A family of Domain-Specific Languages for Integrated Modular Avionics

Ricardo Alves[1], Vasco Amaral, João Cintra[2], and Bruno Tavares[2]

[1] NOVA LINCS, Universidade Nova de Lisboa, Portugal
[2] GMV

Abstract. In the domain of avionics, we can find intricate software product lines constrained by both aircraft's hardware and conformance to strict standards. Existing general-purpose languages are complicated, as they do not hide unnecessary low level-details. This situation potentially leads to a lengthy process in the specification phase and the loss of control over the quality of the specification itself and possibly resulting in the generation of inconsistent products.
In Software development for avionics systems, the pressure of time-to-market is high. Additionally, the long time taken for systems certification of this sort of critical system pushes for the development of solutions that support specifications correct by construction. With that kind of solutions, we can release the burden of the software developer by positively constraining the configuration of the products. In this paper, we put into practice an in-house solution that implements the concept of Product Lines of Domain Specific Languages (DSLs). The solution allows generating dedicated DSLs for each sub-family/configuration in Modular avionics departing from the model of a given aircraft.

Keywords: Model-Driven Development · Family of Languages · ARINC 653 · Integrated Modular Avionics DSL · Implementation Reuse

1 Introduction

In the field of software development for avionics systems, the engineering life cycle of an aeroplane can reach up to almost a decade. Therefore, the pressure of time-to-market is very high. The long time taken for systems certification of this sort of critical system pushes for the development of solutions that support specifications correct by construction. They should also offer expressiveness at the level of abstraction of the problem domain instead of the technicalities of the solution domain. Domain-Specific Languages (DLS)[10], are known precisely for that, as they help to release the burden of the software developer by positively constraining the configuration of the products with less accidental complexity, and therefore increasing its productivity.

However, in Avionics, the variety of products can be vast (with sub-categories of families of products), with a broad set of configuration space possibilities at the configuration of partitions for Integrated Modular Avionics (IMA).

A problem to develop such DSLs is that to cover all products and keep versatile enough to capture the domain rules, it will have to become more general-purpose leading to the loss of control over the quality of the specification itself

and bringing all the complexity again, and unnecessary details, found in lower levels of abstraction. With a vast design space, this situation might lead to the possible generation of inconsistent products, requiring elaborated Testing and Checking techniques. Ideally, there would be dedicated DSLs for each subfamily of products. However, the challenge of automating the process to generate different DSLs, like a common SPL, is still not a common practice, and it lacks successful examples in the bibliography.

This work presents an in-house solution for supporting product lines in the field of avionics (Sec. 3) of DSLs for the architecture of Integrated Modular Avionics (Sec. 2). We define our perspective of a family of DSLs, and we discuss its implication in the language design (4), including to configure and tailor different DSLs each of which will be destined to specify the set of possible applications for a given subfamily of products (partitions in a Real-Time Operating System). We then show our process and framework to generate the visual editors, validation rules and code generation templates from the input configuration models and the standard reference (Sec. 5). To complete the study, we evaluate (Sec. 6) the usability of the proposed framework that implies to analyse the configuration of the generated DSLs and to use them, comparing this solution to the previous approach. Finally, we discuss related work (Sec. 7) and conclude (Sec. 8).

2 Integrated Modular Avionics

IMA is an electronic architecture of an airborne system and consists of a network of physical computation modules which allow real-time processing. Each module supports the execution of multiple application with different critical levels. This means that a set of avionics functions that may be correctly executed, on the same hardware with guarantees of the external behaviour of each application, is the same as the one performed in specific dedicated hardware.

The communication between computation modules is usually achieved by using one shared high-speed network and regulated by a particular standard like ARINC 664 [2]. Most of the aircrafts flying today use a federated architecture, which is a proprietary architecture that uses specific hardware physically segregated for the execution of one avionic function. Plus, the federated architecture requires dedicated interconnection cables between each computer module.

The last generation of aircrafts have an IMA architecture which provides the following benefits: 1) Reduces aircrafts' global weight, with fewer cables and computation modules (so it can be used for more cargo or fuel); 2) Reduces energy consumption (to be used by other systems, requiring fewer batteries); 3) Reduces complexity of the physical construction of the aircraft (less hardware); 4) Simplifies the development process of avionics software (the avionic application developers are focused on the high-level software layer).

Inside of each IMA computation module, it is installed a real-time operating system (RTOS). The standard ARINC 653 [1] specifies the application programming interface (API) between avionic application and operating system. Additionally, this standard restricts the answer of RTOS to the API invocations and provides time and space constraints for correct modules partitioning.

In Fig. 1 it is depicted the module architecture. The real-time operating systems run on the hardware, providing services to the avionics software hosted by several partitions. The avionics software makes calls to the API (denominated APEX) to use ARINC653 services.

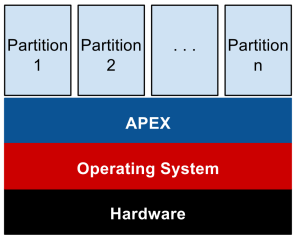


Fig. 1: ARINC 653 Module Architecture

3 Software Product Line (SPL) and DSL families

A product belonging to a family of software products share a set of common characteristics in a specific way. Building a new product is mostly a process of selection of characteristics (integration of components), rather than exhaustive programming. This way, it is a level of indirection where the decision is made regarding features of a given product. Their features characterise a product that belongs to an SPL. The variability of those is usually analysed using the Feature-Oriented Domain Analysis (FODA) [7], mainly using Feature Models.

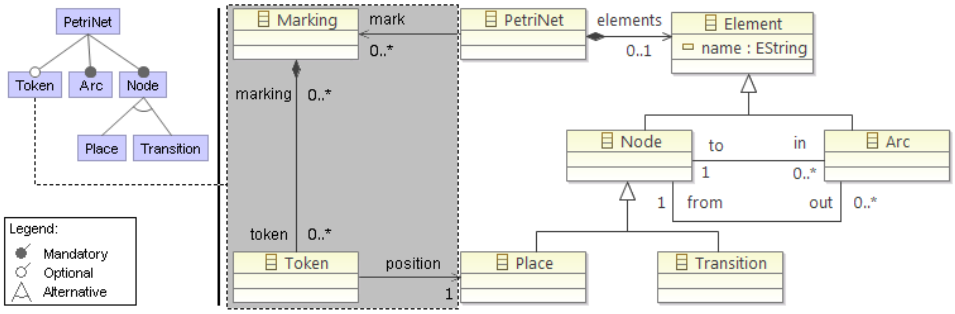


Fig. 2: Common SPL Approach applied to metamodels (Adapted from [9])

If we consider language models (metamodels) as a product, Fig. 2 shows an approach for defining a language product in the family. On the left-hand side, we present a feature model representing the variability of a particular representation of a Petri-Net. When it is required to build a new product, it is possible to choose between the existence or nonexistence of Tokens in the language. If it exists, the metamodel of the product will be composed by the part in the picture with the white and the grey background. If it does not exist, the metamodel of the product will only correspond to the part of the model with the white background.

This case is a simplified example of an SPL which has two variations and that the number of options is limited. To add new features not covered by the

feature model presented, several significant changes will be necessary, i.e. you must change both the metamodel and the feature model.

At this point we can define a family of DSLs as a set of software languages (and corresponding artefacts) to specify products for a particular domain or task, sharing a common set of key concepts, but that is specifically tailored to meet the variability of the requirements and restrict the set of possible software products into a subfamily.

4 Languages Design

Domain-specific languages can be developed using model-driven development (MDD) in an incremental way. Underlying this paradigm are models [16] and model transformations [6, 15]. One type of linguistic models are metamodels, and it defines the concrete and abstract syntax of a domain-specific language in the context. In this work, we mostly focus on the definition of the abstract syntax.

According to our DSL family definition, we will have a set of common and variable concepts represented in all generated target languages. On the one hand, some of those concepts are very rigid and are expressed in the same way in all generated languages' metamodels, on the other hand, some concepts are changed to achieve the particular configuration of each language. To better illustrate this concept of language SPL, we present Fig. 3, where we show the separation between common and specific configuration concepts and their metamodel representation. If these concepts are correctly separated, the development of metamodels can also be done separately. The idea behind this methodology is to develop the metamodel of common concepts and reuse in all languages, avoiding to do the same work repeatedly.

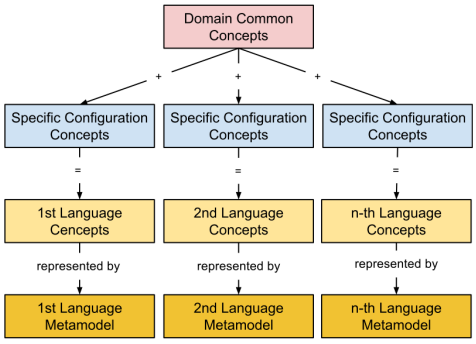


Fig. 3: Language Concepts

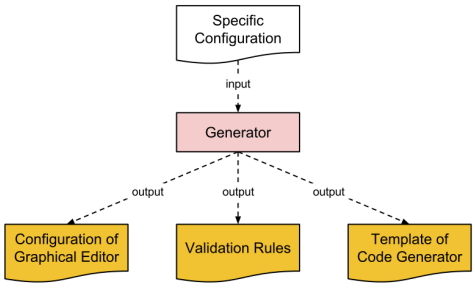


Fig. 4: Generator Module

We use model merging [8]. This consists of joining the metamodels by sharing equivalent entities among them. This implies that if the name of the entity is the main attribute for the merging process, the elements of both metamodels having the same name are made the same (merged).

Fig. 2 illustrates the resulting merged metamodel departing from the fragment metamodel represented in grey on the left together with the fragment in the right part (linked by "PetriNet" and "Place").

The whole process of generating the family of DSLs was designed to be supported by a set of tools, called Languages Modelling Workbenches. These frameworks provide some essential features that facilitate the development and automatic generation of some artefacts and tools like the graphical language editor, and code generators.

After finishing the language's abstract syntax, and its concrete syntax mapping, it is necessary to generate the configuration of the graphical editor, the instance model validation rules and the code generator (shown in Fig. 4). If the DSL product line produces a possibly large set of DSLs, these tasks need to be done automatically, without any human intervention to be manageable.

For automatic generation of languages, we developed a generator capable of creating all the necessary artefacts.

Generating Configuration of a Graphical Editor - Some graphical editors are configurable by choosing user interaction features like: where we put toolbars, select buttons locations, rename our tools and other settings.

As our DSL family has languages with distinct concepts, it is essential to generate for each language the particular configuration of the graphical editor. This can be done by using a simple model-to-text transformation, responsible for generating the configuration in the target language automatically. The produced artefact will be read and interpreted by the external tool responsible for generating the graphical editor.

Generating Validation Rules - The validation rules of a domain-specific language provide additional guarantees of the well-formedness of the new DSL instance models. If two languages are different, then the set of validation rules will probably be different.

As we have different languages in the DSL family, we need to select and generate the specific validation rules tailored to this language. This is achieved by using a simple model-to-text transformation, which means to create the validation rules or assertions automatically, and write them to a file on a proper constraint language (like OCL, EVL, etc.) to be interpreted by the editor.

Generating the Code Generator - Sometimes, although not strictly following the principle of Model-Driven Development (MDD), we find that in practice the software development process based on models, the target artefacts of a given DSL is template code in a general purpose language (GPL), that is going to be completed by the software engineer to represent the final software product. To be able to do that, we need to generate the code template, and at this point, we are talking about code meta-generation.

This transformation is similar to the model-to-text transformation, but we are not generating the final artefact yet. The result is a new template able to be interpreted by the framework tool and then generate the proper end code.

5 DSL Family for IMA

Over the years the company GMV [3] has worked successfully, as part of its core business, in IMA projects and today is continuing to develop an Integrated

[3] http://www.gmv.com

Modular Avionics Development Environment (IMADE) [17]. The main objective is to provide a toolchain for the whole IMA development process while complying with the certification of the end product.

The development of avionics applications is an essential task and, considering this, a family of DSLs is being developed for IMADE. Each generated DSL allows specifying the code skeleton/template of new avionics applications according to the partition specification.

Using a DSL, we can get the benefits of this approach like the increase of the productivity, mainly if is used a graphical environment and improving the quality of the final product by generating some parts of code automatically.

It was observed that a solution based on a single DSL for all types of partition is not enough. The language would be too general, and it would be potentially error-prone, leading to the generation of inconsistent products. The envisaged solution was then the development of a new DSL for each partition configuration and then the creation of a family of DSLs. The main advantage of such an approach is the possibility of creating a different DSL according to the specific settings for each partition (subfamily). This means to constrain the expressiveness offered by the Avionics solution. This contributes to avoiding errors by construction in the early phases. For example, when using an API method regarding communication ports, the DSL user can select only the existing ports that can be correctly used under the partition configuration.

5.1 Languages Metamodeling Workbench

For the implementation of our solution, we used Eclipse Epsilon framework, as it presents several advantages, namely: i) the possibility of integrating various plug-ins that provide extension capabilities, ii) being multi-platform, iv) open source, vi) the availability of extensive online documentation, and v) a large community of users. Another important aspect is the fact that Eclipse is a platform very often used in the field of aviation, mainly due to its ability to integrate with various tools and ease to evolve.

5.2 DSLs Generator Module

This domain has common concepts like the application programming interface specified in ARINC 653 (see the fragment of the metamodel in Fig. 6 a), but also has variable concepts like the specific configuration of one partition under development. This motivates the use of DSL families, and a DSL generator has been developed, as illustrated in Fig. 5.

The process of generating a new DSL of the family starts with an XML file containing an ARINC 653 configuration for a particular module (already produced by another tool outside the scope of this document). This file has all the additional information needed to generate the metamodel extension like the: number, type and other properties of the communication ports. Using the information in the XML file is possible to create the metamodel extension. Later this extension and the metamodel of common concepts are merged. This last metamodel has been developed at the same time as the tool and should never change in its lifetime.

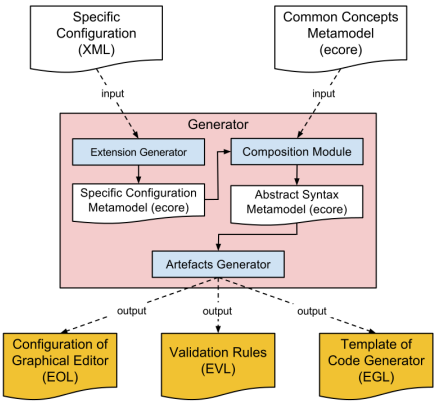


Fig. 5: IMADE Generator Module

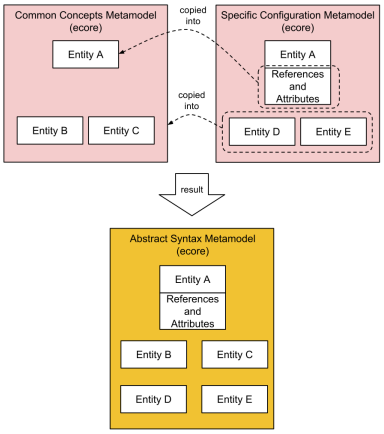


Fig. 6: Merging Metamodels

With these two operations, two Ecore intermediate artefacts are also created: the extension metamodel; and, the abstract syntax metamodel. The first one is useful if we want to verify the particular configuration of one partition using the ecore formalism. The second one is helpful to generate the DSL and its corresponding graphics editor.

The generation of a graphical editor configuration is the next stage. This operation creates an EOL file (Epsilon Object Language is used), which is interpreted by the EuGENia tool and produces the graphical environment.

After that, also based on abstract syntax metamodel, it is created an EVL file. This file contains validation rules used by the graphical environment to verify the well-formedness of the instance models realised with the new language. These validations rules are written using the Epsilon Validation Language.

Finally, it is generated by the EGL file. This last artefact contains the template to create the final code in a GPL, and it is written using Epsilon Generation Language. After the generation of all artefacts, the EuGENia tool is invoked. Some of the artefacts, like the metamodel and the graphical editor configuration file, are only used at this stage to help to generate the graphical environment.

The validation rules (defined in an EVL file) and the code template (defined in an EGL file) are used at runtime by the graphical environment generated in the first stage. The first is used when it is necessary to validate the constructs produced by the programmer by using the language previously created. The second is used when the developer intends to generate code from the constructs.

Generating Metamodel- As previously mentioned, the metamodel of the common concepts is already done. So, the first step is to create the extension metamodel with the information that is coming from an XML file (containing the settings of all existing partitions on the avionics module).

To handle the XML file, it was developed a plug-in for Eclipse. This plug-in allows the DSL user to select the file location and select the partition he wants.

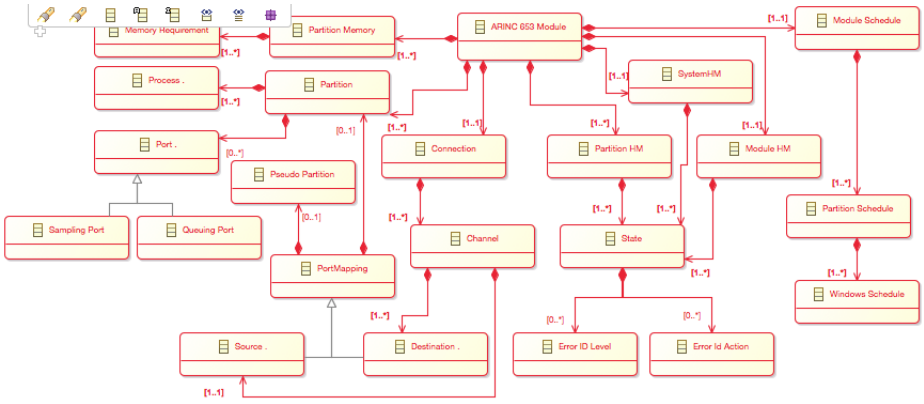


Fig. 7: Simplified fragment of the language metamodel (Ecore) visualised in Eclipse EcoreTools.

Then the plug-in processes this information and populates the internal structures with the newly obtained information.

Later, when the information is required, the appropriate methods are evoked, and they should return the data structures with the information inside these. Thus, it becomes more modular, but also simpler, to realise and develop code. This approach improves the performance of the solution because we used JAVA to develop this plug-in instead of the interpreted languages of Epsilon.

The second step is to create the extension by using the same information obtained before. Using EOL, we create the EClass, EAttribute, EReference and other elements of the extension metamodel. These elements belong to the Ecore formalism used by Eclipse Epsilon to define metamodels. In Fig. 8 it is shown

```
var package = ExtensionMM.resource.contents.first();
var mainEClass = new ExtensionMM!EClass;
mainEClass.name = "Process_Management";
mainEClass.abstract = true;
package.eClassifiers.add(mainEClass);
for(instructionName in getPortInstructions()){
  var portInstruction = new ExtensionMM!EClass;
    portInstruction.name = instructionName;
    portInstruction.eSuperTypes.add(mainEClass);
  var emfAnnotation = new ExtensionMM!EAnnotation;
  emfAnnotation.source = "gmf.node";
  var detail = new ExtensionMM!EStringToStringMapEntry;
  detail.key = "border.color";
  detail.value = "0,0,0";
  emfAnnotation.details.add(detail);
  portInstruction.eAnnotations.add(emfAnnotation);
  package.eClassifiers.add(portInstruction);
}
```

Fig. 8: Extension Generation Code

part of the code responsible for generating the extension. The *ExtensionMM* element is the extension metamodel (this element starts empty). Between line 3 to 9 is created the anchor entity and is also exists on the metamodel of common concepts. This entity will be used for merging both metamodels.

The loop condition has the *getSamplingPortInstructions()* method that returns the instructions taking into account the specific configuration of the partition. Between line 13 and 33, the entities responsible for representing the instructions in the metamodel are created.

While creating entities in the extension metamodel we also added the GMF annotations. These are necessary to generate the graphical environment with Eugenia. Code of lines 19 and 31 in Fig. 8 is responsible for that.

The third step is to merge the metamodel of common concepts with metamodel of specific configuration concept (extension metamodel created in the second step). This operation is illustrated in Fig. 6. It first identifies the entities that have common names in both metamodels. All attributes and references in each identified entity are copied from extension metamodel to the respective entity in metamodel of common concepts. All other entities are copied from the extension metamodel to the common concepts metamodel. Due to the internal representation of the ecore formalism, these two operations ensure the merger of the two metamodels.

Generating Configuration of Graphical Editor - The configuration of the Graphical Editor is defined using the EOL language. This configuration is stored in a file with the name of *"ECore2GMF.eol"*.

In general terms, it was defined as a template that contemplates the possible variations in the abstract syntax metamodel. This template generates only the configuration of the elements that are present in the metamodel and uses its attributes to produce the correct validation rules.

To implement the solution, we could have used instead of the EGL language included in Epsilon. This is a language tailored for model-to-text transformation (M2T) and can be used to transform models into textual artefacts like the code in multiple programming languages.

Our concern from the beginning was to use the same programming language to make the implementation more straightforward and to concentrate the code of the generator module without ever compromising its modularity. Therefore, we also used EOL language.

```
if(getPortInstructions().size() > 0){
  write("var toolGrp = new GmfTool!ToolGroup;");
  write("toolGrp.title = 'ARINC653: Interpartition';");
  write("toolGrp.collapsible = true;");
  write("palette.tools.add(toolGrp);");
}
```

Fig. 9: Generation of the Configuration

```
var toolGrp = new GmfTool!ToolGroup;
toolGrp.title = 'ARINC653: Interpartition';
toolGrp.collapsible = true;
palette.tools.add(toolGrp);
```

Fig. 10: Result of the Configuration

EOL is not appropriate to deal with model-to-text transformations because it cannot deal directly with text files. To tackle this limitation, we developed a plug-in that allows writing a file appropriately, i.e. to deal with opening and closing files and write the desired text in the correct position. In Fig. 9 it is shown some code which generates the configuration. The result of running this code is illustrated in Fig. 10. The write method allows writing directly in the

text file. The code presented in Fig. 9 only writes the text in the file if there are instructions ARINC653 to handle communications.

In Fig. 10 is shown how to change the graphical environment, by creating a toolbar group with the name of *"ARINC653: Interpretation"*. Later on, the buttons can be added to the toolbar group.

Generating Validation Rules - A similar technique to the one described before was used in the same way to produce the validation rules. In this case, the target language is EVL.

The validation rule illustrated in Fig. 11 checks whether the *GET_QUEUING_PORT_ID* instruction has a defined port, otherwise a warning is shown. This rule and similar rules are produced if the configuration of the partition has at least one queuing port.

```
context GET_QUEUING_PORT_ID {
  constraint HavePort {
    check : self.port.isDefined()
    message : 'No port found'
  }
}
```

Fig. 11: Resulting Validation Rule

Generating the Code Generator- The code generator artefact is generated similarly to the previous artefacts. In this case, the final result is an EGL code template to be run in the Eclipse Epsilon framework.

Due to the use of EOL language to perform the model-to-text transformations (described in 5.2) it is now possible to produce EGL code more efficiently. Unfortunately, as a slight technology limitation and drawback, EGL does not cope well with embedded EGL code (for the purpose of being part of the generated rules) in the EGL template. In fact, it tries to interpret all the specified EGL code.

```
write("[%for(queuingPort in getQueuingPorts()) {");
write("var name = queuingPort.get('portname')%]");
write("CREATE_QUEUING_PORT('[%=name%]', ... );");
write("[%}%]");
```

Fig. 12: Generation of Code Generator

A short example of the code is illustrated in Fig. 12. The EGL code is written for a file. Later on, when the template is executed, the EGL code between characters *[%* and *%]* is executed. The final GPL code produced in Fig. 12 is *CREATE_QUEUING_PORT('<PORT NAME>', ...);* for each port present in the configuration file of partition.

6 Evaluating the usability

A quasi-experiment was held to compare the newly generated DSLs and the previous programming experience with C was carried away to determine the degree of success of our approach in terms of usability. The steps were:

1) Subject Recruitment - We looked for subjects proficient with programming in C and showing domain knowledge, including experience with previous projects where with ARINC 653. The universe of subjects was divided into a set of 4 medium experience and a set with 2 very experienced users. Finally, a seventh subject did not have any experience with ARINC 653.

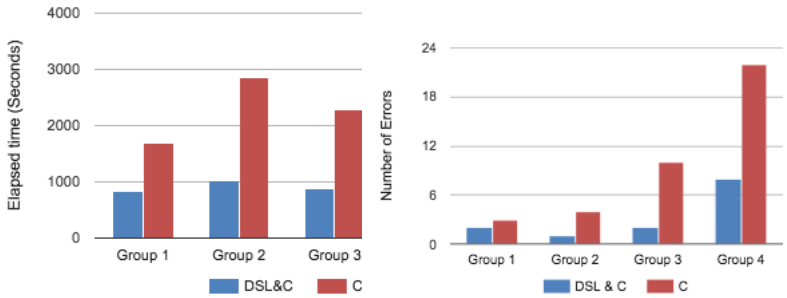
2) Task Preparation - We prepared several materials: slides for the training session; and, exercises for the final evaluation session. Two Desktops with the same hardware were installed with Microsoft Windows 7 Professional Software, with JAVA v1.7.0, Notepad++, CamStudio and the Eclipse framework with our DSL. The subjects were grouped into four groups: group 1 and group 2, with high experience (2 subjects each); group 3, medium experienced (2 subjects); and, group 4, with the non-ARINC 653 knowledge subject. Group 1 was evaluated while solving the exercise first using the DSL and C to introduce lines of code in the generated code, and after that, they produced their solution for the same problem just in C. The second group did the opposite, to minimise the bias. The third group and the 4th group (the inexperienced subject in ARINC) started with solving the problem with the DSL and C and then just C. All training and exercise sessions were monitored by the same person (project developer).

3) Pilot Session - to avoid last minute problems, wasting time from the limited set of volunteers, we duly tested the material prepared. A champion for this technology in the company was asked to take the role of the subject and tested both the training material and the evaluation exercises. As expected, several errors and ambiguities were detected, as well as better strategies were designed to control timing.

4) Training Session - This session was the first contact of the subjects with the DSL, and the subjects had the opportunity to try out some exercises and explore the language features. This session was video recorded for later analysis of the time taken to finish the tasks.

5) Evaluation Session - Each subject had to participate in two sessions, the DSL with C and the session with just C (order in which this sequence occurred could be the reverse). The exercises were the same for both sessions. In the session with the DSL the C code skeleton that was generated in the model to code generation was used by the subject to introduce more lines of C code to complete the exercise. We have prepared three different exercises during the assessment. The first was meant for training, and the remaining two were used for the evaluation session. The degree of difficulty increased with each exercise in the sequence. At the end of each exercise, the subject would have a sophisticated avionics program, with communication between partitions and communication in between processes of the same partition.

As typical with in-house DSLs, the reduced number of subjects is a negative aspect of statistical evidence of the results. However, the mentioned seven subjects, with exception to occasional new recruited programmers, represent a majority of the whole universe of programmers that will make use of this solution in the company. The result of this assessment was positive. This information was also assessed via questionnaires at the end of the evaluation sessions. All users appreciated the tool since it improves their productivity by speeding up the process of producing the required repetitive code that is usually programmed manually. The subjects consider that the code resulting from the automatic model-to-code transformation is already well structured. It is well accepted that few lines of code have to be introduced to finalise the intended functionality.



a) Consumed time b) Number of errors

Fig. 14: Consumed time and errors with the exercises.

As observed in Fig. 13a, the estimated achievement, based on a mean value, is a gain of about 59,91% of the production of the predetermined avionic code.

Using just the C language, group 2 (experienced subjects), have taken more time than group 3 (moderately skilled). This fact can be explained (when looking at the video recorded sessions) because the users in group 2 took the time to comment correctly and indent code to be understandable and re-usable.

7 Related Work

Software Product Lines [13] and Domain-Specific Languages [10] are familiar concepts in both scientific and industrial communities. But, examples of the combination of the two concepts resulting in the particularisation of the family of DSLs are still very scarce. We next discuss MDD that inspired our solution.

Composition of DSLs- The first approach is to perform the composition of various DSLs, arising or not from different modelling aspects. One goal of this method is to create new languages to reduce the effort in the development of DSLs initially used in the composition. The composition of the new abstract syntax metamodel of the language is performed manually by engineers like a puzzle, reusing and adapting parts of metamodels. This process may be assisted by a feature model describing the concepts that are covered by each DSL, dependencies or conflicts between DSLs, and finally how the refinement of languages affects the coverage of the concepts [5]. This approach has several drawbacks. By definition a DSL is tightly coupled to the domain for which it was developed, which implies a reasonable effort from Language Engineers to compose languages from different domains [5]. This effort is due to the need to adapt to the reality of the concepts of the new language, and may ultimately be more appropriate to draw it from scratch. Another significant disadvantage of this process is related to the need for human intervention when it is necessary to generate a new DSL, which makes the time-consuming and error-prone process.

The repeated composition of the same DSL with other languages for specific domains is a practice used when we want to add an essential feature of a host language to a given hosted language [3, 12].

Positive Variability- An approach that uses the metamodel's positive variability of the language can be used to create a family of languages. It starts with a common minimum metamodel to all languages of the same family. Next, it is performed the composition with required extensions to achieve the desired language [11]. The extension process can be accomplished by using the model merging approach [8] or by weaving the models [14]. In both cases, extensions are anchored at pre-specified points through composition transformations.

It is possible to extend a language with the use of positive variability indefinitely. However, it is necessary that the composition of the two metamodels make sense (share the same base semantics) and belong to the same domain [8].
Negative Variability - In this approach, the complete metamodel of the language is initially defined completely. Afterwards, parts of the metamodel are removed, to obtain the desired configuration for the new member of the family of languages [11]. When the user wants to create a new DSL, selects the functionalities that are needed in the feature model. Removal and modification of metamodel elements are performed in an automated way using the predefined transformations. The last step is the language generation and its graphical environment [12].

The negative variability is suitable for use in situations where initially we have the full knowledge of the modelling domain [4]. It allows for the creation of a limited number of DSLs of the same family, according to the total number of combinations that we can select in the feature model.

Analysis of Alternative development approaches- We can find characteristics of automatic generation of a language and high extensibility in the technique that uses **positive variability** [14]. The merge of models became possible thanks to the contribution that explore the semantics of languages composition [8]. On the other hand, **negative variability** also has useful features, but initially requires the full metamodel of the language, which, in many areas, is difficult to obtain because it is too long or is always evolving. This aspect makes the extensibility limited because many additional changes are needed to extend the automatic generator of languages. This situation means that it would be required very frequently the manual intervention of a software engineer.

In conclusion, the positive variability presents the most appropriate approach for developing a DSL Family for IMA.

8 Conclusions

We report an in-house solution for the avionics software industry, that assists the IMA developers. It allows to produce more efficiently and error-prone code thanks to the automatic construction of languages that are part of a DSL family as an intermediate step, to help to constrain the software engineer's design space for the code generation. For that purpose, we used the Eclipse Epsilon and the positive variability technique. Thanks to our generator, it is now possible automatically produce all the required components/artefacts that make part of the new language IDE. For each DSL, we generate automatically, based on a configuration specification model of the standard ARINC 653 (prone to evolution): i) the new language metamodel; ii) the visual editor with the adequate

concrete syntax; iii) the corresponding well-formedness rules (in EVL); and iv), code generator.

As future work, it will be necessary to work on the automatic generation/-configuration of the dedicated set of properties to be checked in the new DSL instance models (supported by model checking tools).

Acknowledgments

NOVA LINCS (Ref. UID/CEC/04516/2019). FCT/MCTES: DSML4MAS (TUBITAK/0008/2014); 2018/2019(Proc. DAAD 441.00) "Social-Cyber-physical Systems modelling".

References

1. ARINC 653 Avionics Application Software Standard Interface, Part 1, Required Services. Annapolis, Maryland, USA (2003)
2. ARINC 664 Aircraft Data Network, Part 1, Systems Concepts and Overview. Annapolis, Maryland, USA (2006)
3. B. Barroca, L. Lúcio, D. Buchs, V. Amaral, L. Pedro: DSL Composition for model-based test generation. In: 3rd Int. Workshop on Multi-Paradigm Modelling: Concepts and Tools. No. 21 in Electronic Communications of the EASST (2009)
4. C. Huang, Y. Kamei, K. Yamashita, N. Ubayashi: Using Alloy to Support Feature-based DSL Construction for Mining Software Repositories. In: Proc. 17th International Software Product Line Conference Co-located Workshops. ACM (2013)
5. J. White, J. Hill, S. Tambe, A. Gokhale, D. Schmidt, J. Gray: Improving Domain-specific Language Reuse through Software Product-line Configuration Techniques. In: IEEE Software, Volume: 26, Issue: 4 (2009)
6. K. Czarnecki, S. Helsen: Feature-based Survey of Model Transformation Approaches. IBM Syst. J. **45**(3) (2006)
7. K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-021, SEI (1990)
8. L. Pedro: A Systematic Language Engineering Approach for Prototyping Domain Specific Modelling Languages. Ph.D. thesis, Université de Genève (2009)
9. M. Barbero, F. Jouault, J. Gray, J. Bézivin: A Practical Approach to Model Extension. 3rd ECMDA-FA'07, Springer-Verlag, Berlin, Heidelberg (2007)
10. M. Völter, et al.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
11. M. Völter, I. Groher: Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development. In: Proc. 11th International Software Product Line Conference. SPLC '07, IEEE Computer Society, USA (2007)
12. M. Völter, I. Groher: A Family of Languages for Architecture Description. In: Proc. 8th Workshop on Domain-Specific Modeling (2008)
13. P. Clements, L. Northrop: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA, USA (2002)
14. P. Sanchez, et al.: VML*–A Family of Languages for Variability Management in Software Product Lines. In: Proc. SLE. ACM Press (2009)
15. S. Sendall, W. Kozaczynski: Model Transformation: The Heart and Soul of Model-Driven Software Development (2003)
16. T. Kuhne: What is a Model? In: Jean Bezivin, Reiko Heckel (ed.) Language Engineering for Model-Driven Software Development. No. 04101 in Dagstuhl (2005)
17. T. Schoofs, et al.: An Integrated Modular Avionics Development Environment. In: Proc. 28th Digital Avionics Systems Conference - DASC '09. IEEE (2009)