to complete.

Thales also wants to expand the range of target languages generated. Currently they use a custom generator to generate C++ header and class files based on the .avdl files. They want the generator to support other output languages, and this could be a good opportunity to enable that goal.

5.3 Desired situation

The behaviour definition system that Thales wants already mostly exists, in the form of ComMA. In ComMA you can define message contents and functions just like in Apache Avro, although not as extensively as in Apache Avro. Additionally ComMA allows you to define an interface which contains the behaviour definition.

Thales specified ComMA as the best tool to solve their issues, but also requested researching alternative tools, to make sure there is no better alternative available.

Whatever tool will be chosen, it will be implemented as a separate system, since adapting Apache Avro isn't an option. Using two separate systems will mean that files containing the behaviour definition need to be changed in two places, once for Apache Avro, and once for the chosen tool. This dual system would be very error-prone, and these simple mistakes cost time and money. Therefore Thales considers it preferable to phase out Apache Avro entirely, and rely on the chosen tool to define both the class signature and the behaviour.

If the chosen tool currently doesn't support all features of Apache Avro, and Thales does not require every feature of Apache Avro, the objective is to:

- Research which Apache Avro features are missing in the chosen tool,
- Discuss which Apache Avro features are required by Thales in the chosen tool, and
- Expand the chosen tool to support all missing features required by Thales of Apache Avro.

Once completed, the end result should be an IDL grammar that can be used to describe any class signature currently described using Apache Avro, and the ability to expand upon the class signature with a behaviour definition.

5.4 Scope

- The focus of this project is on re-defining the chosen tool's grammar to support Thales' needs, mainly by porting over select Apache Avro features.
- Implementing all the grammar changes isn't expected for this project. However, a proof of concept is required, with more than 50% of the proposals implemented.
- A functioning behaviour validation implementation isn't required in the deliverable, this would be a separate project. This project's focus is on finding and refining and IDL that suits Thales' needs.

6 ComMA alternatives

In this section, alternatives to ComMA are researched. First, an overview is presented of the requirements that need to be met. Subsequently alternatives are researched and listed. Finally, these alternatives are compared and a suitable candidate is selected.

6.1 Requirements

Currently Thales uses Apache Avro, so its feature set needs to be supported at a minimum. Some Apache Avro features aren't used and are excluded from this overview. A complete overview of all features ComMA offers can be found in appendix B. As in the rest of this document, this overview uses ComMA nomenclature: 'class signature' is the collection of methods, and 'interface' is the behaviour definition of that class signature.

Apache Avro supports importing other Apache Avro files, which can contain defined types and/or a defined class signature. These defined items can be used in the definition of new types and class signatures. It supports defining a namespace for the Apache Avro file, which can help differentiate similar types and functions in different files. @-Style annotations can be used to modify types and functions, and there isn't a definitive list of annotations: you can use any name for an annotation, and pass any value, since the handling of annotations is done at the generator level, not in the grammar. Lastly there is no official documentation system, but Thales extracts javadoc-style comments from Apache Avro files to generate HTML documentation.

Types

Primitive Types

int 32-bit signed integer

long 64-bit signed integer

float Single precision (32-bit) IEEE 754 floating-point number

double Double precision (64-bit) IEEE 754 floating-point number

boolean A binary value

byte Sequence of 8-bit unsigned bytes

string Unicode character sequence

null No value

Object Types

record A collection of fields, each field is a type

enum A collection of symbols

array A collection of items, all of the same type

fixed A collection of bytes, with a declared size.

Class signature
Apache Avro supports annotations that apply to the entire class signature or just one function. You can also annotate argument types and return types.

Functions

name Custom identifier for the function

arguments Arguments given when calling the function

return value Resulting value from calling the function

error Error returned when an function call goes wrong

signal Receive a function, without needing to reply

6.2 Alternatives research method

To get a list of alternatives, the most popular IDLs were sourced from Wikipedia[6]. IDLs were additionally sourced from the results of the internet search engine DuckDuckGo[1], querying the first 50 results of the terms IDL and Interface Description Language. For each item on this list it was manually verified whether it supported the functionality required for this project. This was done by reading the documentation and comparing the documented features to the requirements.

This method of researching IDLs did not uncover ComMA, since it is a closed-source solution. This could mean there are other proprietary solutions created by businesses that haven't been widely publicised. However, since these solutions aren't publicised, it's not a realistic expectation to find them.

This search resulted in only one viable candidate: Franca[1]. Franca is the only candidate that came close to meeting the requirements put forth by Thales. All other options found were at varying levels of compatibility in its ability to define types and a class signature. But none of these options, besides Franca, includes any kind of behaviour modelling.

6.3 Franca

Franca is a framework that uses its own custom IDL. In this IDL class signature and behaviour can be defined. The Franca framework supports transforming files based on a supported IDLs to a Franca file. Based on a Franca file, Franca can generate class signature and behaviour validation code. The Franca IDL is intended to be a superset of all IDLs. Transformations take place in two different ways: one can transform from an IDL to the Franca IDL, or transform from the Franca IDL to a different IDL. Generation is done based on the Franca IDL file.

However, whilst Franca is a framework to build transformers and generators with, it does not provide many of these components by default. The only first party components are transformers for the IDLs D-BUS, OMG-IDL and Google Protobuff. There are third party generators, the most promising of which is the open source Joynr [2], which can generate C++, Java and JavaScript class signatures based on a Franca IDL definition, but Joynr doesn't generate any behaviour validation code.

Franca's feature set is mostly a superset of Apache Avro and ComMA. An example of Franca type definition is shown in figure 7, a partial class signature in figure 8 and some behaviour definition in figure 9. The features that are required but are missing are mostly behaviour validation features that ComMA does support. Most importantly, using ComMA one can extensively specify transition actions, but many of these actions aren't definable using Franca. Defining data and time constraints, as is possible with ComMA, isn't possible at all with Franca. A complete overview of missing features can be found in appendix B.

[1] `http://duckduckgo.com`

```
1  struct Product {
2      productName name
3      UInt32 cost
4  }
5
6  enumeration result {
7      DELIVERED
8      NOT_ENOUGH_MONEY
9      NOT_ENOUGH_SUPPLIES
10 }
11
12 enumeration switchOnResult {
13     SWITCH_ON_OK
14     LOAD_PRODUCTS_FIRST
15 }
```

Figure 7: "VendingMachine-types.fidl" - Franca type definition

```
1  import model "VendingMachine-types.fidl"
2  interface VendingMachine {
3
4  method orderProduct {
5    in { productName prodName }
6    out { result result }
7  }
8
9  method switchOn {
10   out { switchOnResult result }
11 }
12
13 method switchOff fireAndForget {}
14
15 broadcast inventoryInfo {
16   in { UInt32 items }
17 }
18
19 broadcast outOfOrder {}
20
21 }
```

Figure 8: "VendingMachine-interface.fidl" - Franca messages definition

```
17 import model "VendingMachine-interface.fidl"
18 contract {
19   PSM {
20     vars {
21       UInt32 colaSupply;
22       UInt32 juiceSupply;
23       UInt32 waterSupply;
24     }
25
26     initial idle
27     state idle {
28       on call switchOn [colaSupply + juiceSupply
29         + waterSupply > 0] -> Operational
30     }
31
32     state operational {
```

Figure 9: "VendingMachine-contract.fidl" - Franca state & behaviour definition

6.4 Usability

One of the incompatibilities with Franca is that it allows a function to define multiple out parameters. One of the design goals is language independence, as specified in section 8.1. Multiple output values is a feature not natively supported by modern languages such as Java and C#, therefore to comply with the design goal it will require this to be solved by using a wrapper. In C and C++ multiple output values is a common design paradigm, by passing a pointer to the values as an argument. Thales has deprecated this method of getting output values, and as such doesn't want this feature in the IDL.

To use Franca, it would need to be expanded to support all the missing features from ComMA and Apache Avro. To support existing Apache Avro files, an Apache Avro to Franca transformer can be built. For code generation the existing Joynr generator can be used as a starting point, but it would need to be expanded to support behaviour validation. However, this is a moot point since Thales has indicated they want to use their own code generator, regardless of which framework (ComMA or Franca) is used.

ComMA and Franca are both missing features used by Thales with Apache Avro. Since the code generator still needs to be adapted for both ComMA and Franca, this doesn't impact the decision. Meanwhile Franca supports transforming existing Apache Avro files, which means that these files won't have to be re-written for the new tool. Franca's documentation is also far superior, with ComMA hardly having any documentation. These advantages caused my initial advice to be to use Franca.

To make comparing the viability of ComMA and Franca more measurable, a decision matrix was created. This decision matrix can be found as appendix B.5. Each feature of Apache Avro, ComMA and Franca that is considered relevant by Thales was listed. With input from the company mentor each feature had a weight assigned according to the MoSCoW scale. *Must* features were weighed very heavily, since these features represented the bare minimum for a usable product.

For every feature, every language had a score assigned from zero to one, where zero means the feature is completely unavailable, and one is the feature is entirely present. The feature weight and score are multiplied to get a total compatibility score for that feature. Each total score for a language is summed up to get the final score, which can be compared to the other languages to quantify viability.

Based on this final score, Franca is the most viable candidate.

However, Thales currently values using ComMA due to the value of collaborating with TNO-ESI. Based on this additional value, they decided to choose ComMA over Franca.

```
1 VariableExpression:
2   variable1 = [Variable] "==" variable2 = [Variable]
3 ;
```

Figure 10: Example of Xtext rule definition with cross-references

7 Xtext

Since both ComMA and Franca are developed using Xtext, this section explains the Xtext framework. First, an overview of the framework is outlined. Then the workflow is described, along with the issues Xtext has. Finally competing technologies are discussed.

7.1 Overview

Xtext is an open-source software framework used for developing programming languages and Domain Specific Language (DSL) like Apache Avro, ComMA and Franca, supported by the Eclipse foundation. Xtext uses ANTLR[2] for generating a parser and lexer, but its strength is with the Eclipse integration. Languages defined using Xtext automatically integrate into the Integrated Development Environment (IDE) with features like code completion, syntax highlighting, rename refactoring, custom help pages and more. This integration is very valuable, since it significantly speeds up development compared to developing without any IDE support. Xtext is coupled with Xtend[3], a Java dialect which compiles into Java. Xtext by default uses Xtend to write code like parser rules, tests and generators. The grammar is compiled to an object-based representation of the grammar, which the code can easily traverse. This method of traversal encourages Object Oriented (OO) grammar definition, with many small re-usable parts.

Another unique feature of Xtext is the fact that you can define cross-references in the grammar. For example, you can define that expressions can reference pre-declared variables. This is displayed in figure 10, which shows a comparison expression that compares two declared variables.

With Xtext, you can define your language using the Xtext grammar, combined with validation and scoping rules that are expressed using Xtend. Validation rules are used when certain requirements need to be met for grammar to be valid, like type checking when adding a type to an array. Scoping is used to define which reference is referable in the given location, for example in a function the scope could be all global and certain function variables.

7.2 Workflow

When creating a language, you start by creating a new project. This generates five fresh components, as shown in figure 11. The base component contains the grammar definition, validation and scoping rules, and any generators. The other components contain tests, IDE module definitions, UI code for those IDE models and UI tests.

The most important aspect is the grammar component, shown in more detail in figure 12. The contents of this component is a dedicated folder for validators, scoping. generators, and a folder for the grammar, build file, and two files of boilerplate code. The build file is the .mwe2 file, that generates code based on the grammar, validators etc.

The Xtext file contains some sample grammar, as shown in figure 13. It can be used to define 0 or more greetings. Notice that for each greeting the variable 'name' is assigned using the equals (=) symbol to the rule ID, which is a default rule that matches most text. This variable can be accessed and used in validators and generators when traversing the code. Similarly, the

[2] `https://www.antlr.org/`
[3] `https://www.eclipse.org/xtend/`

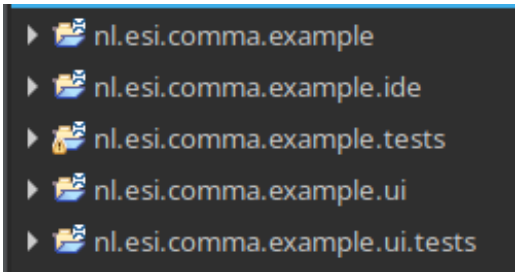


Figure 11: A new Xtext project, called 'example'

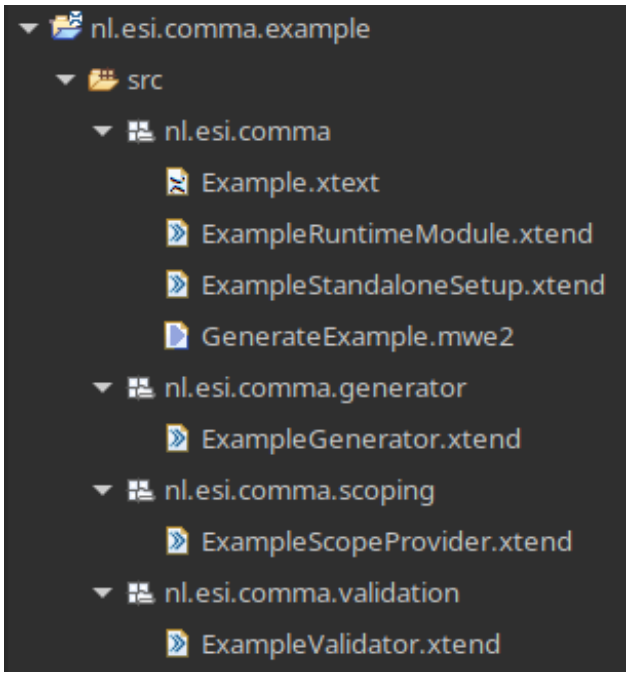


Figure 12: The contents of the grammar component

```
1 Model:
2         greetings+=Greeting*
3 ;
4
5 Greeting:
6         'Hello' name=ID '!'
7 ;
```

Figure 13: Xtext rules used to define greetings

greetings variable contains the greetings. This time, greetings is assigned using a plus sign and an equals symbol (+=), which means the variable it is assigned to will be a list. The final method of assigning a variable which is not displayed in this example, is using a question mark and an equals symbol (?=). This assigns a boolean value based on whether or not the rule is consumed.

A project like shown above can be used to define one component of your language. Once done, newly defined projects can use grammar from this project to build upon your code. This enables you to quickly build complex grammar structures with simple base components. A good example is creating expressions by using simple base components like defining types, defining variables, and assigning values to variables.

7.3 Issues

One of the major issues with Xtext is the split between grammar and validation files. In large projects, this causes developers to quickly lose overview of what rules are defined, and where they are defined. The way ComMA solved this is by adding a comment in the grammar file that specifies that there is a validation rule for the defined parser rule. However, this solution certainly has flaws: it's dependent on the developer remembering to add the comment, and the description of what happens during validation needs to be updated every time the validation rule is changed.

The second problem is related to the first: project organisation. As shown in figure 11, by default a new Xtext project creates five Eclipse projects. Since Xtext encourages creating lots of small components, this means creating lots of projects. And every Xtext project creates five Eclipse projects, which in the end really adds up. ComMA clocks in at over 100 Eclipse projects, which makes it hard to quickly find any part of the project.

The final problem is documentation. Documenting code in Xtext is made unnecessarily hard. When writing code for validators or generators in Eclipse, you can ctrl-click a class to go to its definition. In the case of parser rules, those definitions are generated Java classes. These generated Java classes contain generated documentation, an example being figure 14. The documentation in figure 14 is for the getName method user by the Greeting parser rule. The generated documentation is superficial, and explicitly states that if it isn't enough, more documentation should be written. One would expect that the documentation can be written along with the grammar in the .xtext file. However, there currently is no way in Xtext to generate documentation from the grammar file. What makes this even worse is that any manual documentation in the generated file is overwritten when the project is built using the .mwe2 file.

7.4 Alternatives

Since one of the valuable features of Xtext is the Eclipse integration, alternatives need to support similar levels of integration. The one candidate is JetBrains MPS (MetaProgrammingSystem), developed by the same JetBrains who develop the IntelliJ suite of IDEs. Because of this MPS is deeply integrated with IntelliJ.

```
1   /**
2    * Returns the value of the
3    * '<em><b>Name</b></em>' attribute.
4    * <!-- begin-user-doc -->
5    * <p>
6    * If the meaning of the '<em>Name</em>'
7    * attribute isn't clear,
8    * there really should be more of a description here...
9    * </p>
10   * <!-- end-user-doc -->
11   * @return the value of the '<em>Name</em>' attribute.
12   * @see #setName(String)
13   * @model
14   * @generated
15   */
16  String getName();
```

Figure 14: Generated documentation for the getName method in the Greeting class

Besides IDE integration, MPS and Xtext have a lot of structure in common. They can both define grammar, validation and scoping rules. However, the execution of these concepts is entirely different for MPS. JetBrains chose with MPS to go for a more Object Oriented approach to language definition. They define a parser rule like a class in a file, show in in figure 15. This only defines the data aspects of a rule, including inheritance of different rules, primitive values that are passed as properties, and child rules that it contains. How the end user will enter this data is defined in a separate file called an editor. Here you can specify the syntax of a statement, including static strings that are useless from the language developers perspective like brackets around the boolean expression of an 'if statement'. MPS also supports different ways of inputting data for the end-user other than typing text, like filling in a form, creating a flowchart-like diagram, or a math notation. This is great if you need to develop a language for non-tech users, who are used to tools like Microsoft Excel and Microsoft Visio for making forms and diagrams.

Xtext can do a similar thing: Xtext's modelling and generation framework, EMF (Eclipse Modelling Framework), is also used by graphical editing programs like Graphiti[4]. Because these programs use the same framework, they can be set up to work together to create, for example, a diagram-based language. However, this integration isn't native and requires manual configuration of the components.

MPS doesn't use plain-text files, and because of this versioning software like git can have issues. MPS can be represented as XML, and JetBrains offers an add-on[5] which helps with merging and building the XML files.

To get a feel for MPS, a link to a good walkthrough of an MPS project can be found at the bottom of this page[6].

[4] https://www.eclipse.org/graphiti/

[5] https://www.jetbrains.com/help/mps/version-control-integration.html#vcsadd-ons

[6] https://www.youtube.com/watch?v=0yIj5D6ORXs

```
concept IfStatement extends     Statement
                    implements  IContainsStatementList
                                IDontSubstituteByDefault

  instance can be root: false
  scope: none

  alias: if
  short description: <no short description>

  properties:
  forceOneLine    : boolean
  forceMultiLine  : boolean

  children:
  condition         : Expression[1]
  ifFalseStatement  : Statement[0..1]
  ifTrue            : StatementList[1]
  elsifClauses      : ElsifClause[0..n]

  references:
  << ... >>
```

Figure 15: An MPS concept, defining the attributes

8 ComMA change proposals

In this section, ComMA change proposals are discussed. First the requirements for the proposals are listed, then the feedback system is explained. After this a proposal is discussed, including which feedback was received and what was changed as a result.

8.1 Design goals

When proposing changes to ComMA, certain design goals need to be met:

- ComMA is intended to be programming language agnostic, so for every proposed change there needs to be a reasonable expectation of support in most mainstream programming languages.
- All language-specific notations, like specifying that during C++ generation the class signature needs to be generated as a C service, are left as annotations.

The intention for the first design goal is to maintain the ability to generate the class signature and interface in any mainstream programming language. The intention of the second design goal is that annotations will rarely be used, with the defaults being appropriate most of the time.

8.2 Feedback

Feedback on ComMA proposals is received through feedback meetings. The company mentor has selected multiple software architects that lead the teams that will use ComMA. This feedback group of software architects includes people with different software backgrounds. This is

Avro Implementation	ComMA Proposal 1	ComMA Proposal 2	ComMA Proposal 3	ComMA Proposal 4

record TestRecord {
@min(-12.1234) double a;
@max(13.432) double b;
@min(-3) @max(4) int c;
}
	record TestRecord {
	real min(-12.1234) a
	real max(13.432) b
	int min(-3) max(4) c
	}
		record TestRecord {
		real (-12.1234, maxReal) a
		real (minReal, 13.432) b
		int(-3,4) c
		}
			record TestRecord {
			real range -12.1234 .. max a
			real range min .. 13.432 b
			int range -3 .. 4 c
			}
				record TestRecord {
				real a = Range.atLeast(-12.1234)
				real b = Range.lessThan(13.432)
				int c = Range.closed(-3, 4)
				}

Figure 16: Range proposals, version 1

Avro Implementation	ComMA Proposal 1	ComMA Proposal 2

record TestRecord {
@min(-12.1234) double a;
@max(13.432) double b;
@min(-3) @max(4) int c;
}
	record TestRecord {
	real(-12.1234, maxReal] a
	real[minReal, 13.432) b
	int[-3,4] c
	}
		record TestRecord {
		real(-12.1234, max] a
		real[min, 13.432) b
		int[-3,4] c
		}

Figure 17: Range proposals, version 2

to support the goals for ComMA that all features are language-independent, as stated in section 8.1.

These meetings have proven invaluable, since the feedback meetings uncovered many faulty assumptions by both me, my company mentor and the selected software architects. There have been two feedback cycles in total.

8.3 Proposals

ComMA proposals appear in the following categories:

IDL All changes related to the ComMA IDL as a whole, or multiple components of the IDL.
Fields Changes to the definition of objects and their contents.
function Changes related specifically to function.
Language specific changes Proposals for ports of language-specific annotations.

In the proposal document (appendix A), all custom annotations used by Thales have proposed replacements. A lot of these proposed replacements are denied based on the fact that the to-be-replaced annotations represent old functionality, that is no longer required. Because of this, a total of seven proposals will be implemented, with all the proposed annotations combined counting as two proposals: type annotations and class signature annotations. To illustrate the process of creating a proposal, one proposal will be discussed in detail: the @range annotation replacement proposal.

Currently, @range is used to specify the range of values a variable can be. There were four initial proposals for adapting this to ComMA, as seen in figure 16. These proposals were discussed, and it was agreed upon that proposal two, with min and max keywords was the preferred proposal. An idea from one of the members of the feedback meeting was to use inclusive *[]* and exclusive *()*[7] brackets to specify whether a range was inclusive or exclusive.

Whilst changing the proposal I realised the keyword could be simplified, since using **real** implies that any min or max should be of type **real**. Two new proposals were made that were

[7] In accordance with ISO 31-11

shown during the second feedback round, as seen in figure 17. It was agreed that the simplified **min** and **max** keywords were the best option.

There were also proposals for feature addition, such as adding signed/unsigned types. Two example implementations were drafted, but during feedback the comment was made that this didn't meet the language agnostic requirement, as a major language like Java doesn't support unsigned variables. It was pointed out that the underlying goals of this feature were twofold:

- Limit an int to positive numbers, and
- Guarantee that an int can contain a certain size number.

The first issue was already solved by including the range feature. The second issue could be solved by adding additional keywords **short** and **long** for larger and smaller numbers. The exact size of these keywords could be decided later, and enforced at compile time.

This full list of proposals is extracted as a PDF from the internal wiki, and attached as appendix A.

9 Implementation

This section discusses how changes are implemented, and what has been implemented.

9.1 Method

Every change proposal needs to have its grammar edited, and tests added. It is decided on a case-by-case basis which generators need to be changed to support the added feature.

Based on experience implementing the first two features, I attributed story points to each feature, and assigned an expected start and end date. Every feature will need a different set of generators edited or removed, and this is factored into the assigned amount of story points.

The planned implementation time line is the following:

Start & end date	Feature	Story points
2020-01-20 - 2020-02-14	Range	Test implementation
2020-02-17 - 2020-03-03	Annotations type	8
2020-03-04 - 2020-03-06	Annotations class signature	1
2020-03-07 - 2020-03-12	Immutable	3
2020-03-13 - 2020-03-16	Dynamic array	3
2020-03-17 - 2020-03-25	Templating	5
2020-03-26 - 2020-03-27	Add short & long	1

9.2 Proposals implemented

This section will only display diff statistics and a description of the implementation process, since no permission was given to share the ComMA code.

Range

Feature description Thales wants to be able to specify a maximum value, a minimum value or both for ints and reals. This value limit would be used in code generation, documentation generation and potentially behaviour validation.

Implementation process A large amount of time was spent implementing the first feature, since it was a practice and research run. Everything learned during the implementation of this first feature was documented in the ComMA start-up guide. One of the first issues I ran into was that it wasn't entirely clear to me what I wanted to change, so I listed all generators that base their product on the ComMA file. Out of this list I created a list of generators I wanted to change: in this case this was only the documentation generation.

Changing the documentation generator proved difficult, as ComMA uses a library called M2Doc to generate documention. M2Doc generates a Microsoft Word .docx file, by changing a template .docx file. The template .docx file contains so-called 'field codes'. These fields are hidden by default, and contain query statements written in a unique query language called Acceleo Query Language (AQL)[8]. The query statements are used to define how to extract and display information. Using M2Doc, you can pass an object and a link to the template .docx file to generate the final documentation document.

It took asking for clarification from TNO, getting a new version of word and much trial and error to generate documentation about the specified range. The edited Template.docx reflects that work, although it can't be expressed through line changes since .docx is a binary file.

[8] 3.

Every generated file from the grammar that Xtext uses has generated documentation with a warning, that if that documentation isn't sufficient it should be replaced with proper documentation. I spent a lot of time trying to generate proper documentation, including asking for help on the Xtext forums and reading the Xtext source code. Eventually I gave this up as a bad job, and accepted that the Xtext framework, even though it states that you need to replace that documentation, simply supports no way to permanently replace the generated documentation.

Result Work done includes trial and error to edit the grammar (Types.xtext), limiting its use to just 'int's and 'real's (TypesValidator.xtend) and defining basic goodweather test to see if it properly parses (TypesParsingTest.xtend). With all code changes, a real effort has been made to document the changes in-line as clearly as possible, without retroactively documenting all other features. The test was in line with previous testing done (only goodweather tests) and currently has no way to automatically run (to, for example, test all components simultaneously).

Documentation generation now also adds range information, if relevant.

Diff

Added Range			
File changed	Lines Added	Lines Removed	Hours spent
doc.template/template/Template.docx	N/A	N/A	120 hours
types/Types.xtext	20	2	
types/validation/TypesValidator.xtend	12	0	
types/tests/TypesParsingTest.xtend	4	0	

Type annotations

Feature description Thales currently uses annotations in Apache Avro to specify custom behaviour. They want to also be able to use annotations in ComMA, for language-specific behaviour. These annotations can be used for all types, or for specific type definitions like custom types, records or enums. The values in records can also be annotated.

Implementation process This change was a lot simpler, since there was no research aspect left to be done. This meant it was a lot faster to implement, even though there were some technical difficulties. A choice was made to split up adding annotations to ComMA, by adding it per file type. This would keep complexity low per change.

One of the first choices was architectural: arguments to annotations can be expressions. But expressions aren't imported or used in Types.xtext. One solution would be to import and use expressions, but since expressions are a complex system, the effects of this change are hard to oversee. This issue was discussed with the company mentor, and he advised re-defining the basic types (numbers, true/false and arrays) and using these to avoid the complexity and dangers of expressions.

After this choice, I encountered a nasty bug that entirely broke defining types. This turned out to be related to ComMA using Xtext's 'cross-reference' system for types. Some work and experimenting led me to a proper solution.

Whilst adding annotations I ran into a huge technical issue that delayed me for over a day: every new version broke no matter what I did, and the issue didn't want to go away. Cleaning the build folder caused even more problems, with the build system reporting it was missing a file in the folder with generated files. Manually replacing that file from a known-good backup didn't solve anything, and neither did reverting all my changes. Eventually after a third restart of Eclipse suddenly everything worked again, but the original cause of the problem and what

solved it both remain a mystery. This event happening regularly could be a major issue for the planned schedule, so hopefully it doesn't happen often.

After I thought I was done, I realised I missed an important aspect: array validation. Validating that the contents of an array were all the same type proved to be complicated. For example, an array can't contain an int and a boolean array, so multi-level arrays need an in-depth check. An unexpected issue for array validation was that, for example, a triple-level array could have empty members in the second level, which complicated comparison.

Writing these array validation tests uncovered the fact that to run validation, it needs to be manually specified. Validation tests are specified in a separate folder, and a separate file was made for the annotation validation tests.

Result Grammar for type annotations was added, and validation for array arguments. Unit tests for the grammar and validation were added, with goodweather and badweather tests for each case. Since array validation is so complicated and tricky, a large part of the written unit tests are dedicated to this part.

After a significant amount of work, this is a solid foundation which should make implementing class signature annotations incredibly easy. Only the grammar was implemented, with no generator changes, since Thales has no interest in any of the existing generators/tooling from ComMA.

Diff

Added type annotations			
File changed	Lines Added	Lines Removed	Hours spent
types/Types.xtext	40	1	56 hours
types/validation/TypesValidator.xtend	197	90	
types/tests/TypesParsingTest.xtend	31	0	
types/validation/tests/Annotations.xtend	122	0	

Class signature annotations

Feature description Class signature annotations serve a similar purpose as type annotations, except these annotations apply to an entire class signature or a single method. Other annotations can apply to the method argument or return value. As with type annotations, what the annotation applies to is based on where it is written.

Implementation process Adding class signature annotations was an easy task, even though it touched a lot of files. The previous work on type annotations made this a breeze, with only the type definitions in the class signature requiring a change. The rest of the work was simply additive, including creating tests.

Since many generators use the class signature grammar, the change affected many other files. The changelog shows that many files required a minor tweak, but thanks to the work on type annotations this was quick and easy.

Result Grammar and tests are implemented. No validation code was needed, which resulted in relatively simple tests. Only the grammar was implemented, with no generator changes, since Thales has no interest in any of the existing generators/tooling from ComMA.

Diff

Added class signature annotations			
File changed	Lines Added	Lines Removed	Hours spent
signature/InterfaceSignature.xtext	6	2	3 hours
signature/formatting2 /InterfaceSignatureFormatter.xtend	12	12	
signature/validation /InterfaceSignatureValidator.xtend	2	2	
signature/tests /InterfaceSignatureParsingTest.xtend	37	11	
behaviour/generator/ResourceUtils.xtend	1	1	
behaviour/generator/SMCUtilities.xtend	1	1	
behaviour/generator /StateMachineDezyneGenerator.xtend	1	1	
behaviour/generator /GenerateCustomTypes.xtend	1	1	
expressions/scoping /ExpressionScopeProvider.xtend	1	1	
petrinet/generator /GenerateCustomTypes.xtend	1	1	
project/service/DocService.java	15	4	

Immutable

Feature description The immutable keyword is supposed to represent an unchangeable value, similar to const in C/C++ and final in Java.

Its function is a combination of two existing annotations: @constant was used to define a value that can be referenced throughout the project. @const was used to declare that the fields of a record, or the return value of a method may not be changed. It was also used to declare that the argument of a method will not be changed.

Both functions will now be used with the keyword Immutable.

Implementation process Implementing the immutable keyword went fairly smooth. During implementation of the grammar, I was too enthusiastic and added immutable type definitions. This was not part of the original proposal, and this work is kept separate and will be discussed with the company mentor.

During this task I also realised there was a way to run all existing tests simultaneously, instead of one by one. Since TNO/ESI have created some tests for each component, this would be useful when testing changes. Running these tests showed that many failed, mostly due to my previous changes. Most of these tests were fixed, which luckily didn't take too much time. Some were left broken, since they were parts of ComMA that Thales isn't interested in.

Result Grammar for Immutable, and validation code was created, The validation code checks immutable type usage in records. If a record defines an immutable field, all fields have to be immutable.

Existing broken tests due to unrelated changes (mostly type annotations) were also fixed. Only the grammar was implemented, with no generator changes, since Thales has no interest in any of the existing generators/tooling from ComMA.

Diff

Added immutable keyword			
File changed	Lines Added	Lines Removed	Hours spent
types/Types.xtext	8	4	12 hours
types/validation/TypesValidator.xtend	27	9	
types/tests/TypesParsingTest.xtend	35	0	
types/tests/validation/Immutable.xtenc	58	0	

Dynamic array

Feature description When declaring an array, Thales would like the distinction between a dynamic size array and a static size array to be more explicit. This is a purely syntax change.

Implementation process This implementation wasn't complex since it was purely a grammatical change, no validation needed. However, I ran into the issue of failing compilations, similar to the issue I had with the implementation of type annotations. Since that issue resolved itself after a day with many cleans, rebuilds and restarts of both Eclipse and Windows, this strategy was repeated. After trying multiple hours I gave up and worked on documentation for the rest of the day. When the next day the issue persisted, I took a good look at the error log, experimented a bit with building old build and realised the issue was due to a bug in Xtext. I resolved this by creating a workaround, which solved the issue.

Result The grammar for dynamic arrays was added, plus relevant tests. Static arrays were changed so they now have to specify size, as no size was the previous way to specify dynamic arrays.

Only the grammar was implemented, with no generator changes, since Thales has no interest in any of the existing generators/tooling from ComMA.

Diff

Added dynamic arrays			
File changed	Lines Added	Lines Removed	Hours spent
types/Types.xtext	6	2	8 hours
types/tests/TypesParsingTest.xtend	35	1	

Templating

Feature description Templates was a new feature request from Thales: being able to declare a class signature that uses a common object, without being specific about the object. This is similar to how Java's List definition uses a template object for most methods.

Implementation process Implementing templates accidentally took longer due to a mistake I made. The feature proposals aren't strict definitions, and are open to interpretation. The idea is that at each stage development (proposal and implementation) these are reviewed to see if all parties are happy with the current state. However, this vague feature proposal, written month ago, caused me to at first misunderstand my intentions with this proposal. I wrote validation code which checked if each and every parameter and return type for the class signature is the template type. However, the next day I realised this validation is completely unnecessary. A good example is Java's generic List, which implements methods whose arguments and return types are simple primitives. So a new validation rule was implemented, which checked that the

only generic type allowed in a class signature is the generic type defined in the class signature header.

Result The result is grammar, validation rules and tests for templates. When declaring a new class signature, you can specify which template type is used. This template type is the only template type permitted in the class signature.

Only the grammar was implemented, with no generator changes, since Thales has no interest in any of the existing generators/tooling from ComMA.

Diff

Added templates			
File changed	Lines Added	Lines Removed	Hours spent
types/Types.xtext	8	1	
types/validation/TypesValidator.xtend	12	0	12 hours
types/tests/TypesParsingTest.xtend	22	0	
types/tests/validation/GenericTypes.xtend	50	0	
signature/Signature.xtext	1	1	
signature/validation/SignatureValidator.xtend	46	1	
signature/tests/SignatureParsingTest.xtend	37	0	
signature/tests/validation/ValidationTest.xtend	22	2	

Short and Long

Feature description The primitive types short and long need to be added. These function as int alternatives with different byte sizes. The exact sizes are unspecified and will be decided when implementing in the compiler.

Implementation process Implementation was very easy: ComMA already has a list of all primitives, so all this required was adding short and long to this list. After adding to this list, existing type tests need to be updated to also test for the existence of short and long. The range function, which previously only worked with int and real, needed to get updated validation that it also works with short and long.

Result The result is the ability to specify long and short when required to use a type.

Diff

Added short and long			
File changed	Lines Added	Lines Removed	Hours spent
types/Types.types	2	0	
types/validation/TypesValidator.xtend	3	1	1 hour
types/tests/TypesParsingTest.xtend	2	0	

10 Conclusion

This project is a start of the larger project of Thales being able to use behaviour definition with their IDL, and using that information for behaviour validation. A large set of IDLs were compared to create a list of viable IDLs. Out of a list ComMA was picked to use. This IDL was compared with the current requirements of Thales, and a list of proposed changes to ComMA was created. This list of changes was iterated upon until it was accepted by Thales. Meanwhile a developer guide for ComMA was written, so new developers could easily expand the IDL. Finally a few of these proposals were implemented as a proof of concept.

The final deliverables for Thales are:

- This report, which contains a comparison of IDLs.
- A list of change proposals, which shows what has to change in ComMA's grammar to replace Apache Avro.
- A ComMA developer start-up guide, which will help new Thales and TNO-ESI employees develop ComMA.
- Proof-of-concept implementations for a number of proposals (as specified in section 9), which show that the proposals can actually be implemented.

11 Evaluation

This section describes how this project was executed. First the time line of the project is run through to create an image of what the project execution was like. After this the project schedule is discussed, and compared to the initial schedule.

11.1 Process

September At the start of the project, the first step was creating a first draft of the plan of approach. Whilst waiting on feedback for that first draft, I started learning how ComMA is used by following a guide and creating the sample project mentioned in section 4.2. When done with that, I moved on for some experimenting with Apache Avro by creating the same project in Apache Avro, as far as possible. This helped with understanding the limits of Apache Avro compared to ComMA. With feedback and more knowledge about ComMA and Apache Avro, the plan of approach was adjusted and finalised.

October An initial draft of requirements was made, and sent for review by the company mentor. Upon discussing this draft it became clear that my interpretation of the project differed from the company mentor's intentions with the project. My initial interpretation of the project was that I was expected to implement the changes I proposed. Additionally, I understood that I was expected to write a converter from Apache Avro to ComMA, and a new online validation system for ComMA, as touched upon in section 4.3.

In reality, these implementations were way too large a task (and thus out of scope). The only task that was expected of me was to adapt ComMA to support the features required by Thales. One major issue was that this new objective had no code aspect, something certainly required for a successful graduation project.

A week later, a compromise was reached, and it was agreed that some proof of concept implementations of the proposals would be done, in ComMA. As a result the existing documentation needed to be updated to reflect the new project definition.

November Once the documentation was updated, I received and started working with the ComMA source code. The initial steps were just simple exploratory modifications to understand how the ComMA code base is structured. There was an initial delay with this experimentation because of set-up issues that required help from TNO-ESI to resolve.

Once those issues were resolved, I started trying a basic modification of ComMA to get to know the code base. It took about a week to get a proper understanding of the code base. Once I understood the technology stack used by ComMA, and how the project was organised, I could start making informed modification proposals. A first draft was started of ComMA change proposals that would implement missing Apache Avro functionality.

December This initial draft took about two weeks to finish. A feedback meeting was arranged with software architects that lead the teams that would use ComMA. This feedback group included people from different software backgrounds, since one of the goals for ComMA, as stated in section 8.1, is that all features are language-independent.

In the period before the meeting I spent my time creating a ComMA developer guide. Since getting ComMA to work was a troublesome journey, and hardly any documentation exists for ComMA, I thought I could use my time spent researching ComMA to create a developer start-up guide. This would help structure the research and support other developers in learning how to develop ComMA. TNO-ESI also expressed interest in such a document. It is appended to this document as appendix C.

Two meetings took place over a period of two weeks, during which the entire first draft of the proposal was discussed. A lot of valuable and surprising feedback was received, not just

surprising for me but for the employees as well. This is thanks to all attendees having such diverse software backgrounds, and such diverse needs from this project.

January Once I came back from the Christmas holiday, I was quickly advised based on my concept graduation report to delay my project. I took that advice and started editing my report based on the feedback I had received along with the delay advice. This took about three weeks.

Once done with that, I started implementation 'properly': with the extra time, I could actually solve the underlying issues instead of adding grammar features in a quick and dirty fashion, which would be unmaintainable long-term. Implementing slower but higher quality solutions prevented creating issues down the road.

Feburary/March The next two months were an equal split between improving the documentation, and implementing feature proposals.

11.2 Schedule

This project is iterative, and part of an iterative project is the evolution of the schedule. During this project the schedule was updated on a weekly basis to reflect all changes in the project. Figure 18 shows both the current schedule, and the schedule from the final Plan of Approach document. The block "Preliminary research" in the PoA schedule has been split up into "Franca research" and "ComMA Code research".

The schedule is divided up into blocks that represent one deliverable each:

- Graduation Report: This graduation report
- Plan of Approach: The plan of approach
- Franca Research: The report on IDLs in section 6
- ComMA Code Research: The ComMA start-up guide
- Adaptation Research: ComMA adaptation proposals
- Prototyping: Implementation of some of the change proposals, mentioned in section 9
- Presentation: The graduation presentation

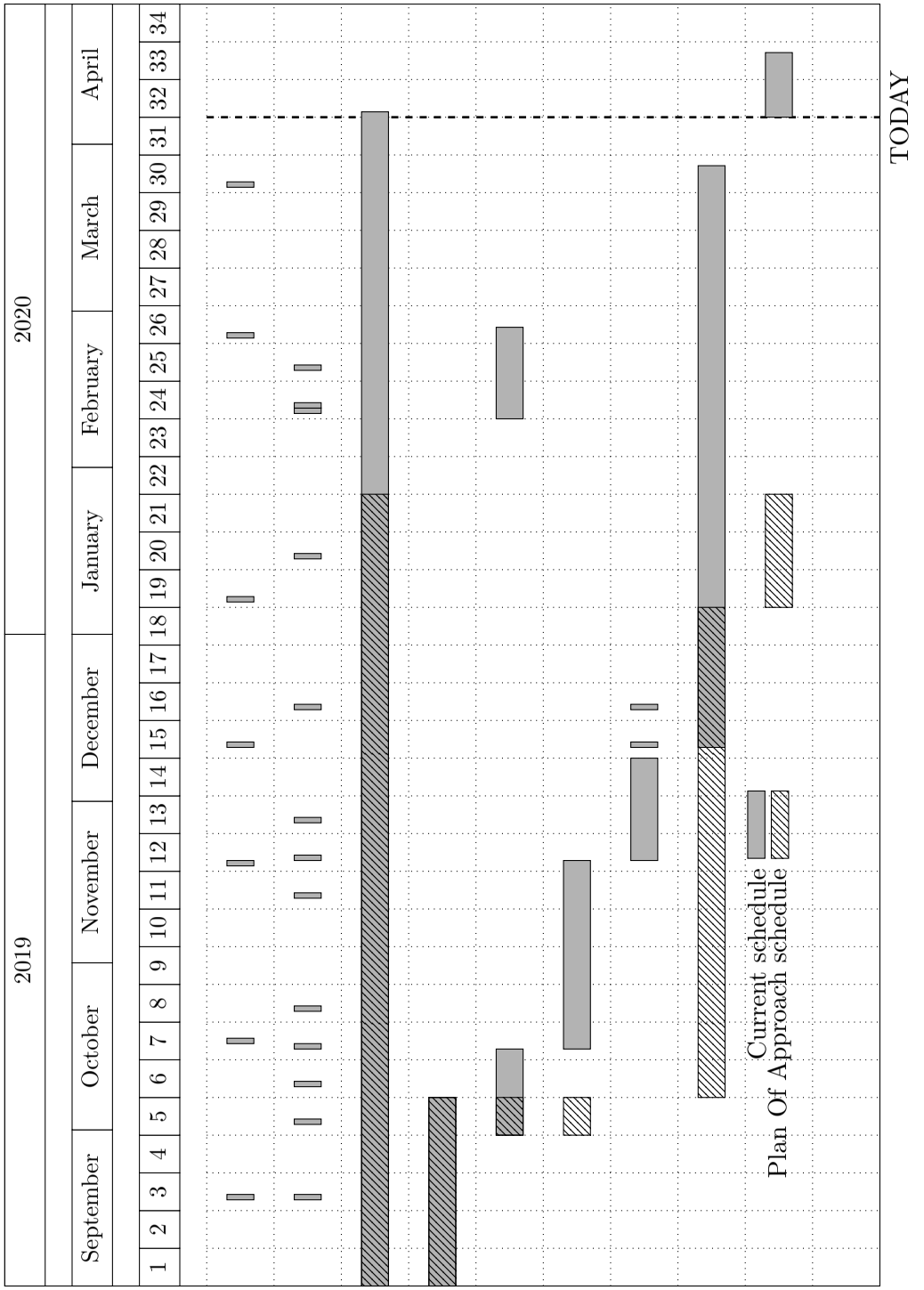


Figure 18: Schedule overview

References

[1] Klaus Birken. *Franca Website*. 2016. URL: https://web.archive.org/web/20180718093806/https://franca.github.io/franca/ (visited on 10/23/2019).

[2] Bmwcarit. *Joynr*. 2019. URL: https://github.com/bmwcarit/joynr/ (visited on 10/24/2019).

[3] Eclipse. *Acceleo Query Language*. 2020. URL: https://www.eclipse.org/acceleo/documentation/aql.html (visited on 03/04/2020).

[4] Oracle. *ComMA*. 2020. URL: https://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html (visited on 01/04/2020).

[5] TNO-ESI. *ComMA*. 2020. URL: http://comma.esi.nl/ (visited on 01/04/2020).

[6] Wikipedia contributors. *Interface description language — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Interface_description_language&oldid=911211589. [Online; accessed 22-October-2019]. 2019.

Appendix

A ComMA Proposals

ComMA Overview

ComMA introduction

ComMA is an IDL like Apache Avro. It can be used to define interfaces, like Avro, and it can also be used to define the behavior of those interfaces.

ComMA Change Proposals

In its current state, ComMA doesn't support all the features Avro supports. This wiki page documents those missing features, and proposed implementations.

One of the design goals is for all ComMA IDL syntax to be language independent. language specific features, like specifying a value is a pointer, should be done using annotations. The intention is that the defaults are good enough that annotations should be rarely used.

All keywords, but existing and proposed, are bolded for emphasis.

Protocol Annotations

Marker interface

In ComMA the interface methods are defined in a separate file. I propose treating it as marked as interface if an empty file is created.

Function	Avro Annotation	Avro Implementation	ComMA Proposal

If a protocol that contains only types and no methods gets marked as an interface, it still generates as an (empty) interface.	@MarkerInterface	@marker-interface (true)
		protocol MarkerInterface {
		}
			import "Marker.types"
			signature "IMarker"

Enum Annotations

Function	Avro Annotation	Avro Implementation	ComMA Implementation

Dynamicaly allocated enum values		**enum** EnumTestWithValues {
		val_field1,
		val_field2,
		val_field3,
		}
			enum EnumTestWithValues {
			val_field1
			val_field2
			val_field3
			}

Specified enum values	@EnumValues	@EnumValues(["val_field1=1","val_field2=3","val_field3=5"])
		enum EnumTestWithValues {
		val_field1,
		val_field2,
		val_field3,
		}
			enum EnumTestWithValues {
			val_field1 = 1
			val_field2 = 3
			val_field3 = 5
			}

Constant Annotations

const

Const is a concept that can be applied to global variables and to parameters.

In C++ this can also be applied to functions, marking them as 'const-safe' (you can pass const arguments).

Since both C++ and Java know the concept of 'immutable', that is used as the keyword for this concept.

Function	Avro Annotation	Avro Implementation	ComMA Proposal

Const values & parameters	@const	**record** TestRecord {
		string @const(**true**) FIELD;
		}
		
		@const(true)
		string name();
		
		boolean equals(**string** @const(**true**) name);
			record TestRecord {
			immutable string FIELD;
			}
			
			immutable string name();
			
			bool equals(**immutable string** name);

A conscious decision has been made to have the const field for a method be part of the ComMA syntax, even though it is C/C++ specific. This choice is made due to it being a more consistent experience compared to it being an annotation.

constants

Constants differ from const because it refers to an IDL-level definition. An @constants defined field can be referenced through the entire .AVDL file.

In the proposal, the declared variables use the **immutable** keyword like with @const. This declaration is done in the *.types* file, so it can be referenced in the *.signature* and *.interface* files.

Function	Avro Annotation	Avro Implementation	ComMA Proposal

Defined constants can be referenced throughout the IDL.	@constants	@constants(**true**)
		record constants {
		  CDT.Float LIGHT_SPEED = "299706720";
		  CDT.Float PI = "3.14159265";
		}
			immutable real LIGHT_SPEED = 299706720
			immutable real PI = 3.14159265

Field Annotations

Ranges

For this proposal, the keywords **min** and **max** must be added. These keywords will represent the largest and smallest possible values.

The inclusive **[]** and exclusive **()** brackets are used to signify whether a limit is inclusive or exclusive.

Function	Avro Annotation	Avro Implementation	ComMA Proposal

Limit the values to the specified minimum or maximum value	@min
	
	@max
		record TestRecord {
		  @min(-12.1234) **double** a;
		  @max(13.432) **double** b;
		  @min(-3) @max(4) **int** c;
		}
			record TestRecord {
			  real(-12.1234, **max**] a
			  real[**min**, 13.432) b
			  int[-3,4] c
			}

Arrays

static

Arrays are a built-in feature of ComMA, including specifying a size where required. As such, specifying a size could be a way to specify a static array.

Since dynamic arrays are significantly different than static arrays, a new keyword is added to differentiate the two: **list**. If the **list** keyword is used to declare dynamic arrays, declaring arrays of unspecified sizes (**int[]**) needs to be disabled.

Function	Avro Annotation	Avro Implementation	ComMA Implementation	ComMA Proposal

Default, dynamic Array		**record** DynamicArrayTest {
		  array<**int**> array1;
		  array<**array**<**int**>> array2;
		}
			record DynamicArrayTest {
			  int[] array1
			
			  int[][] array2
			}
				record DynamicArrayTest {
				  list<**int**> array1
				
				  list<**list**<**int**>> array2
				}

Specify an array as static	@static	**record** StaticArrayTest {
		  @static(20)
		  array<**int**> fixedArr1;
		
		  @static([20, 30])
		  array<**array**<**int**>> staticArr2;
		}
			record StaticArrayTest {
			  int[20] fixedArr1
			
			  int[20][30] staticArr2
			}	

controlled-by

Since the **list** keyword is used to define a dynamic array, it is also used here.

Feedback has been given that it's preferable to keep using an annotation.

Function	Avro Annotation	Avro Implementation	ComMA Proposal

Specify a variable that sets dynamic array initial size	@controlled-by	**record** ControlFieldTest {
		  int size;
		  array<**double**> @controlled-by("size") array;
		}
			record ControlFieldTest {
			  int size
			  list<@controlled-by(size)> array
			}

control-field

Control-field variables are serialized, but not used in the interface. Their only function is specifying the dynamic array initial size.
Since control-field is implicit (its what controlled-by refers to) it doesn't need to get added to ComMA.

Function	Avro Annotation	Avro Implementation	ComMA Proposal

Specify the variable to be serialized, but not used in the interface..	@control-field	**record** ControlFieldTest {
		  int @control-field(**true**) size;
		  array<**double**> @controlled-by("size") array;
		}	

C/C++ Specific Annotations

Annotations

Since language specific modifiers are intended to be added as annotations (or something similar), a good annotation system is required. And because annotations are closely related to the documentation system, this is also displayed.

Some choices for ComMA annotations:

- Annotations are weakly types, so their name (and arguments) can be anything.
- Annotation options are optional.
- Argument types: **bool, int, real, string, array** (primitives)
- Annotations must start by specifying the language it relates to, if it only relates to one language. This is convention, and not enforced by the grammar.

One issue with the current solution is that in a *.types* file the difference between a file-level annotations (at the top of a file) and an annotation for the first item (e.g. a record) isn't clear: both are at the top of a file.

Function	Avro Implementation	ComMA Implementation	ComMA Proposal
Overarching annotations & documentation	@annotation(argument) **protocol** {		**import** "Example.types" @cxx-annotation(optionalArgument) **signature** IExample
method annotations & documentation argument annotations	@annotation(argument) **void** method(**int** @annotation(argument) argumentName)	/* * Method description * \param methodArgument description * \return returnvalue description */ **int** method(**int** methodArgument)	/* * Method description * \param methodArgument description * \return returnvalue description */ @cxx-annotation(optionalArgument) **int** method(**int** @cxx-annotation(optionalArgument) methodArgument)
record, enum etc annotation & documentation	@annotation(argument) **record** {	/* * record description */ **record** {	/* * record description */ @cxx-annotation(optionalArgument) **record** {

Argument Annotations

	Function	Avro Annotation	Avro Implementation	ComMA Proposal
1	C/C++ value types: pass by value C complex types: pass by pointer C++ complex types: pass by reference		**void** add(ComplexType c);	add(ComplexType c)
2	C: pass by ptr C++: pass by ptr	@ptr	**void** add(ComplexType @ptr(true) c);	add(ComplexType @cxx-ptr c);
3	C: ignored C++: pass by reference	@ref	**void** add(ComplexType @ref(true) c);	add(ComplexType @cxx-ref c)
4	C: ignored C++: pass by std::shared_ptr	@shared-ptr	**void** add(ComplexType @shared-ptr(true) c);	add(ComplexType @cxx-shared-ptr c)
5	C: ignored C++: pass by std::unique_ptr	@unique-ptr	**void** add(ComplexType @unique-ptr(true) c);	add(ComplexType @cxx-unique-ptr c)
6	C: ignored C++: pass by rvalue reference	@rval	**void** add(ComplexType @rval-ref(true) c);	add(ComplexType @cxx-rval-ref c)
7	C/C++: pass by value	@val	**void** add(ComplexType @val(true) c);	add(ComplexType @cxx-val c)

cxx-includes

Proposal 1 uses the original syntax of Avro, Proposal 2 uses ComMA syntax for array instances.

Function	Avro Annotation	ComMA Proposal
include specified header files for a record	@cxx-includes(["stdio.h","string.h"]) **record** Info {	@cxx-includes(["stdio.h","string.h"]) **record** Info {

cpf-alias-declaration

In the ComMA project file, type mappings can be defined. A major downside is that the mapping isn't readily available in the source file. Feedback has indicated centralising this info is very important, so this is kept as an annotation over the existing solution.

Function	Avro Annotation	ComMA Implementation	ComMA Proposal

alias a data type against an existing C++ class	@cpf-alias-declaration("AnyCxxClass")
	record DummyType {
		Type Mappings {
		CppMappings {
		DummyType -> "AnyCxxClass"
		}
		}
			@cxx-alias("AnyCxxClass")
			record DummyType {

cpf-default-as-c

Not needed anymore, wont be ported.

Function	Avro Annotation	ComMA Proposal
set default use language as C	@cpf-default-as-c(true) @namespace("examples") @version("1.0.0") **protocol** Service {	

cxx-use-return-values

Not needed anymore, wont be ported.

Function	Avro Annotation	ComMA Proposal
Generate interface to return values instead of error codes	@cxx-use-return-values(**true**) **protocol** Service {	

template-wrap

Not needed anymore, wont be ported.

Function	Avro Annotation	ComMA Proposal

	protocol types {
	@cxx-includes("string_wrapper.h")
	@template-wrap("CapStringWrapper<{}>")
	fixed CapStr(1);
	}	

ComMA Additions

Proposed items that currently don't exist in either Avro or ComMA, but would be useful.

Templating

The proposal uses the ComMA feature of defining types, and using the **generic** keyword to signal the generator to generate a generic interface. Since having a generic type in an object makes no sense, it would need to fail on adding this type to a record.

Function	ComMA Proposal

Template classes and methods function with a generic type It requires all types in an interface to be the same	*in the .types file:* **generic type** T *in the .signature file:* T[] combine(T[] array1, T[] array2)

Universal container

Inspired by C++'s any keyword.

Currently not deemed useful, because a signature with any would be very messy.

Function	ComMA Proposal

A container for all data types
 	
1. Useful as a field in an object
2. Useful as an argument in a method
3. Useful as a return type of a method
	in the .types file:
	record ExampleRecord {
	  string name
	  any container
	}
	
	in the .signature file:
	any getGreaterValue(**any** value1, **int** value2)

Signed/unsigned types

Proposal one is inspired by the Franca language, proposal two is inspired by C/C++.

Currently not added, with the note to use the terms **short int** and **long**. What these keywords mean (e.g. is short 16 bytes?) will be decided compile time.

Function	Type	ComMA Proposal 1	ComMA Proposal 2

Default data type is signed	Default (Signed)	**record** Example {
		  int field1
		
		  real field2
		}
			record Example {
			  int field1
			
			  real field2
			}

Unsigned data type	Unsigned	**record** Example {
		  uint field1
		
		  real field2
		}
			record Example {
			  unsigned int field1
			
			  real field2
			}

Short and long

Based on the feedback for signed/unsigned types the **short** and **long** keywords need to be added. These need to be added to ComMA.

Function	ComMA Proposal

Short type	**record** Example {
	  int field1
	
	  short field2
	}

Long type	**record** Example {
	  int field1
	
	  long field2
	}

Alias

Aliases are nice-to-have, and are documented here. Aliases for fields make sense, and already exist in ComMA. Aliases for methods could be added.

Feedback: The prime use case for aliases is to add an annotation to a type, that doesn't apply to the base type. Hence method aliases aren't required.

Function	ComMA Implementation	ComMA Proposal

Alias a field name	**record** Example {
	  string name
	  age age
	}
	
	type alias **based on** Example
	
	type age **based on int**	

Alias method name		**commands** exampleMethod(**int** argument) **method** alias **based on** exampleMethod

B Apache Avro, ComMA and Franca feature comparison

B.1 Types comparison

	Function	AVRO	ComMA	Franca	Explanation	Required?
Meta	Global Types annotation	Shared with Signature annotations		Shared with Signature annotations		yes
	Type annotation					yes
Primitive Types		int Currently solved with alias e.g. @alias Uint8_t	int	UInt8		yes
				Int8		yes
				UInt16		yes
				Int16		yes
				UInt32		yes
				Int32		yes
		long Currently solved with alias		UInt64		yes
				Int64		yes
				Integer	Int with optional specified range	yes
		bool	bool	Boolean		yes
		float	real	Float		yes
		double		Double		yes
		string	string	String		yes
		bytes		ByteBuffer		no
		null				no
Logical Types		decimal				no
		date				no
		time_ms				no
		timestamp_ms				no
Object types		record	record	struct		yes
		enum	enum	enum		yes
		array	vector	array		yes
		fixed	Bulkdata	Bytebuffer (can't specify size)		Yes, as a replacement of specifying int sizes
		map		map		no
		union (key always string)		union		no
		error			Record exclusively used for error messages	no
type inheritance			record	struct		no
				union		no
				enum		no
					When base class is defined as polymorphic, child classes can be passed & auto-converted to base class.	
				Polymorphic structs	Needs explicit definition, default = low overhead	no
custom types			type CustomType		CustomType handled in the generator, can be based on pre-existing type.	yes, grammar update more work

B.2 Class signature comparison

	Function	AVRO	ComMA	Franca	Explanation	Required?
Meta	Import Types	import avdl	import types	extends interface	import the secified types	yes
	Import Signature				inherit the specified signature, with all its RPCs	yes
	Specify subclasses			manages	Specify which signatures are managed by this signature. Required for certain IPC mechanisms (e.g. D-BUS)	no
	Version			version		yes
	Type Collection	Can define just types, not enforced	.types file	type collection	Can only define types (enforced)	yes
Annotations	Global Annotations	Shared with type annotations		Shared with type annotations		yes
	Signature Annotations					yes
	Argument Annotations			Deprecated, officially recommended to add it to @description		yes
	Return type annotations					yes
Attributes				default	An attribute is a way to easily define a value and getter/setter	no
				readonly		no
				writeonly		no
				noSubscriptions		no
RPC	RPC name					yes
	RPC Arguments	multiple	multiple	multiple		yes
	RPC return	single	multiple	multiple		yes
	RPC error					yes
	RPC Receive non-blocking	oneway	signal	fireandforget		yes
	RPC Send non-blocking		notification	broadcast selective		yes
	RPC Broadcast			broadcast		no
	RPC Overloading				Multiple methods, same name but differen arguments/returns	no

B.3 Interface comparison

	Function	AVRO	ComMA	Franca	Explanation	Required?
meta	imports		import signature		Franca signature in the same file	yes
	version		version			yes
	interface tied to signature		separate	Part of signature		separate - can model some basic parts once, use model multiple times
transition trigger			call	call		yes - Unclear what is required - will take time & experience to know
				respond		
			call	signal		
				broadcast		
				update		
transition	guard				Franca can compare certain values -> similar to ComMA's ability	yes
	manipulate variables					yes - But prefer to remove them -> prefer simple modeling
	if/else					
	non-determinism				Define multiple possible responses	yes
	send oneway					yes
	define response					yes
	transition inheritance		in all states (except)	Requires duplicate code	Same transition in multiple states	yes
Constraints	Time				Behavioural rules outside of transitions	yes
	Data		Sort of duplicate functionality with transition guards	Can be hacked in through transition guard		yes

B.4 Meta comparison

Function	AVRO	ComMA	Franca	Required?
Server-client model	Signature file per host	Signature/interface for server	Signature/interface for server or all hosts	yes - We want to have a server-client model, with some exceptions.
namespace	@namespace		package	Yes
documentation	Inline	Inline, doxygen	Inline, javadoc, html generation based on javadoc	yes - currently generate inline to HTML, want to continue generating to HTML
custom annotations	Requires arguments		Yes, but annotations are in comments -> not actually part of language	YES

B.5 Decision matrix

TYPES	Weight	Avro	Avro Total	ComMA	ComMA Total	Franca	Franca Total
Global Types annotation	40.00	0.80	32.00	0.00	0.00	0.60	24.00
Type annotation	200.00	1.00	200.00	0.00	0.00	0.60	120.00
UInt8	40.00	0.00	0.00	0.00	0.00	1.00	40.00
Int8	40.00	0.00	0.00	0.00	0.00	1.00	40.00
UInt16	40.00	0.00	0.00	0.00	0.00	1.00	40.00
Int16	40.00	0.00	0.00	0.00	0.00	1.00	40.00
UInt32	40.00	0.00	0.00	0.00	0.00	1.00	40.00
Int32	200.00	1.00	200.00	1.00	200.00	1.00	200.00
UInt64	40.00	0.00	0.00	0.00	0.00	1.00	40.00
Int64	40.00	1.00	40.00	0.00	0.00	1.00	40.00
Boolean	200.00	1.00	200.00	1.00	200.00	1.00	200.00
Float	40.00	1.00	40.00	0.00	0.00	1.00	40.00
Double	200.00	1.00	200.00	1.00	200.00	1.00	200.00
String	200.00	1.00	200.00	1.00	200.00	1.00	200.00
record	200.00	1.00	200.00	1.00	200.00	1.00	200.00
enum	200.00	1.00	200.00	1.00	200.00	1.00	200.00
array	200.00	1.00	200.00	1.00	200.00	1.00	200.00
fixed	5.00	1.00	5.00	1.00	5.00	0.00	0.00
type CustomType	40.00	0.50	20.00	1.00	40.00	0.50	20.00
SIGNATURE							
Import Types	40.00	0.90	36.00	1.00	40.00	0.90	36.00
Import Signature	40.00	0.90	36.00	0.00	0.00	0.90	36.00
Version	200.00	1.00	200.00	0.00	0.00	1.00	200.00
Type Collection	200.00	0.50	100.00	1.00	200.00	1.00	200.00
Global Signature Annotations	40.00	0.80	32.00	0.00	0.00	0.40	16.00
Signature Annotations	200.00	1.00	200.00	0.00	0.00	0.60	120.00
Argument Annotations	200.00	1.00	200.00	0.00	0.00	0.60	120.00
Return type annotations	200.00	1.00	200.00	0.00	0.00	0.60	120.00
RPC name	200.00	1.00	200.00	1.00	200.00	1.00	200.00
RPC Arguments	200.00	1.00	200.00	1.00	200.00	1.00	200.00
RPC return	200.00	1.00	200.00	1.00	200.00	1.00	200.00
RPC error	40.00	1.00	40.00	0.00	0.00	1.00	40.00
RPC Receive non-blocking	200.00	1.00	200.00	1.00	200.00	1.00	200.00
RPC Send non-blocking	200.00	1.00	200.00	1.00	200.00	1.00	200.00
INTERFACE							
imports	40.00	0.00	0.00	1.00	40.00	0.90	36.00
version	200.00	0.00	0.00	1.00	200.00	0.50	100.00
interface separate from signature	40.00	0.00	0.00	1.00	40.00	0.00	0.00
receive call	200.00	0.00	0.00	1.00	200.00	1.00	200.00
respond to call	15.00	0.00	0.00	0.00	0.00	1.00	15.00
receive signal	200.00	0.00	0.00	1.00	200.00	1.00	200.00
send notification	15.00	0.00	0.00	0.00	0.00	1.00	15.00
send update	15.00	0.00	0.00	0.00	0.00	1.00	15.00
guard	200.00	0.00	0.00	1.00	200.00	0.80	160.00
manipulate variables	5.00	0.00	0.00	1.00	5.00	0.80	4.00
if/else	5.00	0.00	0.00	1.00	5.00	1.00	5.00
non-determinism	200.00	0.00	0.00	1.00	200.00	0.00	0.00
send oneway	200.00	0.00	0.00	1.00	200.00	0.00	0.00
define response	200.00	0.00	0.00	1.00	200.00	0.00	0.00
transition inheritance	40.00	0.00	0.00	1.00	40.00	0.00	0.00
Time	200.00	0.00	0.00	1.00	200.00	0.00	0.00
Data	40.00	0.00	0.00	1.00	40.00	0.40	16.00
META							
Server-client model	15.00	0.40	6.00	1.00	15.00	1.00	15.00
namespace	200.00	1.00	200.00	0.00	0.00	1.00	200.00
documentation	40.00	0.20	8.00	0.60	24.00	1.00	40.00
custom annotations	200.00	0.90	180.00	0.00	0.00	1.00	200.00
		Avro total:	4175.00	ComMA Total:	4494.00	Franca Total:	4993.00

Weights:	
Must	200
Should	40
Would	15
Could	5

C ComMA start-up guide

ComMA Developer Start-up Guide

This document is a simple how to for starting to develop a change for ComMA. It tries to give a high-level picture of the structure of ComMA. As a living document, you are encouraged to expand it upon it.

Installation & Setup

Requirements:

- Internet Connection
- ComMA source code

Instructions:

1. Download Eclipse Oxygen - RCP and RAP Developers
2. Install Eclipse plugins:
   For each plugin, go to Help Install new software Work with:
   a. "http://download.eclipse.org/modeling/tmf/xtext/updates/releases/2.15.0" Select plugin "Xtext", follow install steps & restart
   b. "http://download.eclipse.org/releases/2018-09" enter "mwe" into filter select MWE SDK, follow install steps & restart
   c. "https://poosl.esi.nl/downloads/ide/updates/latest" select all, follow install steps & restart
3. Import Project into Eclipse:
   a. File Import General Existing projects into workspace
   b. Click Browse select your ComMA folder click next
   c. Deselect the following packages:
      - nl.esi.comma.help.standard
      - nl.esi.comma.product.philips
      - nl.esi.comma.product.standard
      - nl.esi.comma.project.standard
      - all nl.esi.comma.project.standard.* and nl.esi.comma.project.standard.*.*
      - all nl.esi.comma.rcptt.*
      - nl.esi.comma.xpect.interfaces
   d. Click Finish
4. Set ComMA's target
   a. Open nl.esi.comma.target/nl.esi.comma.oxygen.target, and wait until Eclipse is done with loading & preparing. Progress can be viewed in the bottom right corner.
   b. In the opened file, click 'Set as active target platform' in the top right corner. Wait until Eclipse is done.
5. Generate DSL files
   a. To solve build file issues, comment/remove line 5 and 12 from file 'nl.esi.comma.launch/GenerateStandardProject.mwe2.launch'
   b. open 'nl.esi.comma.launch' right click 'Comma_GenerateAll_mwe2.launch' run as 'Comma_GenerateAll_mwe2'
      A warning box titles 'Errors in Workspace' might pop up, in that case check the box 'Always launch without asking' and click 'Proceed'
      On failure: In case of generation failing (console displays a stack trace), do the following: for each of the following projects, in the specified order, rightclick the .mwe2 file <project>/src/<project>/Generate*.mwe2 run as MWE2 workflow, and wait until the console says 'done' before generating the files for the next project
      i. nl.esi.comma.types
      ii. nl.esi.comma.signature
      iii. nl.esi.comma.expressions
      iv. nl.esi.comma.actions
      v. nl.esi.comma.statistics
      vi. nl.esi.comma.behavior
      vii. nl.esi.comma.behavior.interfaces
      viii. nl.esi.comma.behavior.component
      ix. nl.esi.comma.behavior.system
      x. nl.esi.comma.inputspecification
      xi. nl.esi.comma.branchingscenarios
      xii. nl.esi.comma.dezyne
      xiii. nl.esi.comma.petrinet
      xiv. nl.esi.comma.scenarios
      xv. nl.esi.comma.sscfheader
      xvi. nl.esi.comma.traces
      xvii. nl.esi.comma.traces.capture
      xviii. nl.esi.comma.traces.events
      xix. nl.esi.comma.project
      xx. nl.esi.comma.project.philips
6. First start
   a. create custom runtime launch configuration:
      i. Run debug configurations Eclipse application right click 'New Configuration'
      ii. Set the name of the configuration

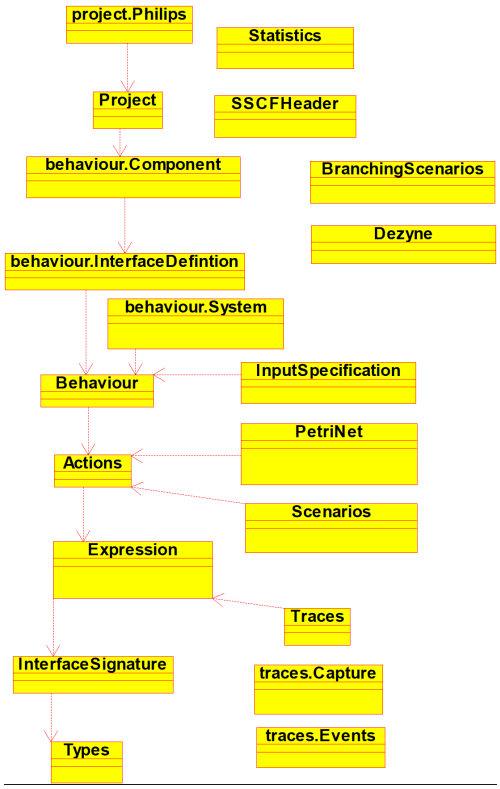
iii. Optionally, set the workspace location to an existing ComMA workspace.
iv. Press 'apply'

b. In Run Debug configurations 'Eclipse Application', select the created configuration and click 'debug' in the bottom right
c. In the future, this option can be reached through Run Debug History configuration name

A disclaimer: This guide specifies to exclude nl.esi.comma.project.standard. The source instructions support including this package, but require more manual intervention because of this inclusion. Since the function of nl.esi.comma.project.standard is unclear, it was left out of this guide.

ComMA project layout

Project Diagram



All projects are prepended with 'nl.esi.comma.', but this is dropped for brevity. This is a dependency tree of the projects' grammars.

New project contents

For a new Xtext Project (for this guide called 'example'), a couple of default projects and packages are generated:

- nl.esi.comma.example
- nl.esi.comma.example.ide
- nl.esi.comma.example.tests
- nl.esi.comma.example.ui
- nl.esi.comma.example.ui.tests

These projects are entirely empty, besides two files: in 'nl.esi.comma.example/src/nl.esi.comma' there is an Example.xtext file (with default 'hello world' contents), and a GenerateExample.mwe2 file.

The .mwe2 generation file can be executed to generate boilerplate functions that can be expanded upon. The generated files are:

Project	Folder	Package	File	Function
nl.esi.comma.example	src	nl.esi.comma	ExampleRuntimeModule.xtend	Boilerplate code, ignore
			ExampleStandaloneSetup.xtend	Boilerplate code, ignore
		nl.esi.comma.generator	ExampleGenerator.xtend	Empty generation method, can be expanded upon.

		nl.esi.comma.scoping	ExampleScopeProvider.xtend	Empty Scope provider, you can add methods to define scope of grammar rules.
		nl.esi.comma.validation	ExampleValidator.xtend	Empty validator, you can add methods to define grammar rules in code.
	src-gen			Generated Files, ignore
	xtend-gen			Generated Files, ignore
	model			Generated Files, ignore
			plugin.xml	Configuration File
			.antlr-generator-3.2.0-patch.jar	Grammar jar, ignore
nl.esi.comma.example.ide	src	nl.esi.comma.ide	ExampleIdeModule.xtend	Empty Class used to register IDE compontents
			ExampleIdeSetup.xtend	Boilerplate code, ignore.
	src-gen			Generated Files, ignore
	xtend-gen			Generated Files, ignore
nl.esi.comma.tests	src	nl.esi.comma.tests	ExampleParsingTest.xtend	Test file with sample test method
	src-gen			Generated Files, ignore
	xtend-gen			Generated Files, ignore
nl.esi.comma.ui	src	nl.esi.comma.ui	ExampleUiModule.xtend	Register components for the Eclipse IDE
		nl.esi.comma.ui.contentassist	ExampleProposalProvider.xtend	Extends from AbstractExampleProposalProvider. Can override methods generated for each grammar rule, used for Eclipse content assist function (ctrl-space code suggestion). Unsure of exact function.
		nl.esi.comma.ui.labeling	ExampleDescriptionLabelProvider.xtend	Used to define hover-over labels for your grammar
			ExamplerLabelProvider.xtend	
		nl.esi.comma.ui.outline	ExampleOutlineTreeProvider.xtend	Allows you to create a custom outline view in Eclipse.
		nl.esi.comma.ui.quickfix	ExampleQuickfixProvider.xtend	Allows you to implement automatic fixes for validation warnings, that users can run when encountering these warnings.
	src-gen			Generated Files, ignore
	xtend-gen			Generated Files, ignore
nl.esi.comma.ui.tests	src			Empty, JUnit tests can be added here.
	src-gen			Generated Files, ignore
	xtend-gen			Generated Files, ignore

ComMA Project Function

This table shows the function of each project, helping developers find the relevant grammar and xtend files they need to change. It is filled in to the best of my ability, but is surely lacking and/or inaccurate in places.

Project	Defines	Grammar Function	Code Function
types	Types	.types file	
signature	Interface signature	.signature file	
expression	Variables Expressions		
actions	Actions done in a method		

behaviour	Statemachine Time, data and generic constraints Ports	Baseclass	
behaviour.system	System	unknown	
behaviour. interfaces	Interface behaviour	.interface file	
behaviour. component	Component Functional constraint	unknown	
project	Project	.prj file	Generator files: • Documentation • Dezyne • GraphML • Simulation and Scenarios • Monitoring • PetriNet • POOSL • UML
project.philips	Generatorblock	unknown, related to C++	Main.xtend, used presumably as an entry point for running a project file? Generator files: • CPP Stub files • Traces
traces	Server Client Message	.traces file	
traces.capture	Message	Alternative grammar, unused	
traces.events	Connection Component	Alternative grammar, unused	
scenarios	Scenario	unknown, possibly alternative method of defining behavior	
petrinet	Statemachine Place Transition	unknown, possibly alternative method of defining behavior	
inputspecification	Body	unknown	
dezyne	NamespaceDeclaration InterfaceDeclaration ComponentDeclaration TypeDeclaration	unknown	
branchingscenarios	Node Event	Unclear: possibly tree-like structure used to define behaviour	
sscfheader	Interface	Unknown, related to C++	
statistics	Row	.statistics file	

Steps to add/change the grammar

1. Grammar Changes: Identify what file needs to change based on the overview above, and make the necessary changes. Simply changing the Xtext files will make eclipse autocomplete support the new grammar, which can help with testing whether the grammar is properly added, without unintended side effects.
2. Generator changes: Compile a list of generator changes you want based on the changed grammar. A non-complete list of generators you might want to edit:
    - UML Diagram
    - Documentation
    - Simulation
    - Monitoring
    - Statistics
    - CPP stub files generation
3. Identify the relevant generator files based on the tables above, and edit it.

Editing grammar best practices:

- Document grammar:
    - All generated code has 'add user-doc'. All changes to this file dwill be overwritten on generation, and thus should not be edited.