Domain-Specific Languages for the Definition of Automotive System Requirements

Florian Bock, Reinhard German, Sebastian Siegl

Domain-Specific Languages for the Definition of Automotive System Requirements

Florian Bock and Reinhard German
Department of Computer Science 7
Friedrich-Alexander-University of Erlangen-Nuremberg
D-91058 Erlangen, Germany
Email: {florian.inifau.bock,reinhard.german}@fau.de

Sebastian Siegl
Audi AG
D-85045 Ingolstadt, Germany
Email: sebastian.siegl@audi.de

***Abstract*—Over the last decades, the complexity of modern cars and the included hardware and software has vastly increased. To maintain a constant development and release cycle for car manufacturers, more sophisticated development methods are required. The fields of requirements engineering and test management have high optimization potential. In order to improve the transitions between the different development stages, the combination of a controlled natural language and a domain-specific language can be used, which is drafted in this paper. It enables the user to specify requirements textually with automotive vocabulary and generate artifacts such as functional models.**

***Keywords*—Software Engineering, Automotive, Artifact Generation, DSL.**

I. Introduction

Modern cars include a wide variety of complex software functions, ranging from essential types such as engine controlling to user-assisting types such as driver assistance functions. The amount of software code required for this functions is constantly increasing. That is why software engineering methods used over the last decades are no longer sufficient to be able to maintain the established development cycle and at the same time produce the necessary quantity of code. New methods such as agile development techniques are introduced and thus, a larger variety of tools are employed. These tools differ in their characteristics as well as in their field of application. Generally speaking, the development process can be separated into several phases mainly based on the V-model [1]:

- Specification: The system is specified in an abstract way, with focus on the use cases, actors and system boundaries.
- Design: The system is enriched with more details, the system architecture is defined and technical characteristics such as required hardware/software and interfaces to other systems are included.
- Implementation: The system design is transformed into code, either via code generation or manually.

Each of these phases is linked to an individual testing phase. To accumulate all relevant information of a single development phase, artifacts are used, e.g. models, diagrams or documents. The degree of detail of these artifacts increases during the development process. Because the phases are based one on the other, the artifacts should be reused in subsequent phases to avoid duplication of effort caused by regenerating them.

Especially in large projects, a single engineer cannot handle all phases at once, so separate engineers treat each development phase. Due to different reasons such as the lack of communication, local separation of the engineers or complex hierarchical structures, the desired reuse of the artifacts either does not occur at all or is carried out manually in a superficial way.

A possible solution for this would be to use mandatory guidelines for all engineers. Such guidelines are quite hard to define due to the great variety of involved modeling techniques and tools. In addition, they have to be complied with manually. A key point in this discussion is the designation of the master in the development process. Different approaches are possible, from implementation-driven (requirements according to the implementation, often used in predevelopment phases) to specification-driven (implementation according to the requirements). Many of the projects in the automotive domain start implementation-driven whereas the desired trend is focused more towards specification-driven projects, where the specification is created first and is used to derive basic models and diagrams for any subsequent development phase. This can be performed manually, though an at least partly automated derivation is more efficient and less prone to errors.

Such a derivation can be achieved by the combination of a *Controlled Natural Language* (CNL) and a *Domain-Specific Language* (DSL) for system specification, which are drafted in this paper. They are particularly designed for the automotive domain and include an automatic generation of various required artifacts (e.g. models) for the development process.

II. Concept

A CNL is based on a natural language such as English, but is limited in terms of grammar and vocabulary. There is no general definition of a DSL in the literature, but it can be summarized as a programming language designed for a specific domain with the capability of generating predefined target artifacts.

Contrary to CNLs, which typically use a textual form of presentation due to their relation with common natural languages, a DSL can either be textual, graphical or use a combination, which is called hybrid. This choice is based on the target stakeholders and the requirements of the project

environment. In the automotive domain, many system specification documents are currently created and modified in *IBM Rational DOORS* in form of plain textual requirements without any embedded functionality. If diagrams or models have to be included due to the project settings, they are attached as manually created pictures or simple models (e.g. as *Microsoft PowerPoint* documents). Therefore, a hybrid approach is most appropriate for our case, because textual requirements match the existing documents and additional graphical information in form of models can be directly embedded.

The adoption of such a newly introduced methodology to specify systems mainly depends on the acceptance among the established system architects. To minimize the familiarization difficulties, the presentation and the terminology have to be leaned on already available documents and editors. That is to say the syntax and semantics of the specification should be as similar as possible to existing requirements in *DOORS*. For this, a detailed analysis of available requirements have to be carried out, so that common phrases, established terms and the intended underlying information can be collected and abstracted. The level of detail of the requirements differ depending on the current development phase, the personal capabilities and project experience of the specifier and the target audience. The inclusion of technical information such as interface specifications, detailed algorithms or code are not unusual. Such information are typically extracted manually afterwards and used in the system design and implementation.

The following categories of information embedded in the requirements have already been identified:

- Functional requirements, which can be used to create behavior models (e.g. state machines, activity models or timing models).
- Constraints, which specify technical or legal limitations which have to be taken into account in the system architecture.
- Descriptive requirements, which provide system details used for documentation.
- Test or scenario descriptions, which can be used to formulate test cases or formal test scenarios.

A future extension of this list is likely and depends on the investigated existing requirements and the individual project settings.

Besides the structure and representation of the DSL, the desired output artifacts have to be specified and the proper generators for each artifact have to be implemented. In the automotive domain, a vast variety of development tools and methods are used, ranging from specification instruments (e.g. *DOORS*) to implementation methods (e.g. *C++*), so an individual selection of desired target artifacts for a specific project scenario has to be defined.

The concept of our CNL/DSL-combination for the given scenario consists of a worflow divided into four steps:

1) As initial situation, the unstructured information about the system to develop are available in form of documents, slides or plain ideas.

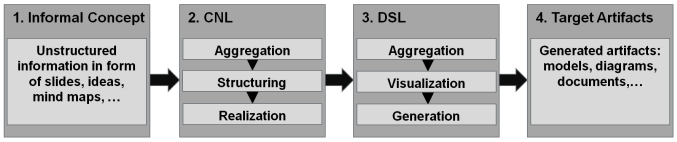

Fig. 1: Concept workflow

2) These information are aggregated, structured, and a predefined lexicon and grammar (based on the English language and automotive related terminology) are used for realization in form of a semi-formal requirements collection. This can be done either by manually transforming the available data or by using parsing techniques [2].
3) The data incorporated in the collection are then transferred into a DSL, where the required information for the generation process are aggregated. A visualization of some additional details in the form of state charts or tracing information are provided. Afterwards, the generators of the DSL generates the specified target artifacts.
4) The generated artifacts can be manually enriched with additional details and then be distributed or used for the system implementation.

III. Related Work

In case of CNLs, Marko et. al. [3] proposed an implementation of a CNL for writing requirements in *Eclipse XText*[4]. The general requirements structure is analyzed and the topic of a domain specific meta-model is discussed. To integrate the solution with other tools, the use of the *Open Services for Lifecycle Collaboration* (OSLC) standard is suggested. This paper focuses mainly on the collection of specification information, generator aspects are not discussed.

Beside the CNLs, there are already several tools available. For example *KPIT medini analyze*[5] is a functional safety tool particularly focused on considering the *ISO26262* standard. It enables the user to specify requirements, design the system and perform various safety analysis. Nevertheless, neither does it use requirements in natural language form, nor are generator aspects included.

Due to the fact, that each DSL is by definition limited to a specific domain, they cannot be generally used in each project scenario. Recently, DSLs are becoming more popular in the automotive domain in the course of ongoing development process improvements. Some attempts already have been published, of which three major examples shall be summarized and reviewed at this point:

Voelter et. al. proposed a DSL focused on requirements embedded in the framework *mbeddr* [6]. The specification is phrased in a pseudo code notation, which covers additional features such as tracing, state machines and tables. The goal is to generate code from the specification, which is supported by the possibility of embedding code directly in the requirements. They conclude, that such a DSL offers great advantages at the expense of reasonable effort.

Ballhause et. al. [7] proposed a DSL focused on requirements engineering, which uses a formal pseudo code like

notation to describe use cases and requirements and transform them to predefined representations, e.g. tabular or graphical diagrams. They describe their experiences and problems with the definition and use of the DSL. They come to the conclusion, that such a DSL is a benefit for the development process, if the costs of its definition are acceptable and the DSL is limited to a specific company. A general approach is possible, but not expedient in view of effort and complexity.

Trindade et. al. [8] introduced a DSL focused on safety requirements. They also use a pseudo code notation and include a generator, which transforms the DSL into code. The implementation and design is closely related to *AUTOSAR* and its components. They proof DSLs as proper mechanism to automate repetitive generation tasks and to include domain specific aspects in a formal but easily comprehensible document.

Although these examples offer several DSLs, they lack the extraction and creation techniques of detailed models out of the requirements and they partly use a pseudo code notation, which does not reflect the optimized textual presentation form similar to *DOORS*, that we intend to achieve.

IV. Prototype

Several frameworks for the stand-alone definition and implementation of CNLs are already available [9]. This is mainly relevant when intending to use parsing technologies to transform already available textual documents into the CNL. We confine the first prototype to the direct integration of the grammar and the lexicon into a drafted DSL, instead of implementing the CNL separately. This is sufficient to demonstrate the intended workflow.

For the definition of a DSL, several workbenches exist, e.g. *JetBrains Meta-Programming System (MPS)*[10], *Eclipse XText*[4] or *MetaCase MetaEdit++*[11]. They differ in their features and basic characteristics, but after a comparison, *MPS* was chosen as suitable solution for the given project. One main reason for this choice is *MPS* being a so-called projectional editor, the opposite of common source editors. In a source editor, the code is edited as text, if it should be compiled, an interpreter converts the text to an abstract syntax tree and transforms the tree to the target code. Within a projectional editor, this intermediate step of interpretation is left out, instead the user directly edits the abstract syntax tree in the editor. Editing with such an editor is more limited than with plain text, because only valid commands can be used, which makes it more sophisticated for the user to use it. However, it allows the combination of an arbitrary number of DSLs without worrying about any combination problems, because the grammars will automatically match. In advance, there are already many predefined DSLs available, which cover different aspects, e.g. mathematical equations, and can be integrated in the new DSL with few effort. As side remark, both the DSLs and documents which use them are saved in an *ASCII-XML* format, which makes them easily versionable in version control systems such as *GIT*[12].

The structure and presentation of the drafted DSL is illustrated in Figure 2. The configuration paragraph allows the user to specify the artifacts to be generated. With help of the abbreviation definition of *ABS*, this acronym can be used

```
Specification Document: Braking Assistant
-------------------------------------------------------------
Configuration:
  Create ML/SL-Artefact?          true
  Create UML-Artefact?            true
-------------------------------------------------------------

  1. General
  General information and definitions.

    Requirement DEF1
    -------------------------------
    ABS : Antilock Braking System
    -------------------------------

  2. System Behavior
  Description of the system behavior.

    Requirement REQ1
    -------------------------------
    The driver assistance function aggregates the data
    from sensor A and sensor B and validates the data
    and calculates the braking recommendation.
    -------------------------------

  3. Constraints
  Technical and legal constraints/limitations.

    Requirement LIM1
    -------------------------------
    The value of sensor A shall not fall below 0.
    -------------------------------
-------------------------------------------------------------
```

Fig. 2: Textual DSL draft used for system specification

throughout the specification document while guaranteeing, that the full form can be included in any generated artifact. The system behavior in requirement *REQ1* describes the abstract algorithm of the whole system. In this case, it consists of three concatenated rules. In general, any number of rules can be combined or even divided in and distributed over several separate requirements. The strength of this approach is the similarity to natural language specifications, whereas the challenge for the DSL is to extract the underlying system behavior in order to transform it into a formal model. Constraints in any form such as in requirement *LIM1* can be used to specify on the one hand intended or given technical limitations, or on the other hand legal issues such as dependencies to a specific standard, e.g. *ISO26262*.

Essential for the benefit of our DSL are the integratability and usability of the generated artifacts. Generally speaking, a generator of a DSL in *Jetbrains MPS* gathers all information stored by the specifier in the DSL document, adds relations and associations and generates Java source code, which then is finally used to create the desired artifacts. An advantage of this concept is, that the transformation is not limited to simple text-to-text transformations, but additional tools or *Application Programming Interface* (API)s can be accessed. For example the generation of *Mathworks MATLAB Simulink*[13] models can either be executed by creating the model textually as file, or by accessing the corresponding API, which is less prone to errors caused by changes in the file structure.

To show the variety of possible target artifacts, our prototype supports the following types:

- *Unified Modeling Language (UML)*[14] sequence diagrams: Behavior specifications such as in requirement *REQ1* are usually illustrated and documented in this commonly known type of diagram. After preparing the artifact creation, the generator uses a web API[15] to

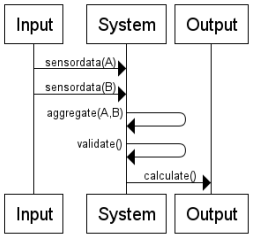

Fig. 3: *UML* sequence diagram

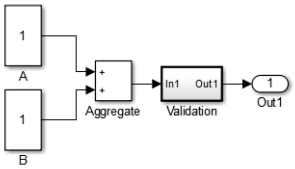

Fig. 4: Behavior model in Simulink

render the final image (cf. Figure 3).

- *Mathworks MATLAB Simulink* behavior models: Such models also cover the behavior extracted out of requirement *REQ1* (cf. Figure 4). In addition, constraint information from other requirements such as *LIM1* are included. This type of model is mainly used in the implementation of the system.
- *Electrobit Assist ADTF*[16] source code: To a certain degree, test cases are derivable from requirements and some test cases can be already defined in the specification process. For example, a specific vehicle speed constraint implies several test cases, which can be automatically extracted. Such test cases are gathered and inserted in a predefined test framework in form of *C* code. The framework itself is also generated, so additional information from the specification document can be included. As a result, a folder structure and several source code files are created (cf. Figure 5), which are compiled and can be executed afterwards. A manual extraction of the test cases and the repetitive creation of the source code for the test procedure are avoided with the automatic generation.

Apart from these artifacts, various additional models and files are possible. The question, which of these artifacts should be included in the DSL mainly depends on the intended target stakeholders. The more implementation-related the stakeholders are, the more technical information and models have to be generated.

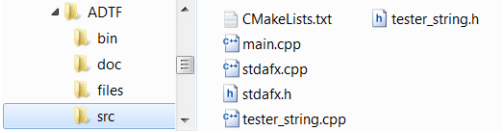

Fig. 5: ADTF folder structure

V. Conclusion and Future Work

With help of the drafted CNL/DSL-combination, the development process for modern software systems in the automotive domain can be improved. The automatic generation of target artifacts significantly reduces the required development effort and supports the process in terms of consistency and traceability. As already mentioned in [7], the key factor for success of such a DSL is the tailoring to a specific company and its vocabulary. This is mainly caused by the different in-house development procedures, wordings and strategic objectives of the different companies. Obviously, this also includes the intended and included target artifacts, because they depend on the used applications. The described DSL implementation is the first step in the creation of a complete solution for the given scenario. Ongoing research topics include the collection of company-related vocabulary and grammar, the creation of a separate CNL with parsing capabilities, the definition and implementation of reasonable generators and the trial of our approach in several actual development projects. The results will be part of future publications.

References

[1] Bröhl, A.P., *The V-Model*, ser. Software - Application Delevopment - Information Systems (in German). Munich: Oldenbourg, 1993.

[2] M. Osborne and C. K. MacNish, "Processing Natural Language Software Requirement Specifications," *Proceedings of ICRE '96*, 1996.

[3] N. Marko, A. Leitner, B. Herbst, and A. Wallner, "Combining Xtext and OSLC for Integrated Model-Based Requirements Engineering." in *EUROMICRO-SEAA*. IEEE Computer Society, 2015, pp. 143–150. [Online]. Available: http://dblp.uni-trier.de/db/conf/euromicro/euromicro2015.html#MarkoLHW15

[4] Eclipse Foundation, "XText - Language Engineering for Everyone!" [Online]. Available: https://eclipse.org/Xtext/

[5] KPIT medini Technologies AG, "medini analyze." [Online]. Available: http://www.products.kpit.com/medini-analyze

[6] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz, "mbeddr: instantiating a language workbench in the embedded software domain," *Automated Software Engineering*, vol. 20, no. 3, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10515-013-0120-4

[7] C. Ballhause and J. Leuser, "Anforderungen programmieren - eine domänenspezifische Sprache (DSL) im Praxiseinsatz (in German)," *OBJEKTspektrum Requirements Engineering 2015*, 2015.

[8] R. Trindade, L. Bulwahn, and C. Ainhauser, "Automatically Generated Safety Mechanisms from Semi-Formal Software Safety Requirements," *Lecture Notes in Computer Science*, 2014.

[9] K. Angelov and A. Ranta, "Implementing Controlled Languages in GF," *Lecture Notes in Computer Science, Volume 5972*, 2009.

[10] JetBrains, "Meta Programming System - DSL Development Environment." [Online]. Available: https://www.jetbrains.com/mps/

[11] MetaCase, "MetaEdit++ - Model your idea. Generate the rest." [Online]. Available: https://www.metacase.com/

[12] Software Freedom Conservancy, "git –local-branching-on-the-cheap." [Online]. Available: https://git-scm.com/

[13] MathWorks, "Simulink - Simulation and Model-Based Design." [Online]. Available: http://www.mathworks.com/products/simulink/

[14] Object Management Group, "Unified Modeling Language (UML) Resource Page." [Online]. Available: http://www.uml.org

[15] Hanov Solutions Inc., "WebSequenceDiagrams." [Online]. Available: https://www.websequencediagrams.com/

[16] Elektrobit, "EB Assist ADTF - Driver assistance systems start with EB Assist ADTF." [Online]. Available: https://automotive.elektrobit.com/products/eb-assist/adtf/

All links were last followed on June 06, 2016.