Reducing Design Time and Promoting Evolvability using Domain-specific Languages in an Industrial Context

Benny Akesson*, Jozef Hooman*†, Jack Sleuters*, and Adrian Yankov‡
*ESI (TNO), Eindhoven, the Netherlands
†Radboud University, Nijmegen, the Netherlands
‡Altran, Eindhoven, the Netherlands

***Abstract*—The complexity of contemporary systems is increasing, driven by integration of more functionality and trends towards mass-customization. This has resulted in complex systems with many variants that require long time to develop and are difficult to adapt to changing requirements and introduction of new technology. New methodologies are hence required to reduce development time, simplify customization for a particular customer, and improve evolvability both during development and after deployment.**

This chapter explains how these challenges are addressed by an approach to model-based engineering (MBE) based on domain-specific languages (DSLs). However, applying the approach in industry has resulted in 5 technical research questions, namely how to: RQ1) achieve modularity and reuse in a DSL eco-system, RQ2) achieve consistency between model and realizations, RQ3) manage an evolving DSL eco-system, RQ4) ensure model quality, RQ5) ensure quality of generated code. The five research questions are explored in the context of the published state-of-the-art, as well as practically investigated through a case study from the defense domain.

***Index Terms*—Model-based Engineering, Domain-specific Languages, Evolvability, Simulation, Validation, Co-evolution, Modularity**

I. Introduction

Development of contemporary systems is becoming increasingly complex, time consuming and expensive. This happens in response to a number of trends. Firstly, more and more dependent software and hardware components are being integrated to realize a wider range of functionality. Increased integration results in systems with complex behaviors that are difficult to design and validate, increasing development time. This problem is exacerbated by an increasing *system diversity* due to recent trends towards mass-customization of systems [1], which increasingly creates situations where every manufactured system has a unique hardware configuration and feature set. Lastly, system requirements frequently change as new technology is being introduced, or because of new expectations from the market. This means that substantial effort goes into reengineering systems to ensure they match customer needs throughout their life-cycle.

These trends in development of complex systems result in three key challenges:

C1) Development time needs to be shortened to reduce cost and time-to-market.
C2) Systems must be quick and easy to customize for a particular customer to manage increasing diversity.
C3) System functionality must be evolvable to assert that it continuously matches the needs of the customer during its life-cycle.

Model-based Engineering (MBE) is an engineering approach where models play an important role in managing complexity by providing abstractions of the system that separate the problem domain from the implementation technologies of the solution space. This has helped bringing development closer to domain experts, enabling them to express their ideas using familiar notations from their domain and automatically generate system artifacts, such as documentation, simulation models, and production code [2]–[4]. MBE can take many forms as there is a plethora of development methodologies used in industry with as much as 40 modeling languages and 100 tools being reported as commonly used [5]. The most commonly used modeling languages, at least in the embedded systems domain, are UML and SysML for software engineering and system engineering, respectively [6], [7]. However, domain-specific languages (DSLs) are becoming increasingly prevalent in narrow and well-understood domains [4].

This chapter is an experience report about addressing the increasing system complexity in an industrial context using an MBE development approach based on DSLs. First, Section II discusses how DSL technology addresses the three complexity challenges mentioned above, as well as stating five technical research questions related to this approach. Section III then discusses the five research questions in context of the published state-of-the-art to determine the extent to which they are recognized in literature and identify the range of available solutions. We continue in Section IV by presenting our approach to practically investigate the research questions in the context of a case study from the defense domain. The design of a DSL eco-system developed for this case study is discussed in Section V, after which we explain how the research questions were addressed by the case study in Section VI. Section VII presents an intermediate evaluation of the work before we end the chapter by discussing conclusions in Section VIII.

II. Domain-specific Languages

Determining whether a particular design methodology is a good fit for a given problem is not easy. This problem has also been recognized in the context of MBE [8], [9]. This section discusses how an MBE methodology based on DSLs addresses each of the three challenges outlined in Section I and presents

five research questions related to the approach that will be investigated through a literature study (Section III) and a case study (Section VI).

A. Reducing Development Time (C1)

Compared to general-purpose programming, DSL-based development approaches require an initial investment in terms of effort [6], [7]. This investment involves defining the abstract and concrete syntaxes of the language and implementing model validation, as well as model-to-text transformations that can generate artifacts for all supported variants. However, once this investment has been done, development time is ideally reduced, resulting in return on investment on longer term if sufficiently many model instances are created [6]. Note that this assumes that the considered variants are not so different that they constantly require the DSL and its transformations to be extended, limiting reuse and increasing development effort. For this reason, DSLs are particularly well-suited in the context of (mass-)customization, since a potentially large number of variants are needed that fit within the confines defined by product lines. In this context, which is the context of this work, DSLs efficiently address Challenge C1.

B. Improved Customization (C2)

Customization of a system or component can also be addressed using feature models [10]. However, a limitation of feature models is that they are context-free grammars that can only specify a bounded space that is known a priori. This means that feature models are only suitable for a restricted form of customization, i.e., selecting a valid combination of features that are known up front [11]. However, it is not possible to use a feature model to specify new features that were not previously considered at an abstract level. If this is necessary, an alternative approach is to specify variability using general-purpose programming languages, which are fully flexible, but expose low-level implementation details and do not separate the problem space and solution space. DSLs bridge the gap between feature models and general-purpose programming languages, as they are recursive context-free grammars that can specify new behavior from an unbounded space, while keeping problem space and solution space separate [11]. DSL technology is hence a good fit for systems with a high degree of variability, addressing Challenge C2.

C. Improved Evolvability (C3)

Since DSLs make it quick and easy to customize systems or components by modifying model instances of DSLs and then generate artifacts, it also follows that the instance can be easily modified, and artifacts regenerated if the system is evolved, e.g. due to changing requirements. In contrast, a change in the underlying implementation technology does not require DSL instances to be changed, as the specification in terms of domain concepts has not changed. Instead, changes to implementation technology implies a change only in the model-to-text transformations that express the semantics of the models, e.g. in terms of code. This provides a rather clean separation of concerns, which is reflected in surveys [9], [12] and case studies [6] listing improved flexibility, reactivity to changes, and portability as benefits of DSLs. DSL technology hence also addresses Challenge C3.

D. Industrial Research Questions

The above reasoning suggests that a design methodology based on MBE and DSLs might be suitable to address all three challenges stated in Section I. However, any design methodology has its drawbacks and it is essential to make sure that these do not offset the benefits [5]. Although there are surveys suggesting that the benefits of MBE often outweigh the drawbacks [7], leading to adoption of the approach, there are also examples of the opposite [4], [13]. A credible business case hence has to be built on a case-by-case basis. It is widely recognized that this involves not only technical, but also organizational and social, considerations [4], [5], [13]–[15]. A list of 14 industrial research questions, or pains, can be found in [16]. These research questions are based on experiences from our partner companies that are active in different application domains in the high-tech industry, e.g., defense, healthcare and manufacturing. In this work, we limit the scope to discuss a subset of five research questions that are relevant in the context of our case study.

RQ1) How do you achieve modularity and reuse in a DSL eco-system?
RQ2) How do you achieve consistency between model and realizations?
RQ3) How do you manage an evolving DSL eco-system?
RQ4) How do you ensure model quality?
RQ5) How do you ensure quality of generated code?

III. State-of-the-Art

This section continues by discussing the five research questions and the extent to which they are recognized in the state-of-the-art. We choose to focus on state-of-the-art work in an industrial context, i.e. empirical studies, case studies, and best practices in industry, and review the proposed solutions. We choose this focus to limit the discussion to relevant industrial problems and proven solutions. A broader exploration including more academic solutions is highly relevant, but is left as future work. Note that many of the research questions correspond to broad research areas and that an exhaustive discussion is outside the scope of this chapter.

A. Modularity and Reuse (RQ1)

Software engineering has seen great increases in productivity by enabling software to be decomposed into reusable modules that can be used as building blocks. This practice allows commonly used functionality to be implemented only once and then gradually mature as it is gradually reused, extended, and maintained. This same development is also desirable in language engineering. As DSLs evolve to cover a broader and broader domain, they inevitably reach the point where they need to be split into multiple modules or sub-languages to create a separation of concerns and reduce complexity. Since

multiple languages describing aspects of the same domain are likely to share common concepts, further modularization is often beneficial to enable reuse and improve maintainability [17]. Examples of this can be found in industrial case studies from a variety of domains [18]–[20]. In [20], it was reported that modularizing a large DSL into a number of sub-languages incurred an overhead of approximately 10% in terms of grammar rules and 5% in terms of lines of code. However, a great reduction of complexity was reported by separating concerns, as well as improvements in maintainability.

The widespread use of small DSLs that can serve as modules in a larger DSL eco-system, even within a single project, results in an integration challenge [5]. The available features for language composition vary significantly between different language workbenches and the meta-meta-models they support [21]. For example, composition features such as language extension/restriction where a base language is extended/restricted without modifying its implementation are quite common. In contrast, language unification that allows the implementation of both languages to be reused by only adding glue code is relatively rare [22]. It is hence clear that the problem of modularity and reuse is recognized and features to address it are considered differentiating features of existing language workbenches. We continue by briefly describing the language composition features available in Xtext, which is the language workbench used our in case study.

Xtext has quite limited and heavy-weight support for DSL modularity [20]. Each module is created as a DSL in its own right and results in five Eclipse projects being created. A DSL eco-system hence quickly contains tens to hundreds of Eclipse projects. Xtext supports *single inheritance* at the level of grammars, which works similarly to the concept of inheritance in many object-oriented programming languages. This feature enables language extension or specialization by overriding concepts in the inherited grammar. It also supports a feature called *mixin*, which allows the meta-model defined by another grammar to be imported and its elements referenced. However, it is not possible to use the imported *grammars* by referring to its rules, and the including language can thereby not use its syntax to create objects. This is only possible through inheritance. Lastly, Xtext also has a feature called *fragments* that allows frequently occurring rule fragments to be factored out and reused, reducing duplication and improving reuse and maintainability. However, this feature is limited to reuse within the particular grammar in which it was defined. In addition to the features supported directly by Xtext, it is shown in [20] how to creatively combine Xtext features to create a notion of interface-based modularity, where unassigned rule calls in Xtext can be used to create abstract rules that are later implemented by languages importing the grammar.

The language composition features offered by the language workbench affect the extent and at what granularity modularity and reuse happens. The limited language composition features provided by Xtext are sufficient to enable fine-grained reuse within a single grammar (fragments) and coarse-grained reuse between languages of the same DSL eco-system (inheritance and mixins). However, an implication of these features is that there is very limited reuse, at any granularity, between languages in different domains, i.e. different eco-systems. While this may sound natural, since DSLs are domain specific, not even common language concepts, such as expressions or concepts for date and time are typically reused. This shows that the equivalence of libraries in regular software engineering is missing from Xtext. Instead, reuse between languages in different domains often happens by copying and pasting rules and generator fragments from previous languages. Although it is stated in [23] that this type of reuse already goes a long way, we believe that further improvements to Xtext are necessary to achieve the required productivity and maintainability benefits offered by DSLs. *We hence conclude that for industrial cases where advanced language composition features are required, it may be worthwhile to consider other mature language workbenches than Xtext*. A suitable candidate in this case may be JetBrains MPS[1], where modularization and language composition are fundamental design concepts [24].

B. Consistency between Model and Realizations (RQ2)

The problem of inconsistencies between software artifacts is mentioned as a current challenge for MBE in [25], [26]. A concrete example of this is that software designs, modeled in languages like UML, often quickly becomes forgotten and inconsistent once development starts. This problem may occur for multiple reasons, one being that many practitioners do not take diagrams seriously and see them as doodles on the back of a napkin before the real implementation work starts with textual languages [27]. Another reason for inconsistencies is that many tools are not able to keep models at different levels of abstraction synchronized. This problem is recognized in a survey about MBE engineering practices in industry [4], which suggests that 35% of respondents spend significant time manually synchronizing models and code. The problem of manually synchronizing artifacts is also explicitly mentioned in a case study at General Motors [28]. In this case, lacking tool support for merging and diffing models, resulted in tedious and error-prone manual workarounds that would lead to inconsistent artifacts.

Consistency between models and realizations (or other artifacts) can bridged by generating all artifacts from a single source. In fact, this way of working is considered a best practice of MBE [15] and is a key benefit of MDE approaches that easily and efficiently support generation, which is a core purpose of DSLs. This benefit was explicitly highlighted in [6], where both code and documentation were generated from models specified using DSLs. This means that the model was always consistent with the generated artifacts. Similarly, [19] generates a simulation model, C++ code, visualizations, run-time monitoring facilities, and documentation that is consistent with an interface description based on a family of DSLs. *These works suggest that DSL technology is appropriate for ensuring consistency between model and realizations.*

[1]https://www.jetbrains.com/mps/

Generation of artifacts can ensure that they are always up-to-date with respect to the model, but this does not necessarily mean that they all correctly and consistently implement the semantics of the DSL. This is because the semantics of the DSL is typically hidden inside the generators and there are no simple ways to ensure that these semantics are consistent with each other [17]. This problem is addressed in [29], which combines formalizing (parts of) the semantics of a DSL with conformance testing to validate that these semantics are correctly implemented by generated artifacts, in this case code and an analysis model. The approach is demonstrated through case study using a DSL for collision prevention developed by Philips. A drawback of this approach is that it requires substantial effort (possibly years) and very particular expertise to formalize two non-trivial languages to the point where equivalence can be proven. Proofs furthermore often become (partially) invalid as models or generators change, making software evolution more costly and time-consuming. Using this approach to address RQ2 may hence exacerbate problems related to RQ3. For this reason, *formally proving semantic equivalence between realizations is not considered practical for complex industrial systems.*

C. Evolving DSL Eco-systems (RQ3)

Just like regular software, DSL eco-systems evolve over time. This may be in response to required changes in syntax, semantics, or both [30] as domain concepts are added, removed or modified. While evolution is often positive and helps the DSL stay relevant in a changing world, it creates a legacy of old artifacts, such as models, transformations, and possibly editors, that may no longer conform to the evolved meta-model and cannot be used unless they co-evolve [31]. This problem is well-recognized in the literature and is explained with examples from popular meta-models, such as UML and Business Process Model and Notation (BPMN), in [31]. Although it is possible to manually co-evolve models and transformations to reflect changes in the meta-model, this manual process becomes tedious, error-prone, and costly when the legacy is large [31], [32]. For example, the Control Architecture Reference Model (CARM) eco-system [33] developed at ASML consists of 22 DSLs, 95 QVT transformations, and 5500 unit test models to support development of those transformations. Co-evolving a DSL eco-system is more difficult than a single language, due to dependencies between its constituent parts [32]. Manually co-evolving a large industrial eco-system like CARM is hence not feasible in terms of time and effort, but requires extensive automation.

Co-evolution of meta-models and artifacts has been an active research topic for many years. A list of 13 relevant aspects that can be used to classify co-evolution approaches, such as the type of artifact they consider or the technique used to determine the evolution specification for the meta-model, is presented in [31]. Furthermore, an overview of five existing representative co-evolution approaches and a classification using the 13 aspects is presented. Together, the five presented approaches cover co-evolution of all artifacts, i.e. models, transformations, as well as editors. No precise conclusions are drawn about the state of existing tools. However, it is suggested that there is no single tool that adequately considers all cases of co-evolution and that dealing with the problem requires modelers to learn use different tools and techniques to co-evolve their artifacts. It is mentioned that co-evolution of transformations is intrinsically more difficult than models, which is reflected in the availability of mature approaches. In rest of this section, we focus on co-evolution of models, an easier problem for which industrial strength tools exist. For example approaches for co-evolution of model transformations, refer to e.g. [34], [35]. Other interesting aspects of evolution, such as its impact on code generation are relevant and challenging, but outside the scope of this work.

Apart from manual co-evolution of artifacts, there are four (semi-)automated approaches for obtaining an evolution specification [36], [37]: 1) *Operator-based* approaches, where evolution of the meta-model is manually specified in terms of reusable operations representing frequently occurring patterns of evolution. Based on the specified sequence of operators, a co-evolution specification for artifacts can be automatically derived. The usability of this approach is to a large extent determined by the completeness of libraries with reusable operators. Edapt[2], the standard co-evolution tool for the Eclipse Modeling Framework (EMF), previously known as COPE [38], is a prominent example of a well-known tool in this category. 2) *Recording* approaches that record modifications to the meta-model and automatically creates a specification reflecting the performed changes. This approach is also supported by Edapt. 3) *State-based differencing* approaches that compare the original and evolved version of the meta-model and derives an approximate specification of the changes. Example approaches in this category include EMFMigrate [39] and EMFCompare[3]. 4) *By-example* approaches [40], where the user manually migrates a number of model instances and the specification is derived by looking at the changes. The strengths and weaknesses of these four approaches are further discussed in [36].

The mentioned methods for co-evolution apply to co-evolution of models that have been manually specified by a user. However, another method applies to models that have been automatically created using static or dynamic techniques for model inference, e.g. using the Symphony process [41]. In this case, it may be faster to simply update the software creating the models to comply with the new meta-model and just rerun it to infer the models again. Of course, this method assumes that the data from which the models are inferred is stored.

It is clear that several methods and tools exist to address the co-evolution problem, although there are only limited studies that evaluate their applicability in the context of industrial DSL eco-systems. The extent to which Edapt could be used

[2]https://www.eclipse.org/edapt/

[3]https://www.eclipse.org/emf/compare/

to perform DSL/model co-evolution in the CARM eco-system was investigated in [42]. It was concluded that the standard operators could fully support 72% of the changes, with another 4% being partially supported. Implementing a set of model-specific operators increased the supported changes to around 98%. With these extensions, the authors concluded that Edapt is suitable for maintenance of DSLs in an industrial context. Further extensions to improve the usability of Edapt in industry have also been proposed in [43]. *Based on this evidence, we consider Edapt and its extensions relevant candidates for managing evolution of EMF-based meta-models in cases where the workflow used to modify the meta-models supports the usage of such tools.*

Determining the required operators by means of case studies, even on a large DSL eco-system like CARM, is not necessarily sufficient to make statements about the suitability for other eco-systems. This was shown in [37], where a theoretically complete operator library for specifying any sequence of evolutionary steps for the EMF meta-meta-model was derived. This investigation showed that state-of-the-art operator libraries could only specify 89% of DSL evolutions and that most of the remaining deficiencies could not be identified using a case study of the CARM eco-system.

D. Ensuring Model Quality (RQ4)

If models are used as the sole source of all generated artifacts, it is essential to validate models to ensure their correctness. In addition, it is frequently stated as a best practice to test and find defects as early as possible [44], since this has been shown to increase quality and reduce the total time and effort required to develop or maintain software [6], [45].

There are several ways to improve the quality of models and ensure correctness. For DSLs, a good starting point is to use the validation features of the language workbench. Features for model validation exist in all language workbenches, although the supported validation features vary [21]. Validation of structure and naming in model instances are relatively common features, while built-in support for type checking is less commonly supported. Many language workbenches have a programmatic interface allowing domain-specific validation routines to be implemented to make sure the model makes sense in the domain where it will be used, which is considered a best practice [44].

A more refined approach to model validation may involve tools and methods external to the language workbench. In [14], the quality and correctness of models is established by simulating the models against an executable test suite. The methodology proposed in [46] generates POOSL [47] simulation models connected via a socket to custom-made visualization tools for the considered system. The main benefit of this is that it helps make interactions between components and the behavior of the system explicit to reach an early agreement between stakeholders. Another example is to generate formal models to validate domain properties. For this purpose, the work in [48], [49] generated satisfiability modulo theories problems that were solved by an external solver. The results from this solver were then fed back into the validation framework of the language workbench to interactively notify the user directly in the development environment. Lastly, one best practice is to review models, just like source code [44]. This is currently done by many practitioners to build confidence in the quality of code generators and the generated code [45], [50], [51]. Based on this brief review, *we conclude that suitable validation methods are available both internally in language workbenches and through external tools.* The exact choice of method, as well as criteria for validation, is highly problem specific and should be determined on a case-by-case basis.

E. Quality of Generated Code (RQ5)

Quality of generated code is a very broad research question, since software quality can mean a lot of different things [52]. A tertiary study, i.e., a study of literature surveys, in the area of quality in MBE is presented in [53]. The study considers as many as 22 literature surveys, many of which choose maintainability as the quality metric of choice. They conclude that the field is not yet fully mature as most surveys target researchers and focus on classifying work, rather than targeting industry practitioners and aggregating quantitative evidence according to established quality metrics. We proceed by discussing a few relevant primary studies, most of which conclude that code generation leads to quality improvements.

A case study [6] in the Dutch IT-industry showed that introducing MBE in the maintenance phase of a software project improves software quality. More specifically, they showed that a lower defect density was achieved using modeling, although at the expense of increasing time to fix a defect. However, the total result of these effects was a decrease in the total effort spent on maintenance of versions of the software. A reduction of defects is also claimed in [23], [54], although the latter does not substantiate this with any quantitative evidence. A similar observation was made by Motorola in [14], which states that it is sometimes faster and sometimes slower to find the root cause of a software defect using MBE. They also provide quantitative estimates suggesting a reduction in the time to fix defects encountered during system integration, overall reduction of defects, and improvements in phase containment of defects (i.e. that defects are more likely to be detected and fixed in the development phase in which they are introduced) and productivity.

Motorola also points out a problem related to code quality using MBE. They state that code generation using off-the-shelf code generators can become a performance bottleneck unless it is possible to customize the generation [14]. The problem of generated code not being of desirable quality is also recognized in surveys with more than 100 participants [7], [26]. In the most recent of the two surveys, 21% of participants is negative or partially negative about the quality of the resulting code [7]. While this number shows that there are practitioners that are not satisfied with the quality of generated code, the number of practitioners that are neutral (30%) or (highly) positive (49%) is much higher. From this, *we conclude that the quality of generated code is a problem worth investi-*

gating further, especially for performance-critical applications designed with tools that provide little or no control over code generation.

Another aspect of generated code quality is the extent to which it is readable by humans. Best practices state that generated code should follow acceptable style guides. This may seem like a waste of time, since other best practices suggest that generated code should not be modified [44]. However, people still benefit from readable code in several ways: 1) Just like for any other code, generated code is inspected by developers trying to track down the root cause of a defect and this goes faster if the meaning of the code is clear. 2) Manual code reviews of generated code are part of the development practice in many places to ensure correctness of the code and its generators [45], [50], [55]. 3) Readable code provides an exit strategy in case the company decides to stop using MBE by simply checking in the generated code and continue using it manually [50]. A case where this did not work out was reported in [2], where generated Simulink code was not human readable, making the adoption of MBE hard to roll back.

Testing is an essential way to ensure the quality of software. However, code generation complicates testing, since there are often many possible paths through the code generator. There are two fundamental approaches to address this issue. The first approach implies *testing the code generator* itself. The challenge with this approach is to achieve sufficient coverage of the possible paths. Testing all possible (combinations of) paths through the generator is typically not feasible, due to the combinatorial explosion of possibilities. However, it may be possible to exercise all possible control flows in the code generator (i.e. every outcome of every single if statement), or use Pairwise Independent Combinatorial Testing[4] to exercise pairwise combinations of control flows. If the desired number of test models is too large to generate manually, a generator can be implemented to generate models that trigger the appropriate paths through the code generator.

The second approach is to ignore the code generator and *test the generated code*. In this case, all existing testing practices remain valid, but testing needs to be repeated for each generated variant. Although this suggests that the overall testing effort is increased, it is important to recognize that the quality of the generated code increases over time as the generator matures. This approach is common for software in safety-critical domains, such as healthcare, automotive and avionics, since it is often more practical and cost-effective to certify generated code than trying to qualify the code generator itself [55].

IV. Approach to Practical Investigation

Having discussed the relevant state-of-the-art for the five research questions related to our MBE approach based on DSLs, we proceed by explaining the organization of the practical investigation into these questions. We start by motivating our choice of modeling technology, before presenting our case study from the defense domain.

A. Modeling Technology

The five research questions in this work are practically investigated using an MBE approach based on DSLs. This is because Section II suggested that DSL technology is an intuitive fit with the three challenges stated in Section I, assuming the five stated research questions could be answered. The review of the state-of-the-art in Section III suggests that there are promising solutions that answer many of those questions. In addition, we have many years of experience of transferring DSL technology to industry and applying it in different domains, e.g., [18], [19], [56], [57]. There are many approaches [58] and tools [21] for developing DSLs. This work uses Xtext, which is a mature language workbench that has been around for more than a decade and has relatively high coverage in terms of important features for language development [21]. It is additionally open source and available as a plugin for the Eclipse IDE, one of the most commonly used tools for MBE [7]. Generators are defined in the Xtend language, which is a DSL built on top of Java that can be combined with regular Java code. Details on how to develop DSLs and generators based on Xtext and Xtend can be found in [59].

B. Case Study

A suitable case study is needed to drive the practical investigation into the five industrial research questions stated in Section II. We start by presenting the general context of our case study from the defense domain. This study is centered around the engagement chain of a Combat Management System, shown in Figure 1. This work considers a single ship, referred to as the *own ship*, with a number of sensors, e.g., surveillance radars and tracking radars, and a number of effectors to counteract possible threats.

The engagement chain consists of a number of steps that execute periodically, e.g., every few seconds. The input to this chain is the current state of the world. In the first step, surveillance radars are observing the world and produce sensor tracks, which can be intuitively understood as a radar blip with a position and speed corresponding to e.g., another ship, a missile, or a jet. The sensor track is then passed on to a track management process that fuses sensor tracks from multiple sensors to generate a single, more accurate, system track. The system tracks are sent to the threat evaluation process, which determines the types of threats, investigates their intentions, and produces a ranking that indicates the relative threat level. A sorted list of threats is then sent to the engagement planning process, which determines the combinations of sensors and effectors that should be used against each threat and at what time. Depending on the choice of planning algorithm, it may plan actions to counteract the threats strictly following threat ranking, or it may plan more flexibly using the ranking as a guideline. The generated engagement plan is then executed,

[4]https://github.com/Microsoft/pict

Fig. 1. Overview of the engagement chain

and the actions of the sensors and effectors close the loop by interacting with the world, affecting its state.

For this case study, we have implemented two DSLs (partially) corresponding to two steps of the engagement chain in Figure 1. The first DSL covers threat ranking, which is a part of threat evaluation, and the second covers engagement planning.

V. DSL Eco-system Design

This section presents the design of the DSL eco-system that was developed for the case study driving our investigation into the five research questions. First, we explain the rationale behind the design of this eco-system, followed by a description of the Threat Ranking DSL. This particular DSL was chosen because it is the smallest and conceptually simplest language to discuss, allowing us to describe it in limited space, yet give a feeling for the level of abstraction chosen in this work. A complete and detailed description of the entire eco-system is left as future work.

A. Design Rationale

In terms of the classification of DSL development patterns in [58], this work used informal domain analysis, primarily based on discussions with relevant domain experts and architects, to identify suitable domain models for the different components. The design of the languages followed the language invention pattern, i.e., new DSLs were designed from scratch. The two DSLs were developed one at a time, starting with Threat Ranking, to incrementally build trust in the overall approach and evaluate its benefits and drawbacks [16].

The DSL design process was incremental and iterative through a series of meetings with domain experts and architects, being the main technical stakeholders. The meetings discussed relevant domain concepts and possible variation points in the languages. After the meetings, there was a formal design phase where we prototyped the DSL by specifying the abstract and concrete syntaxes through a grammar in Extended Backus-Naur form (EBNF), which is the starting point for DSL design in Xtext. The proposed grammar and a few example instances were then discussed in the following meeting along with new possible concepts and variation points that could be introduced in the next iteration. This process was repeated until the languages were considered sufficiently expressive. Only at this point, generators with model-to-text transformations were implemented. In our experience, this incremental way of working with frequent prototypes helps drive development forward, as well as mitigate analysis paralysis [60].

```
JET assign level SEVERE
MISSILE assign level MODERATE
OTHER assign level NONE

If JET isInbound then INCREASE level
If ANY distance < 1 km then assign level CRITICAL

Weight a = 1.5
Weight b = 0.9
Metric custom = a * keepOutRange + b * lethality
Tiebreaker: custom higherIsMoreDangerous

Objective: protectOwnShip
```

Fig. 2. Example instance of the Threat Ranking DSL

In terms of implementation pattern, we used the compiler / application generator approach to translate constructs of our DSL to existing languages. This choice of implementation pattern was motivated by the desire to enable analysis and validation of DSL instances, as well as being able to tailor the notation to the specific domain. In that sense, the choice of implementation pattern is consistent with recommendations in [58]. Since the intended users of the language are domain experts and system engineers, rather than software developers, it was decided that the language should look and read more like text than code. This means some extra keywords have been added to make it easier to read better and understand, at expense of slightly longer specifications. This is not expected to be an issue as specifications are quite short. It was also decided to give the languages a common look and feel by using the same structure and notation, wherever possible.

B. Threat Ranking DSL

This section aims to give a feeling for the DSLs created in this work by discussing the concepts of the Threat Ranking DSL in the context of an example instance. The basic idea behind our Threat Ranking DSL is to assign priority levels to each threat and to use a tiebreaker metric to resolve the order in which threats with the same priority level are ranked. As seen in Figure 2, instead of using numbers to indicate priority, we use six threat levels, going from higher to lower: CRITICAL, SEVERE, SUBSTANTIAL, MODERATE, LOW, and NONE. The first five levels (CRITICAL to LOW) indicate threats that will appear in the output threat ranking, while threats with the last level (NONE) are filtered out and are not considered for engagements. The benefit of this use of threat levels over priority levels represented by numbers is that it ties into an existing classification that is used in the domain.

Threat levels are assigned in two ways in the language: 1) statically per threat type (e.g., JET and MISSILE), and 2) dynamically per individual threat. The static assignment associates each threat type with a threat level that initially applies to all threats of that type. The proposed DSL requires all threat types to have a statically assigned threat level and is hence a common feature among all instances. To facilitate this in a simple way without explicitly listing all 10 currently

supported threat types, the types OTHER and ANY have been introduced. ANY covers all types, whereas OTHER captures all threat types that have not been listed (i.e., neither explicitly or by an ANY).

The static threat level assignment can be dynamically modified per threat during each execution of the Threat Ranking algorithm based on properties of the threat at that particular time, e.g. kinematic information or the distance to the own ship. This is done using optional if-statements, making this a variable feature of the language. Values representing distances, speeds or times are required to have an appropriate unit to improve readability and remove ambiguity that can lead to incorrect implementation. A number of units are available in each category, allowing the user to choose whatever feels more natural. Behind the scenes, the generators convert all values into common units, i.e. meters for distances, seconds for time, and meters per second for speed.

The DSL instance in Figure 2 contains two examples dynamic threat level modifications. First, it states that any inbound jet, i.e., a jet flying towards the own ship, should have its threat level increased by one step, i.e., from SEVERE to CRITICAL in this case. This is an example of a *relative threat level assignment*, as the resulting threat level depends on the level before this assignment. Secondly, it states that any threat that is less than 1 km from the own ship should have its level reassigned to CRITICAL. This is an *absolute threat level assignment* that is independent of the previous threat level. It is possible to have any number of if-statements and they are executed in order. If a relative INCREASE or DECREASE of the threat level is done on a threat with the highest or lowest threat level, respectively, the level remains unchanged.

All threats will be assigned a final threat level based on the combination of static and dynamic threat level assignments. To arrive at a final ranking, the order in which to rank threats with the same threat level must be decided. This is done by either choosing any of 9 pre-defined tiebreaker metrics or by specifying a custom metric as an expression consisting of different threat properties, such as kinematic information (e.g., speeds and distances). The latter possibility vastly increases the possibilities for how to rank threats with the same threat level, as custom metrics can specify arbitrarily complex expressions, which as previously discussed in Section II form an unbounded space of behaviors that cannot be captured by feature models. This is a key argument for modeling the Threat Ranking algorithms using DSLs. The example instance in Figure 2 defines a new metric as a weighted combination of the specified keep out range and the lethality of the threat type. For each metric, it is possible to indicate whether a higher or a lower value is more dangerous.

Lastly, there is the concept of a High Value Unit (HVU), which is a critical unit, e.g., a cargo ship or an aircraft carrier, that may require protection by the own ship. The DSL is extended with the ability to specify an objective related to an HVU, i.e., to protect the HVU, protect the own ship, or protect both.

In conclusion, the presented Threat Ranking DSL defines threat ranking algorithm at a high level of abstraction using terminology from the application domain, which is commonly considered a best practice [13], [44], [61]. The DSL is furthermore so narrowly defined that it is impossible to use it to model different domains, which is a useful test to determine if the right balance between generic and specific has been found [60].

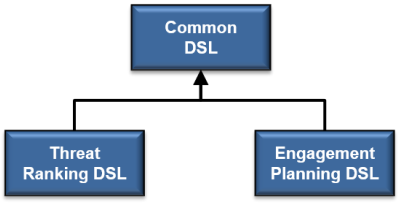

Fig. 3. DSL eco-system for parts of a Combat Management System.

VI. Results of Practical Investigation

This section explains the techniques employed and lessons learned from applying our MBE approach using DSLs to the case study with the goal of addressing the five industrial problems highlighted in Section II. Note that a less complex DSL eco-system than e.g. the CARM [33] eco-system sufficies for our case study, which may impact some of the conclusions in this section. We proceed by discussing each research question in turn.

A. Modularity and Reuse (RQ1)

The structural design of the DSL eco-system developed for the case study is shown in Figure 3. The eco-system comprises three DSLs, one for each of the two considered functions in the Combat Management System, Threat Ranking and Engagement Planning, and an additional language that factors out common domain concepts that are shared among the other two languages. Examples of concepts that are shared between the languages are expressions, units, objectives, metrics, threat types and threat properties. As suggested by the figure, the languages are composed by means of Xtext's single inheritance mechanism, where Threat Ranking DSL and Engagement Planning DSL are both inheriting the grammars of Common DSL. Our eco-system is neither making use of Xtext's mixin feature, nor the fragment feature. Mixins are not used because there is no need for either Threat Ranking DSL or Engagement Planning DSL to refer to the meta-model of the other language. Fragments are not used as there are no repetitive patterns in the rules of any of the individual grammars.

Although the language composition features of Xtext are limited, we conclude that they are sufficient for the needs encountered during our case study. However, it is easy to see the world through the limitations of tools [60] and it is possible that another language workbench with more and lighter-weight language composition feature would have encouraged us to modularize at finer granularity to enable reuse of, e.g. expressions and units in other languages outside this work.

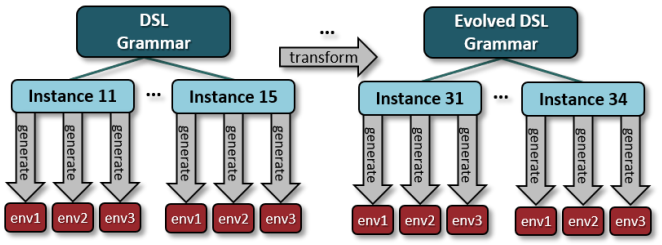

Fig. 4. Generation of models for multiple simulation environments, as well as production code. Model-to-text transformations migrate DSL instances to newer versions as the DSL grammars evolve.

B. Consistency between Model and Realizations (RQ2)

The proposed DSL-based approach to MBE generates both simulation models and code, reflecting the observation in [7] that code generation and simulation are the most common uses of models in the embedded domain. The DSL instance is the sole source of truth from which both simulations models and code are generated, following best practices from [15]. Code generators have been implemented for the relevant programming and modelling languages and their execution environments (env), corresponding to different simulators at different levels of abstraction, or the system itself. This ensures that simulation models and production code are always consistent with their corresponding DSL instance, as illustrated in Figure 4. The fact that multiple artifacts are generated from a single DSL instance means that our approach is consistent with "the rule of two", i.e. that DSL instances should be used for at least two different purposes to fully benefit from a model-based design approach [15]. We do currently not any generate documentation from the DSL instance, but an additional generator could be implemented to generate documentation using LaTeX. There are also available tools for automatic generation of Word documents, e.g. Gendoc[5] and m2doc[6]. However, we leave this as future work.

Generation of artifacts from a single source does not ensure that the semantics of the DSL is consistently implemented in the generators. This means that simulation models and production code may be consistent with the model, but have inconsistent views on what the model actually means. In our particular case, a consistent interpretation of the semantics implies that generated simulation models and production code always produce the same ranking, given the same input. Ensuring consistent semantics hence boils down to validating that this is really the case. The challenge is that the models and code execute in different environments that use different languages and model some system components at different levels of abstraction. As a result, even if the exact same Threat Ranking algorithm is used in all environments, the inputs of the Threat Ranking component are not expected to be the same. For example, a threat may be detected slightly earlier or later, impacting the set of threats to rank at a particular point in time, which in turn affects the scheduled engagements and the set of threats later in the scenario. For this reason, it is not always possible to compare results across environments and draw meaningful conclusions about consistency of semantics between generators.

Our solution to mitigate this concern is to remove the differences in environment and execute all implementations in a single execution environment. This is achieved by wrapping the generated production code and run it in one of the simulation environments as software-in-the-loop, which ensures that all generated implementations have the same inputs and that all other components are implemented identically. This in turn enables us to establish the consistency of semantics between generators by extensive regression testing through comparison of results, following the recommendation in [17]. Over time, through extensive testing and use, this approach builds confidence that the different generators implement the same semantics of the DSL and hence that the Threat Ranking component works correctly.

There are two main drawbacks of this approach: 1) The semantics are implemented in each generator and any semantic change must hence be consistently implemented in all generators manually, which is error prone. 2) It is limited to cases where the exact same result is expected from all realizations, or slightly more generally, where results maximally differ by some known maximum bound, which is often not the case. An interesting option could be to specify the semantics on an abstract level and generate implementations that are consistent by construction. This would ensure that changes in semantics would only be made in a single place, which could be advantageous for highly evolvable systems. This direction is considered future work.

C. Evolving DSL Eco-systems (RQ3)

The DSL eco-system in our case study consists of three DSLs (Threat Ranking DSL, Engagement Planning DSL, and Common DSL), four model-to-text transformations that generate simulation models and production code in a variety of languages, as well as an analysis model for custom metrics, described later in Section VI-D. Our DSL eco-system is hence considerably smaller than the CARM eco-system [33], previously discussed in Section III-C, resulting in a smaller legacy as meta-models evolve. The evolution of our eco-system has taken place entirely during the development phase of the DSLs, which has been approximately two years.

We considered using Edapt [42], since it is relatively mature and has been positively evaluated for other industrial DSL eco-systems. However, Edapt works directly with the EMF meta-model, which Xtext generates from the specified grammars. The grammars will hence become inconsistent with the evolved meta-model, unless a formal link is established that propagates the changes to the grammar. To the best of our knowledge, there is currently no tooling implementing this link, removing Edapt from further consideration in this work. Instead, we focused on solutions available within Xtext itself. We only needed to consider co-evolution of models and

[5]https://www.eclipse.org/gendoc/
[6]http://www.m2doc.org/

transformations, since Xtext regenerates the editor based after changes to the meta-model. Due to the limited evolvability burden in our case study, we have opted for the simplest option that satisfied our needs. This involved manually implementing a generator with a model-to-text transformation whose input was a model conforming to the non-evolved grammar and output a textual representation of that instance conforming to the evolved grammar, as shown in Figure 4. This corresponds to a largely manual approach, as the mapping between concepts in the non-evolved and evolved meta-model, as well their semantics, was done manually. However, the implemented transformation could quickly and easily be applied in a batch run to evolve all existing model instances. This simple approach is hence not limited to eco-systems with a few models, but is primarily restricted by the number and complexity of the languages in the eco-system and the complexity of their dependencies.

D. Ensuring Model Quality (RQ4)

In the proposed DSL-based approach to MBE, the model is the sole source of truth from which both simulations models and code are ultimately generated, following the best practice from [15]. It is hence important that the quality of these models is high and that any problems are detected as early as possible. Towards this, we experimented with three ways to improve model quality:

1) The Eclipse-based IDE for the Threat Ranking DSL, which is automatically generated from the DSL grammar by Xtext, ensures syntactic correctness and immediately validates that the syntax of an instance complies with the grammar.
2) A number of model validation rules have been implemented that exploit knowledge about the domain to detect problems with instances. These validation rules can either lead to warnings, which only alert the user but still allows generation of artifacts, or to errors, which prevent the generators from running altogether until the problem is resolved. This is generally a good place to address deprecation issues as the DSL is evolving. A warning can be triggered when a deprecated construct is encountered in a model, assuming an appropriate model transformation is available to map it to an equivalent construct in the evolved DSL. In contrast, if a model transformation is not available (anymore), an error is triggered.
   More specifically for our Threat Ranking DSL, one validation rule triggers a warning if there are multiple static threat level assignments to a single threat type to alert the user that only the last assignment is useful. In contrast, another rule throws an error in case not all threat types have a static threat level assignment, since this violates a fundamental assumption of the ranking algorithm. Yet another validation rule checks the correctness of units, i.e., that metrics related to time or distance are only compared to values whose units relate to time and distance, respectively. This prevents comparing apples to pears, or more literally, seconds to meters by raising an error. For many of these validation rules, quick fixes were built into the editor to help the developer resolve violations quickly and reliably.
3) An analysis tool was also implemented in a generator that immediately produces a report providing visibility on the results provided by custom metrics, previously introduced in Section V-B, without having to run the simulator. The generated report is based on a single given list of threats to be ordered. Representative lists of threats are easily obtained by recording inputs to the Threat Ranking components during simulation. The report, shown in Figure 5, demonstrates how the custom tiebreaker metric is computed for each threat. This immediately shows the user an example outcome when applying the metric and gives insight into what caused that outcome. For example, it could show that a particular parameter is typically dominating the metric and that weights should be adjusted to make the metric achieve the desired goal. This is particularly helpful when experimenting with complex custom metrics.

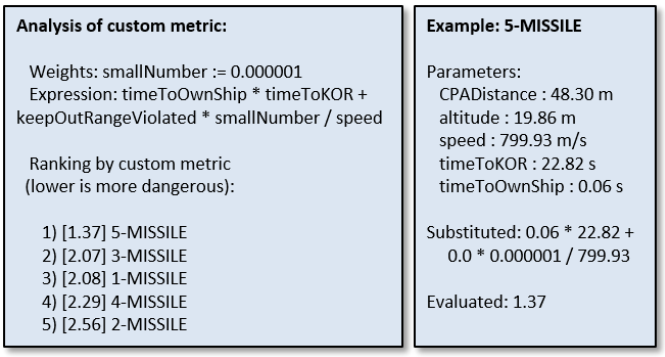

Fig. 5. Generated analysis showing result of applying a custom metric to a particular set of threats. The numbers in the example are not indicative of any real systems.

E. Quality of Generated Code (RQ5)

Most of our practical work related to quality of generated code is related to testing, which is done at three different levels: 1) unit testing, 2) component testing, and 3) integration testing. Unit testing of the DSLs follows the method described in [59] and performs low-level validation of the generators by asserting that particular model constructs result in the expected code being generated. In contrast, the component-level testing focuses on the semantics of the generated code and validates that this is consistent across implementations, as previously described in Section VI-B. Note that comparing results from multiple implementations is useful to validate consistency, but it does not necessarily imply that any implementation is correct. However, following this approach, all implementations must provide the same incorrect result in order for it to pass the test, which is rather unlikely. Lastly, we perform integration testing in the complete system to verify that components communicate correctly and that system-level results, such as when and where threats are neutralized in a particular scenario

for a given DSL instance, do not unexpectedly change during development. For this purpose, golden reference results have been generated for relevant threat scenarios and DSL test instances and are used for comparison during testing. To further increase confidence in the results, different generators can be implemented by independent developers based on a common specification, following requirements for certification of software components in safety-critical avionics systems [62].

Manual validation is tedious and time-consuming labor, especially when software is being developed in parallel on many different branches. Following the GitFlow workflow[7], our repository has two main branches, master and development. New features are developed on feature branches that are integrated into development after passing testing at all three levels mentioned above. Despite passing all tests, the development branch contains newly integrated experimental features and is not considered perfectly stable. Once it is time to make a new release, additional manual validation is done on this branch and once it is considered to be sufficiently stable for users, it is integrated into the master branch.

To reduce the manual effort of all commits on these branches, testing has been automated to make it possible to run all combinations (or a chosen subset) of DSL instances and scenarios by pushing a single button. As recommended in [44], the DSL instances used for testing have been designed in such a way that they exercise as many constructs of the DSLs as possible to improve coverage. Since we are preparing for a situation where the DSL itself evolves over time, it is important that integration testing is always done with the latest versions of the eco-system and its generators. However, Xtext does not support automatic generation of a command line DSL parser and generator that can be used for integration testing after each commit. As a contribution of this work, we have defined a method for automatic generation of such a command tool that can be used with any Xtext project without even requiring an Eclipse installation. A description of this method and an example project is available online[8].

To automate all aspects of testing and enable *continuous integration*, we have set up a Jenkins Automation server that checks out the latest version of the code after each commit on any branch, builds the compiler and runs all tests. General benefits of this setup include enabling defects to be caught early, improving phase containment of defects, and ensuring that only the latest changes must be reviewed and debugged when a bug is detected. There are also specific benefits with respect to Challenge 3. For example, it allows DSL instances of deployed systems to be stored in branches and continuously validate that evolved versions of the DSL do not accidentally change their behavior. This makes it easier to maintain and upgrade systems after deployment.

The automation server also provides *continuous deployment* by automatically generating Eclipse plugins based on both the development branch and master branch for Threat Ranking DSL and Engagement Planning DSL as soon as a commit to either of these branches has passed all tests. These plugins are then made available on an internal update site. Experienced users or developers can hence subscribe to the latest development version and experiment with the latest features, while regular users can subscribe to the latest stable version. This setup ensures that the plugins used by both of these communities are always up-to-date.

Unlike Motorola [14] and some of the survey participants in [7], [26], previously discussed in Section III-E, we have not experienced problems with the performance of generated code during our case study. The two main reasons for this are: 1) our models are relatively small, making them less prone to performance problems, and 2) DSL development using Xtext gives full control over the model-to-text transformations used for code generation, which means that the differences with hand-written code are typically small. When these differences do occur, it is mostly to simplify the structure of the generators and avoid complex control flows that slow down development and complicate testing.

VII. Evaluation

As a part of an informal evaluation of an intermediate version of the DSL eco-system, an event was organized on the premises of the industrial partner where about 20 employees with various functions ranging from software and system engineers to domain experts and even a director participated. Some of the participants were familiar with the domain from before, but many of them were not. The event consisted of a 30-minute introduction after which participants were divided into four groups that experimented with the DSL eco-system. After a short tutorial that explained the basics of the DSLs and the associated tooling for simulation and visualization, the teams were tasked with using the DSL eco-system to solve a particular assignment. It turns out that a short tutorial was sufficient to get three out of the four groups to productively experiment with making their own model instances to solve the assignment, at which point we only needed to answer a few simple questions, e.g., about the definitions of keywords in the language. The last of the four groups completed the tutorial, but did not get off the ground with making their own instances. This was due to a combination of lacking motivation, insufficient domain knowledge, and group dynamics.

The feedback from the participants was largely positive. Some participants had domain knowledge and suggested features that could be included in future versions of the DSL. It was also reported that the participants found experimenting with the DSL an effective way to learn about the domain, as the DSL and associated tooling made it and easy to customize, deploy, and evaluate model instances. This suggests that our DSL was on its way towards delivering on Challenges C2 and C3. This feedback also resonates with the claim that MBE empowers users without software background, e.g. domain experts and system engineers, by enabling them to work productively without having to rely on software engineers to implement their ideas [2]–[4].

[7]https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[8]https://github.com/basilfx/xtext-standalone-maven-build

As the work was concluded, a final evaluation was organized to assess the potential of the DSL-based development methodology. The goal of the evaluation was to let a number of intended DSL users experience with the DSL way-of-working and assess its potential within the organization. The means to achieve this was to let them experiment with the DSL eco-system. These experiments took place in the intended application context, in this case together with suitable tools for simulation and visualization. Although this is a specific example of a DSL in context, the participants were asked to assess the general potential of DSLs and not limit themselves to the particular DSL or the domain of engagement planning. 10 participants considered representative for the potential users of DSLs were asked to join the evaluation. Some participants in the evaluation were system engineers/architects with only limited software development experience, corresponding to the primary audience of the developed DSL eco-system. Others had experience with implementing algorithms directly in general-purpose programming languages and could hence provide a complementary perspective. The setup of the final evaluation was nearly identical to that of the intermediate evaluation, but featured newer versions of the DSL eco-system and the tutorial to reflect improvements made during the six months between the two events.

The participants identified a number of classic gains of DSLs during the session, e.g.: 1) the demonstrated DSL-based environment was easy to use and accessible to non-technical people, 2) the DSL hides the implementation technology, allowing the problem to be decoupled from its implementation, 3) DSLs enable faster customization and prototyping, at least of variants that fit within the boundaries of the language, 4) DSLs may improve communication within a group of people, but also with the outside world. These observations relate to known benefits of DSLs, previously discussed in Section II, and together they address all three challenges identified in Section I. This is an encouraging result! A number of pains were also identified during the evaluation, e.g. DSLs require higher upfront investment compared to traditional software development, modeling requires different skills, and adopting a DSL-based methodology requires organizational support. These pains have been previously identified and mitigation techniques have been documented and shared with the industrial partner.

VIII. Conclusions

This chapter addressed the problem of reducing design time and improving evolvability of complex systems through a Model-based Engineering (MBE) approach based on Domain-specific Languages (DSLs). Five research questions raised by our industrial partner related to the approach were investigated by means of a literature study and a practical case study from the defense domain, namely how to: RQ1) achieve modularity and reuse in a DSL eco-system, RQ2) achieve consistency between model and realizations, RQ3) manage an evolving DSL eco-system, RQ4) ensure model quality, RQ5) ensure quality of generated code.

A DSL eco-system with two DSLs inheriting common concepts from a third language was developed for the case study. The eco-system also features four model-to-text transformations to generate an analysis report and code for a variety of programming and modeling languages. Further transformations have been developed to support migration of models as the DSL eco-system evolves. We discussed how the generated analysis report and model validation rules help ensuring correctness of models and how the quality of code generated by the eco-system is improved using continuous integration and continuous deployment practices.

Both intermediate and final evaluation results suggest that the proposed DSL-based development methodology and example DSL eco-system deliver on their design goals and addresses the aforementioned challenges for complex systems. A number of relevant pains related to DSL-based development were explicitly identified by the users of the eco-system, but they were already known and had been discussed along with existing mitigation techniques within the company. Based on this work, next steps involve the industry partner deciding whether the gains of DSL-based development outweigh the pains, and for what application domains the gains are maximized.

References

[1] H. Geelen, A. van der Hoogt, W. Leibbrandt, and F. Beenker, "HTSM Roadmap Embedded Systems," 2018.

[2] J. Aranda, D. Damian, and A. Borici, "Transition to model-driven engineering," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 692–708.

[3] H. Burden, R. Heldal, and J. Whittle, "Comparing and contrasting model-driven engineering at three large companies," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 14.

[4] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, 2014.

[5] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2014.

[6] N. Mellegård, A. Ferwerda, K. Lind, R. Heldal, and M. R. Chaudron, "Impact of introducing domain-specific modelling in software maintenance: An industrial case study," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 245–260, 2016.

[7] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice," *Software & Systems Modeling*, vol. 17, no. 1, pp. 91–113, 2018.

[8] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "A taxonomy of tool-related issues affecting the adoption of model-driven engineering," *Software & Systems Modeling*, vol. 16, no. 2, pp. 313–331, 2017.

[9] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proceedings of the 33rd international conference on software engineering*. ACM, 2011, pp. 471–480.

[10] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability management with feature models," *Science of Computer Programming*, vol. 53, no. 3, pp. 333–352, 2004.

[11] M. Voelter and E. Visser, "Product line engineering using domain-specific languages," in *Software Product Line Conference (SPLC), 15th International*. IEEE, 2011, pp. 70–79.

[12] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio, "Relevance, benefits, and problems of software modelling and model driven techniques – a survey in the Italian industry," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2110–2126, 2013.

[13] D. Wile, "Lessons learned from real DSL experiments," *Sci. Comput. Program.*, vol. 51, no. 3, pp. 265–290, Jun. 2004.

[14] P. Baker, S. Loh, and F. Weil, "Model-driven engineering in a large industrial context - Motorola case study," *Model Driven Engineering Languages and Systems*, pp. 476–491, 2005.

[15] P. F. Smith, S. M. Prabhu, and J. Friedman, "Best practices for establishing a model-based design culture," SAE Technical Paper, Tech. Rep., 2007.

[16] B. Akesson, J. Hooman, R. Dekker, W. Ekkelkamp, and B. Stottelaar, "Pain-mitigation techniques for model-based engineering using domain-specific languages," in *Proc. Special Session on Model Management And Analytics (MOMA3N)*, 2018, pp. 752–764.

[17] M. Voelter, "Architecture as language," *IEEE Software*, vol. 27, no. 2, pp. 56–64, March 2010.

[18] J. Verriet, L. Buit, R. Doornbos, B. Huijbrechts, K. Sevo, J. Sleuters, and M. Verberkt, "Virtual prototyping of large-scale IoT control systems using domain-specific languages," in *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, 2019.

[19] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman, "Integrating interface modeling and analysis in an industrial setting." in *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, 2017, pp. 345–352.

[20] C. Rieger, M. Westerkamp, and H. Kuchen, "Challenges and opportunities of modularizing textual domain-specific languages," in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, 2018, pp. 387–395.

[21] S. Erdweg, T. Van Der Storm, M. Voelter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems & Structures*, vol. 44, pp. 24–47, 2015.

[22] S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language composition untangled," in *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. ACM, 2012, p. 7.

[23] F. Hermans, M. Pinzger, and A. Van Deursen, "Domain-specific languages in practice: A user study on the success factors," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 423–437.

[24] M. Voelter, *Language and IDE Modularization and Composition with MPS*. Springer Berlin Heidelberg, 2013, pp. 383–430.

[25] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. C. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, J. Kienzle, M. Schöttle, F. Steimann, D. Stikkolorum, and J. Whittle, *The Relevance of Model-Driven Engineering Thirty Years from Now*. Cham: Springer International Publishing, 2014, pp. 183–200.

[26] A. Forward and T. C. Lethbridge, "Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals," in *Proceedings of the 2008 international workshop on Models in software engineering*. ACM, May 2008, pp. 27–32.

[27] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of" semantics"?" *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[28] A. Kuhn, G. C. Murphy, and C. A. Thompson, "An exploratory study of forces and frictions affecting large-scale model-driven development," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 352–367.

[29] S. Keshishzadeh and A. J. Mooij, "Formalizing and testing the consistency of DSL transformations," *Formal Aspects of Computing*, vol. 28, no. 2, pp. 181–206, 2016.

[30] J. Mengerink, L. van der Sanden, B. Cappers, A. Serebrenik, R. Schiffelers, and M. van den Brand, "Exploring DSL evolutionary patterns in practice: a study of DSL evolution in a large-scale industrial DSL repository," in *6th International Confenrence on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, 2018.

[31] D. Di Ruscio, L. Iovino, and A. Pierantonio, "Coupled evolution in model-driven engineering," *IEEE software*, vol. 29, no. 6, pp. 78–84, 2012.

[32] J. Mengerink, R. Schiffelers, A. Serebrenik, and M. van den Brand, "DSL/model co-evolution in industrial EMF-based MDSE ecosystems," in *ME@ MODELS*, 2016, pp. 2–7.

[33] R. R. Schiffelers, W. Alberts, and J. P. Voeten, "Model-based specification, analysis and synthesis of servo controllers for lithoscanners," in *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*. ACM, 2012, pp. 55–60.

[34] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Dealing with the coupled evolution of metamodels and model-to-text transformations." in *ME@ MoDELS*, 2014, pp. 22–31.

[35] J. García, O. Diaz, and M. Azanza, "Model transformation co-evolution: A semi-automatic approach," in *International Conference on Software Language Engineering*. Springer, 2012, pp. 144–163.

[36] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, "An analysis of approaches to model migration," in *Proc. Joint MoDSE-MCCM Workshop*, 2009, pp. 6–15.

[37] J. Mengerink, A. Serebrenik, R. R. Schiffelers, and M. van den Brand, "A complete operator library for DSL evolution specification," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 144–154.

[38] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "COPE - automating coupled evolution of metamodels and models," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 52–76.

[39] J. Di Rocco, L. Iovino, and A. Pierantonio, "Bridging state-based differencing and co-evolution," in *Proceedings of the 6th International Workshop on Models and Evolution*. ACM, 2012, pp. 15–20.

[40] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Model transformation by-example: a survey of the first wave," in *Conceptual Modelling and Its Theoretical Foundations*. Springer, 2012, pp. 197–215.

[41] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva, "Symphony: View-driven software architecture reconstruction," in *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*. IEEE, 2004, pp. 122–132.

[42] Y. Vissers, J. G. M. Mengerink, R. R. H. Schiffelers, A. Serebrenik, and M. A. Reniers, "Maintenance of specification models in industry using Edapt," in *2016 Forum on Specification and Design Languages (FDL)*, Sept 2016, pp. 1–6.

[43] J. G. M. Mengerink, A. Serebrenik, M. van den Brand, and R. R. H. Schiffelers, "Udapt: Edapt extensions for industrial application," in *Proceedings of the 1st Industry Track on Software Language Engineering*, ser. ITSLE 2016. New York, NY, USA: ACM, 2016, pp. 21–22.

[44] M. Voelter, "Best practices for DSLs and model-driven development," *Journal of Object Technology*, vol. 8, no. 6, pp. 79–102, 2009.

[45] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, and J. Zimmermann, "What is the benefit of a model-based design of embedded software systems in the car industry?" *Emerging Technologies for the Evolution and Maintenance of Software Models*, pp. 343–369, 2012.

[46] J. Hooman, "Industrial application of formal models generated from domain specific languages," in *Theory and Practice of Formal Methods*. Springer, 2016, pp. 277–293.

[47] B. D. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten, and J. P. Voeten, "Software/hardware engineering with the parallel object-oriented specification language," in *proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. IEEE Computer Society, 2007, pp. 139–148.

[48] A. J. Mooij, J. Hooman, and R. Albers, "Early fault detection using design models for collision prevention in medical equipment," in *International Symposium on Foundations of Health Informatics Engineering and Systems*. Springer, 2013, pp. 170–187.

[49] S. Keshishzadeh, A. J. Mooij, and M. R. Mousavi, "Early fault detection in DSLs using SMT solving and automated debugging," in *International Conference on Software Engineering and Formal Methods*. Springer, 2013, pp. 182–196.

[50] A. J. Mooij, J. Hooman, and R. Albers, "Gaining industrial confidence for the introduction of domain-specific languages," in *Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 662–667.

[51] A. J. Mooij, G. Eggen, J. Hooman, and H. van Wezep, "Cost-effective industrial software rejuvenation using domain-specific models," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2015, pp. 66–81.

[52] International Organization for Standardization, *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. ISO, 2011.

[53] M. Goulão, V. Amaral, and M. Mernik, "Quality in model-driven engineering: a tertiary study," *Software Quality Journal*, vol. 3, no. 24, pp. 601–633, 2016.
[54] P. Mohagheghi and V. Dehlen, "Where is the proof? – a review of experiences from applying MDE in industry," *Lecture Notes in Computer Science*, vol. 5095, pp. 432–443, 2008.
[55] M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann, "Using language workbenches and domain-specific languages for safety-critical software development," *Software & Systems Modeling*, pp. 1–24, 2018.
[56] A. J. Mooij, M. M. Joy, G. Eggen, P. Janson, and A. Rădulescu, "Industrial software rejuvenation using open-source parsers," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2016, pp. 157–172.
[57] R. Doornbos, B. Huijbrechts, J. Sleuters, J. Verriet, K. Sevo, and M. Verberkt, "A domain model-centric approach for the development of large-scale office lighting systems," in *Complex Systems Design & Management (CSD&M) conference*. IEEE, 2018.
[58] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.
[59] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
[60] S. Kelly and R. Pohjonen, "Worst practices for domain-specific modeling," *IEEE software*, vol. 26, no. 4, 2009.
[61] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design guidelines for domain specific languages," *arXiv preprint arXiv:1409.2378*, 2014.
[62] RTCA, Inc., *RTCA/DO-178C*. U.S. Dept. of Transportation, Federal Aviation Administration, 2012.