embeddedworld2022
Exhibition&Conference
... it's a smarter world

Supporting Joint Development of Systems and Software with Domain-Specific Languages

Juha-Pekka Tolvanen
MetaCase
Jyväskylä, Finland
jpt@metacase.com

Steven Kelly
MetaCase
Jyväskylä, Finland
stevek@metacase.com

***Abstract*—Languages for designing systems tend to be separate from languages for designing software. Changes on one side are thus harder to see and trace to the other side – and when such synchronization is done it is largely a manual effort. Domain-specific languages can make a difference, as they can enable joint development of products in fields that require both systems and software engineering expertise. Both teams use the same shared language and edit the same specifications – yet can focus on the aspects that are relevant to them. We describe this approach with examples where system and software aspects have been combined, and tracing is enabled or naturally present in the model. From these domain-specific models, many different artefacts can be automatically produced: software program code, configuration, verification, tests, deployment, material needs, instructions for installation etc. We conclude by describing the process and guidelines for defining domain-specific languages, and evaluating the effort needed.**

Keywords—model-based development; domain-specific language; system engineering; software engineering

I. INTRODUCTION

Languages for designing systems and languages for designing software have traditionally been separate. Changes in either part are not easy to see and trace to the other part. When the trace and integration between system models and software models is done it is largely a manual effort. That effort is often made even more error-prone and challenging by different versioning systems and formats for storing the specification models. Domain-specific languages can make a difference, as they can enable joint development of products that require both systems engineering and software engineering related work and design decisions. Both teams use the same shared language and edit the same design models – yet can focus on the aspects relevant to them. With appropriate tooling, setting up collaboration in the shared models is easier, and designs can be managed and versioned together.

We start by introducing domain-specific languages and showing how they differ from general-purpose languages like SysML [10] or UML [11]. We show examples from practice in which software and system development efforts are brought together through domain-specific languages. In these cases, the domain-specific languages are defined within the companies applying them. The examples cover a variety of industries and types of products, such as fish farm automation systems and heating systems. In these examples non-software and software related aspects are combined and collaboration is based on working with the same shared models.

From the domain-specific models various verification and validation efforts can be conducted, and all kinds of artefacts, like software code, material needs, tests as well as installation and deployment instructions can be generated. In the second part of this paper, we outline the process and provide guidelines for defining domain-specific languages. Finally, we conclude by describing the language creation efforts.

II. SOLUTION: A SHARED LANGUAGE

A shared language is one solution for joining systems engineering and software engineering efforts. Both teams could then collaborate, see each other's changes, and apply automation for model checking and verification. Such continuous and trouble-free integration would enable short design and testing cycles as well as early analysis and simulation. Tool support could also provide continuous integration, tracing among models and versioning the whole model together.

One approach for a shared language could be integrating existing general-purpose languages like SysML and UML: they even have a shared common foundation. This would not be practical though: First, there is no clear integration among these languages – even if using their profiling mechanism for extensions. The effort to keep work consistent and collaborative would be left to humans as these languages do not recognize integration. Consider for example the consequences and checks with a class diagram of UML when a block is added to a SysML model. The inspection of models and the necessary changes would be a manual effort left to engineers to decide – and expect that they all would do it similarly with the same results. Second, integrating existing languages would expect teams to master both languages, which are already very large in the case of SysML and UML. Third, the possibilities for generating code from these models are modest. The parts of UML that could generate code simply mirror a subset of the code, leading to duplication in that part and yet still requiring the bulk to be written by hand. For this reason software engineers today still prefer working in a programming language, with management-mandated UML just as an initial sketch or final documentation.

The situation improves when the modeling language's focus is narrowed down to a particular type of product and/or to the requirements of a single company. Examples of the former are AUTOSAR [1][9] and EAST-ADL [3], both in the automotive domain. Examples of the latter are more typical and numerous — but far less visible, as they are applied inside a single team or company. We provide examples of these company-specific solutions in Section III.

Creating domain-specific languages is easier than creating general-purpose languages: the language needs to satisfy a considerable smaller set of requirements, and typically only meet the needs of one company or product development team. For example, the language could be specific for certain medical products (pacemakers, blood separators etc.), consumer products (smart watches, camera etc.), or automotive systems. Such domain-specific languages differ considerably from general-purpose languages:

- Domain-specific languages are usually more concise and based on already known domain concepts and related rules. For example, a language supporting both automotive system engineers and safety engineers would be based on commonly known concepts originating from established practice and standards (e.g., ISO 26262 [4]). The modeling concepts would not be abstract blocks or classes but more familiar and concrete, e.g. ECU, Sensor, Safety Goal, Hazard and ASIL. Such concepts are already known, defined and applied to define automotive systems that are functionally safe.
- A domain-specific modeling language guides development as it "knows" the rules of the domain: what values are possible, mandatory, unique etc. For example, following the ISO 26262 example, a safety goal is linked to hazardous events and the events again to hazards. In languages like SysML or UML, such language concepts would not be available, nor could the correctness of their links be ensured: their interpretation and checking is left to engineers. The tight focus of the domain-specific language makes capturing the domain's information easier and more natural.
- Focusing on the domain raises the level of abstraction: ideally the language maps directly to the domain it addresses. A visible indicator of this is the notation, where the models may closely mimic the "real world" as in Fig. 1, 3, and 4 in the next section, where we describe examples from practice.
- Domain-specific models provide better possibilities for automation via generators. Even from the same set of models the program code, configuration, deployment etc. can be produced. Generation can also improve understanding, e.g. the reports using the models' functional safety information in [12].
- Modeling support can be controlled by the users, as there is no traditional vendor lock in which modeling capabilities or generators would be fixed by the tool vendor. When languages and generators can be freely changed, the users are in the driver's seat.

Research has shown that the languages which provide the best performance and satisfaction seem to be those that are narrow and defined inside the companies applying them [13][16].

III. Examples From Practice

We next describe two real-world cases where domain-specific languages joined the tasks of system engineering and software engineering. Since these languages are domain-specific and made for different needs, the way how the joint development is achieved is also somewhat different, giving a broader insight.

A. Case 1: Fish Farm Automation Systems

Developing fish farm automation systems requires designing and implementing complex functionality with various sensors (e.g., water measurement like pH value), actuators (e.g., feeders, aerators), ponds and their related functionality, as well as developing software-based functionality for storing persistent data and providing applications for controlling the automation system. Also, the infrastructure for electricity, cabling etc. must be planned and specified as well as deployed and maintained. All these various parts require joint system and software development.

A company building these systems applied a model-based solution based on a domain-specific modeling language. The language was created specifically for fish farms, and combines all aspects of the automation system. Fig. 1 illustrates this with a small model of a fish farm automation installation for a given customer. The blue ellipses are ponds for the fishes – located in the same way as in the real world. Each pond can be detailed with their characteristics along with related sensors, actuators, cabling etc. Fig. 2 shows such a specification for an aerator in the pond: it defines voltage and oxygen. Here the domain-specific language offers and guarantees that the values given are legal for the aerator.

The development of both system functionality and software is based on same shared model – even literally on the same diagrams in this case. All developers can edit different parts of the model and the language can support the preferred development methodology. It could be that the process goes as follows: a sales engineer makes the initial design together with

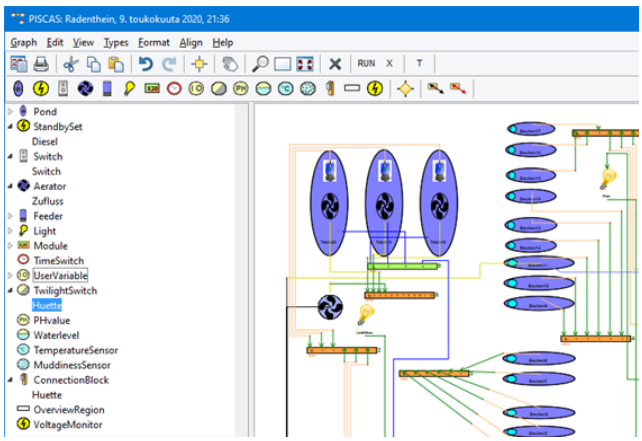

Fig. 1. Implementation of a fish farm automation system.

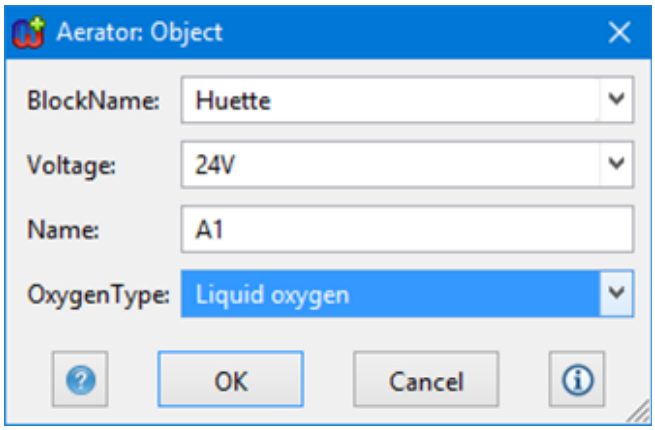

Fig. 2. Details of aerator

a customer detailing how many ponds there are, where they will be and what are their main features such as feeders or types of monitoring. Details like the oxygen type of the aerator and its related electricity and cabling needs are then specified later on by other team members.

Since the language follows the rules of the fish farm automation systems it also enables checking the specifications at modeling time. For example, if the customer requirements change and the sales engineer adds further ponds to the model, it can be analyzed for completeness and consistency.

Even more importantly, the code needed for the automation system functionality, user interface and database schemas can be generated directly from the model, along with the necessary network configurations. At the same time other artefacts like hardware mappings and simulation for testing can also be generated. This company had pushed the automation even further as they also generate the documentation and installation guides (material lists, wiring plans and even stickers for the wiring closet). As a result, there is no need to maintain the necessary artefacts separately and spend time keeping those artefacts consistent with each other. They are already consistent in the single source, the model in which development occurs.

Also, all model editing is based on the same "live" shared specification that is collaboratively specified and versioned together. As a result, any changes in a design, like adding an aerator to the pond, will lead to specifying the necessary mandatory information (as in Fig 2.), checking that the aerator has the required cabling, and ensuring controls for it (e.g. switching it on/off, monitoring etc.) will be provided in the user application.

B. Case 2: Heating Systems

Another typical alternative for achieving joint development is providing different views or sublanguages for different aspects of the final system. Figures 3 and 4 shows two different views applied for developing heating systems. Originally these views, established as interlinked languages, were created by the company to develop solar power systems, but for the sake of confidentiality we show them in the context of a heating system.

Fig. 3 shows the piping and instrumentation of the heating system: a model shows the structure of the system using familiar visualization of the instruments like sensors, valves, pipe

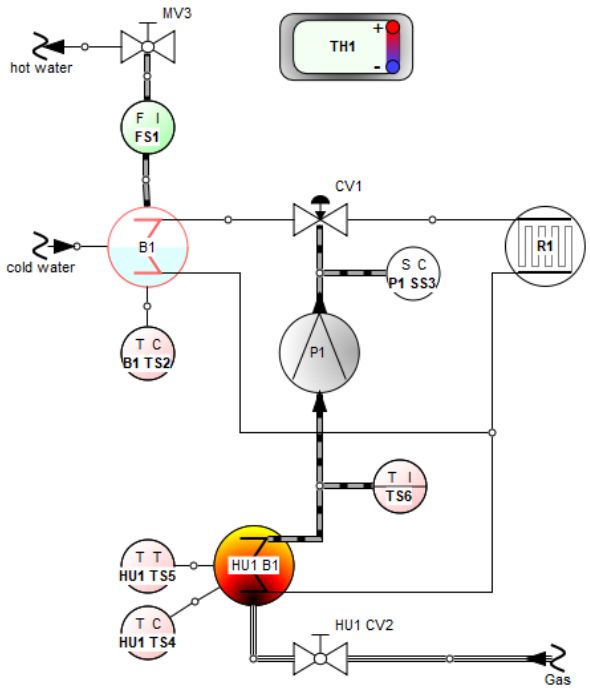

Fig. 3. Piping and instruments of a heating system

insulation etc. The notation follows the industry standards and conventions used for specific type of sensors, valves, etc.

Hardware engineers use this language to specify structural parts of the system in terms of the instruments and their piping. Based on the instruments defined, the behavior of the system is detailed in a state-machine oriented language. From these behavioral models the actual software code is produced.

In the bottom part of Fig. 3 a burner 'HU1 B1' with five pipe connections is specified. The functionality of the burner is detailed in Fig 4. via a state machine. When started, the burner is first initialized by turning it off and closing the valve that provides the gas. After initialization it moves to waiting for heat requests that are triggered either by the need to heat the water or the radiator. For heating the burner is started and the gas valve is opened.

It is important to note that these two languages still operate on the same domain concepts and can be verified and validated together. Fig 4. illustrates this integration as the burner controller is expected to open and close the gas valve 'HU1 CV2'. Yet this valve is set in the piping and instruction diagram to be controlled manually (Fig 3.). This inconsistency of the designs, between physical instruments and required control behavior, is then indicated to the engineers at the bottom of the modeling editor (see Fig. 4). The resulting check also guides engineers to change the model. Similarly, if some behavioral data, like sensor data, is expected in the state machine but such a sensor is not defined in the piping and instrumentation diagram, a warning would be shown to the engineers. This kind of integration is possible because the views operate on the same elements and two languages are integrated by their definition.

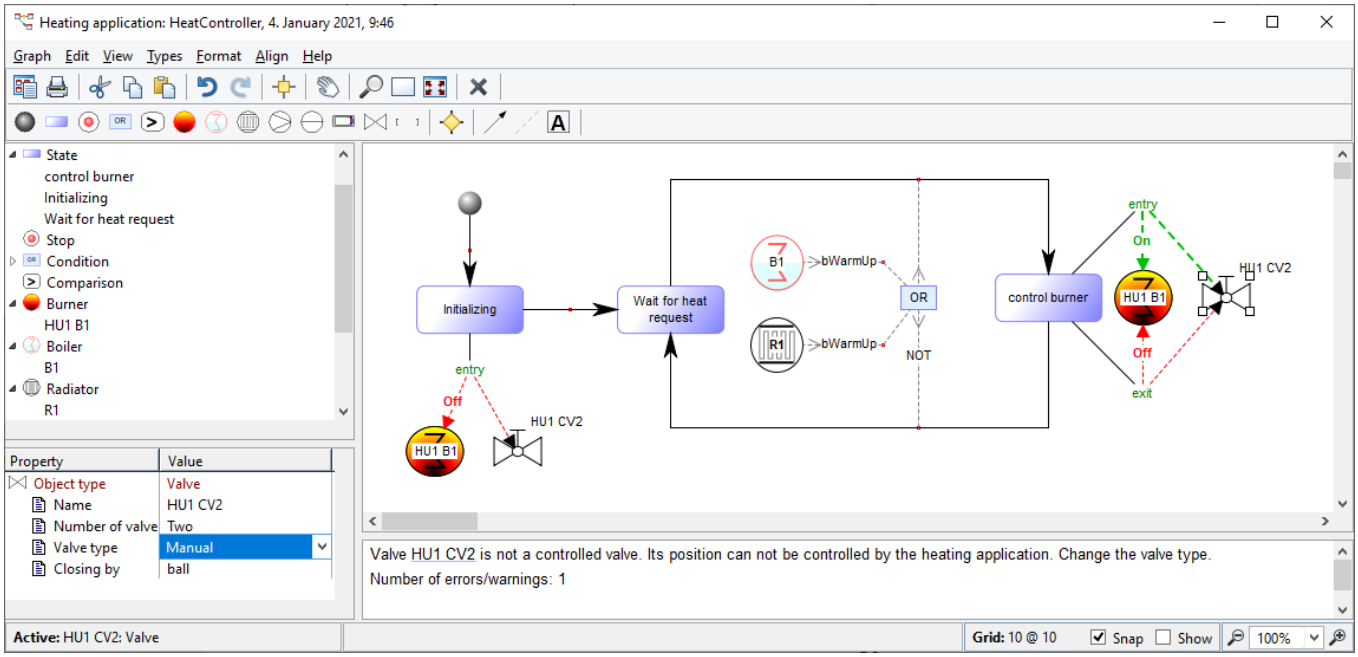

Fig. 4. Behavior of heat controller (defined in [14])

These combined views are also used for producing other artefacts needed in addition to the code. For example, code for simulators can be generated, enabling testing and verification of the system in situations that would be hard or dangerous in real life, like high gas pressures or using the system in very hot temperatures. Whether generating code for the production system or for simulation, the produced code is complete and ready to be executed with no need for manual modifications.

In addition to producing the code, the company applied the same domain-specific models as input for other artefacts needed, such as producing the sensor interface API, documentation, installation guidelines, and calculating the materials needed for the given system (e.g. total pipe lengths and types of insulation).

C. *Summary of the examples*

These two examples demonstrate how domain-specific modeling languages can be used to better integrate system engineering and software engineering tasks. More specifically in both examples:

- Teams can communicate using familiar terminology.
- Design work can be highly collaborative as both teams can see each other's work. Access controls can be set as desired, limiting the teams to read-only access to the other team's specifications, or allowing more freedom, e.g. letting engineers focusing on behavior (in Fig. 4.) also change the general system structure (in Fig. 3.).
- The feedback loop is fast and, in many cases, can be almost immediate: Changes from others can be seen without having to search or update manually – even in the same diagram that is being edited.
- Specification models covering systems and software are versioned together. This is a natural consequence of having joint language(s).
- Specifications based on domain-specific concepts can be precise enough that code, tests, deployment data and material needs can be produced from the models automatically – as done in the cases described above.

IV. Creating Integrated Languages

Language creation requires finding the right level of abstraction with relevant language concepts and then implementing them to get tooling support.

A. *Identifying language concepts*

Identifying the right level of abstraction is the most important phase. Although it can seem as if low-level language concepts (e.g., visualizing code with class diagrams) would be the easiest tactic, it is better to take concepts from the problem domain and thus raise the abstraction level. Describing things in problem domain terms instead of implementation concepts also enables the use of various generators from the same specification models.

Starting to seek candidate language concepts from the system level, like from physical items and structures, leads naturally to higher abstraction and to already known domain concepts. The pond in Fig 1. and various instruments connected via pipes in Fig. 3. are examples of deriving language concepts directly from physical system elements. For other sources of candidate language concepts see [5].

When identifying the language concepts, it is of key importance to focus on a narrow application domain, and on the actual needs for it rather than trying to make it work for any possible future need. It is worth repeating that the domain-specific language can be changed at any time a requirement changes – as opposite to waiting years before a new version of a modeling language is released from standardization organizations.

Other places to find suitable language concepts are the terminology used in the domain, the product architecture, and reused components and services. In other words, language should borrow from the domain-specific jargon or vocabulary already used. This vocabulary provides natural concepts that describe the systems in ways that people already understand. Starting from the existing vocabulary also means that there is no need to introduce a new, unfamiliar set of terms.

A good practice is to start with the mature, well-established concepts and then add the others incrementally. Often structural parts are easier to start with and then extend cover of the language towards behavioral parts. These behavioral parts often include elements from state machines, interaction diagrams, or flow diagrams, but with domain-specific extensions and links. Good tool support helps here by allowing reuse and the quick testing of different options.

B. *Implementing the languages within tooling*

While tool support can be obtained by developing the tool itself like any other software, a far more practical approach is using systems dedicated for the task of producing such tooling: Language Workbenches. At best, these tools reduce the total effort to a few working days [8][17]. Another important reason to use such tooling is that testing language specification early on becomes possible. In the longer run, these tools can also cover maintenance steps like automating language delivery to the users or update existing models when the language changes. Typically, the implementation includes the following four steps:

1) Define abstract syntax

Almost all modeling languages are specified via metamodels. A metamodel defines the grammar that language users apply. Depending on the way the language is specified, the metamodel covers at least all the concepts, their connections along with some constraints on how the language can be applied. The constraints can ensure that needed information is given, is in the correct form, or changes in one part or view keep others consistent.

A good metamodeling language also assists during language definition, helping categorize different kinds of language elements, like relationships among modeling objects or ports when connecting the objects as well as define sublanguages or other kind of integration among different languages. For a review of different language definition approaches see [7].

2) Set rules and constraints

Normally a plain metamodel cannot cover all the relevant rules and constraints. Also, the metamodel does not cover guidance or indicate changes that need to be considered while working on the models. Fig. 4. shows an example of one rule at the bottom of the editor: the behavior of controller is expected to open and close a valve 'HU1 CV2' but that valve was set to be only manually controlled in the system structure (Fig. 3.).

For this reason, the metamodel-based definition is extended with additional rules – typically defined in the formalisms applied by the tooling. The origin for the rules normally comes from the domain similarly with the rest of the language.

3) Define concrete syntax

Each modeling concept, such as objects, their connections, or even individual properties, has a visual representation so that humans can create, read and validate the models. Moody's [15] work on the Physics of Notation provides a basis for creating notations that are easy to read, remember and validate. Consider for example the notation in Fig. 3 using symbols for pipes and instruments that make the model easier to read.

Normally the concrete syntax sets one symbol per language element, but there can be different representations in different sublanguages of the same element [5], or additional information on error annotation, warnings, guidance etc. depending on the visualization needs [6].

4) Define generators

The final step is defining the generators. This is best done last because generators operate on and navigate through the models based on language concepts and structures, which must thus be ready first. The generators can produce code, tests, configuration, deployment instructions, bills of materials, documentation etc. as needed. Generators may also be applied to produce other models (possibly for other tools) as well as transform external data like libraries, code or test results back to models.

Generators are crucial for improved productivity and quality as automation removes time-consuming and error-prone manual tasks. Once the language has raised the level of abstraction, several kinds of artefacts can be produced directly from the same model. The relevant data is defined – and checked and maintained – only once in the model. Compare this to creating and maintaining all the artefacts manually.

V. CONCLUSIONS

All too often, system engineering and software engineering work is kept separate with limited possibilities for fast feedback and integration of the work by the teams. Working in silos prevents checking of the whole product and tracing among system and software engineering artefacts, and limits the collaboration that is critical to any successful development work. It also forces versioning of items separately. Everyone working in systems and software who has used languages like SysML or UML has experienced these limitations.

Domain-specific languages make a difference: By narrowing the language support to the product the company or its team is developing, integration of the development work becomes possible. In this article we presented examples from practice in which joint development of system engineering and software engineering was achieved. As in the examples, with suitable modeling support the teams can communicate using familiar terminology and work in collaboration. The feedback loop is fast and, as in the cases described, almost immediate: Changes from others can be seen in near real-time – even in the same diagram that is currently been edited. Another major benefit is that specifications based on domain-specific concepts are precise enough so that code, tests, deployment data and material needs can be automatically produced from the models.

Experience from industrial use has shown that with the fastest tools it takes on average two weeks to create specific

modeling support for company specific needs (for various cases, see [17]). The process of language implementation is also quite well guided and supported by tools, and today the most challenging part is identifying the right level of abstractions. There are now many examples that language developers can look at for inspiration, e.g. [2] shows dozens of public examples.

REFERENCES

[1] AUTOSAR, https://www.autosar.org/ [Accessed 4 Jan 2022]

[2] DSM Forum, http://www.dsmforum.org/ [Accessed 4 Jan 2022]

[3] H. Blom, D. Chen, K. Kaijser, H. Lönn, Y. Papadopoulos, M. Reiser, R.T. Kolagari, S. Tucci, "EAST-ADL: An Architecture Description Language for Automotive Software-intensive Systems in the Light of Recent use and Research". In: International Journal of System Dynamics Applications, 2016

[4] ISO Functional Safety, 26262-1, 2018

[5] S. Kelly, J-P. Tolvanen, Domain-Specific Modeling: Enabling full code genration, Wiley, 2008

[6] Kelly, S., Tolvanen, J.-P., Automated Annotations in Domain-Specific Models: Analysis of 23 Cases. FPVM 2021: 1st International Workshop on Foundations and Practice of Visual Modeling, 2021

[7] H. Kern, A. Hummel, S. Kühne. Towards a comparative analysis of meta-metamodels. In Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11 (SPLASH '11 Workshops). Association for Computing Machinery, New York, NY, USA, 2011

[8] A. El Kouhen, C. Dumoulin, S. Gérard and P. Boulet. Evaluation of Modelling Tools Adaptation. CNRS HAL hal-00706701, 2012. http://tinyurl.com/gerard12

[9] A. Nyßen, P, Könemann,. Model-based Automotive Software Development using Autosar, UML, and Domain-Specific Languages, Embedded World Conference, 2013.

[10] Omg.org, System Modeling Language, version 1.6. [online] Available at: https://www.omg.org/spec/SysML/, 2019 [Accessed 4 Jan 2022]

[11] Omg.org, Unified Modeling Language, version 2.5.1. [online] Available at: https://www.omg.org/spec/UML/, 2017 [Accessed 4 Jan 2022]

[12] B. Sari, Fail-Operational Safety Architecture for ADAS/AD Systems and a Model-driven Approach for Dependent Failure Analysis. Springer, 2020.

[13] J., Whittle, J. Hutchinson, M. Rouncefield, The State of Practice in Model-Driven Engineering, IEEE Software, vol.31, no.3, 2014.

[14] MetaCase, MetaEdit+ User's Guide. [Online]. Available at: https://metacase.com/support/55/manuals/, 2018 [Accessed 4 Jan 2022].

[15] D. Moody, "The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," in IEEE Transactions on Software Engineering, vol. 35, no. 6, 2009.

[16] J.-P. Tolvanen, S. Kelly, "Model-Driven Development Challenges and Solutions Experiences with Domain-Specific Modelling in Industry". In: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016), ScitePress, 2016

[17] J.-P. Tolvanen, S. Kelly, "Effort Used to Create Domain-Specific Modeling Languages". In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. ACM, 2018.