Pain-mitigation Techniques for Model-based Engineering using Domain-specific Languages

Benny Akesson[1], Jozef Hooman[1,2], Roy Dekker[1,3], Willemien Ekkelkamp[1,3], and Bas Stottelaar[4]
[1]*Embedded Systems Innovation by TNO, Eindhoven, The Netherlands*
[2]*Radboud University, Nijmegen, The Netherlands*
[3]*Thales Nederland, Hengelo, The Netherlands*
[4]*Altran, Eindhoven, The Netherlands*
{*benny.akesson, jozef.hooman, bas.stottelaar*}*@tno.nl*, {*roy.dekker, willemien.ekkelkamp*}*@nl.thalesgroup.com*

Keywords: Model-based Engineering, Domain-specific Languages, Simulation, Validation, Threat Ranking

Abstract: Changing an established way of working can be a real headache. This is particularly true if there are high stakes involved, e.g., when changing the development process for complex systems. New design methods, such as model-based engineering (MBE) using domain-specific languages (DSLs) promise significant gains, such as cost reductions and improvements in productivity and product quality. However, transitioning between design methods comes with a great deal of uncertainty, as any approach has associated pains. While the gains may be intuitively appreciated, it may be less clear what the new pains will be and whether or not they will cancel out the gains. For this reason, it may sometimes feel safer to stick with the devil you know than to meet the one you do not, preventing the full design potential of the company from being reached.
This paper is an experience report from an investigation into how to mitigate the pains associated with a transition to a model-based design flow using DSLs. The main contributions of the paper are: 1) a list of 14 pains related to MBE as a technology that is representative of our industrial partners designing high-tech systems in different domains, 2) a selected subset of six pains is positioned with respect to the state-of-the-practice, 3) practical experiences and pain-mitigation techniques from applying a model-based design process using DSLs to an industrial case study, and 4) a list of three open issues that require further research.

1 INTRODUCTION

As systems get increasingly complex, design times go up and it becomes harder and more time-consuming to react to frequent changes in requirements or introduction of new technology. In most current document-based design flows, there is only limited reuse between different stages of design, increasing design time. Changes often result in confusing inconsistencies between different component *artifacts*, such as simulation models, production code and documentation. These issues can be tackled by a model-based engineering (MBE) approach using domain-specific languages (DSLs), where the different artifacts, can be generated and quickly regenerated from a domain-specific model, being the sole source of truth (Smith et al., 2007). This helps reducing design time and improves the evolvability of the system, as changes only have to be made in a single place, and consistency between artifacts is ensured as they are regenerated. While this makes the approach intuitively promising, it comes with its own set of associated pains. It is hence essential for a company to think carefully about the strengths and weaknesses of their organization and processes before transitioning their design flow to make sure that the pains do not offset the gains (Smith et al., 2007; Whittle et al., 2014).

This paper is an experience report about first steps towards transferring a model-based design approach using DSLs to a company in the defense domain. The scope of this work is limited to investigating the pains and possible mitigation techniques to assert that there are no immediate show-stoppers. Based on this investigation, the company will decide whether to take further steps towards a transfer. This work is successful if it helps the company make the right decision for how to proceed, no matter which decision this is.

The company has a largely document-based design flow, but uses a variety of models in different languages and at different levels of abstraction during the design process. For instance, different modeling techniques are used during early design-space exploration and detailed performance estimation in later phases. To reduce design time, it is desired to improve the continuity and reuse between the stages of the design process by creating an explicit relation be-

tween the different artifacts of a component. This will also make it easier to ensure the consistency between models at different levels of abstraction and their correspondence to the final implementation.

The four main contributions of this paper are: 1) a list of 14 pains related to MBE that is representative of our industrial partners designing high-tech systems in different domains. This list is useful for researchers and practitioners aiming to transfer or adopt model-based technologies, 2) a subset of six pains is positioned with respect to the state-of-the-practice to determine whether or not the pain is generally recognized in the literature and what the known pain-mitigation techniques are, 3) practical experiences and results from applying a model-based design process using DSLs that minimizes the anticipated pains and provides continuity and reuse between design phases in an industrial case study of a Threat Ranking component, and 4) a list of three open issues that require further research is presented, which may help guide future research in this area.

The rest of this paper is organized as follows. Section 2 presents the list of pains related to MBE and we choose six of these for further consideration in this work. Section 3 discusses the state-of-the-practice for the selected pains, before we continue in Section 4 by explaining our choice of technology, case study, and method for our practical investigation. The Threat Ranking DSL developed as a part of this work is presented in Section 5, after which we discuss the results from applying it to the selected pains in our case study in Section 6. Open issues are presented in Section 7, followed by conclusions in Section 8.

2 PAINS AND GAINS

The first step in this work is to identify the pains and gains relevant to MBE. Inspiration for these pains and gains is primarily taken from management processes, engineering practices, and interactions with experienced people from partner companies in different domains in the high-tech industry, e.g., defense, healthcare and manufacturing. This work focuses on pain-mitigation techniques for MBE and does not intend to present the identified gains for brevity. For empirical studies discussing the benefits of MBE and its industrial impact, refer to e.g., (Vetro et al., 2015; Torchiano et al., 2013; Mellegård et al., 2016). Note that the presented pains are not laws of nature that apply to all situations, but may also include concerns and objections from people who are just not convinced about the merits of MBE. Some of the pains are furthermore not exclusive to MBE, but are still raised in the context of a potential technology transfer. The presented list of pains is hence useful to any practitioner trying to transfer or adopt MBE technology, or academics that need to position their work with respect to industrial concerns.

The 14 pains below are related to MBE in a broad sense without considering a particular tool or method. The term model can hence refer to e.g., UML diagrams, DSL instances, and executable (simulation) models. It is important to recognize that introducing MBE in a company is not just a matter of technology, but is also widely recognized as an organizational and social challenge (Hutchinson et al., 2014; Whittle et al., 2014; Baker et al., 2005; Wile, 2004; Smith et al., 2007). However, these aspects are out of scope of this paper.

1. No continuity in the development process ("if everybody has their own tools or only covers part of the problem, there is no continuity in the process")
2. No proper modeling strategy ("models cannot solve everything; one needs to define goals / strategies for the modeling")
3. No management of tools ("different versions, backwards compatibility, etc.")
4. Too much dependency on tools ("more tool vendors - which may go bankrupt or are taken over - and more tool versions increase the possibility that models become unsupported or obsolete")
5. Difficult to deal with many possible system configurations ("many possible system configurations, because there are many optional components, many different instances of components, many different connections between components - how to model, test and simulate them?")
6. Issues and large effort when interfaces of components change ("how to deal with changing interfaces; consequences for the model, architecture, etc.?")
7. Difficult to deal with different versions of a component, variability within a component, and different models of a single component ("how to deal with different versions and models of components?")
8. No consistency between model and realization ("the model has to represent the product correctly; if the product changes, this has to be reflected in the model")
9. No consistency between models and documentation ("how to keep documents up-to-date after frequent changes to the model?")
10. Incorrect models ("how to ensure that the model is correct and gives the right outcome?)"
11. Large maintenance effort of models and generators ("how to arrange maintenance of models / generators?")
12. Code generation leads to low quality code ("what is the quality of generated code?")

13. Integration and testing of code, generated code, and models is difficult ("do the generated simulation models, generated code, and existing code work together properly?")
14. Confusion about the relation between results and versions of component models & tools ("keep track of input, versions, output, description of the models, etc.")

While all the pains represent valid concerns, we choose to focus on a subset of six pains that are most relevant to the company, our case study, and the considered modeling technology. However, since there is some overlap between the different pains, we will briefly touch upon a few others. The main considered pains are Pains 1, 7, 8, 10, 12, and 14.

3 STATE-OF-THE-PRACTICE

After selecting a subset of six pains for further consideration in this work, this section continues by positioning them with respect to the state-of-the-practice, i.e. empirical studies, case studies, and best practices in industry. We choose this focus to limit the discussion to relevant industrial problems and proven solutions. A broader exploration including academic solutions is highly relevant, but is left as future work. For Pains 1 and 14, we have not found relevant related work in an industrial context. In the rest of this section, we hence focus on Pains 7, 8, 10 and 12.

3.1 Different Models of a Component and Different Grammars (Pain 7)

Different versions of a component can be managed using existing source code control systems, such as Subversion or Git, which allow changes to be tracked between revisions and any revision can be retrieved from the system at any time. This approach works particularly well if the component-definition is text-based, which is the case for source code and many types of DSL instances.

Variability within a component can be addressed using feature models (Beuche et al., 2004). 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 suitable for configuration, i.e., selecting a valid combination of features that are known up front (Voelter and Visser, 2011). 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 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 (Voelter and Visser, 2011). DSLs hence seem like a promising technology for evolving systems with variability. However, while DSL technology may conveniently address the evolvability of components, a new challenge is to manage the evolution of the DSL itself, its generators, and models.

3.2 Consistency between Model and Realization (Pain 8)

Consistency between models and realization (or other artifacts) is a pain, unless it can be bridged by means of generation from a single source. In fact, this way of working is considered a best practice of MBE (Smith et al., 2007) 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 (Mellegård et al., 2016), 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, (Kurtev et al., 2017) 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 DSLs is a good choice of technology for our requirement of supporting multiple environments in a consistent manner.

3.3 Ensuring Model Quality (Pain 10)

If models are used as the sole source of truth and the 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 (Voelter, 2009), since this has been shown to increase quality and reduce the total time and effort required to develop or maintain software (Mellegård et al., 2016; Broy et al., 2012).

There are several ways to improve the quality of models and ensure correctness. One best practice is to review models, just like any other piece of software (Voelter, 2009). This is currently done by many practitioners to build confidence in the quality of code generators and the generated code (Broy et al., 2012; Mooij et al., 2013). In (Baker et al., 2005), the quality and correctness of models is established by simulating the models against an executable test suite. Another best practice is to use model-level validation (Voelter, 2009) to verify that the model is a valid instance of

the language, but perhaps more importantly, to validate that the model makes sense in the domain where it will be used.

3.4 Quality of Generated Code (Pain 12)

This pain is phrased rather broadly, since software quality can mean a lot of different things (International Organization for Standardization, 2011). A tertiary study, i.e., a study of literature surveys, in the area of quality in MBE is presented in (Goulão et al., 2016). 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 quality of generated code is actually a gain rather than a pain.

A case study (Mellegård et al., 2016) in the Dutch IT-industry showed that introducing MBE in the maintenance phase of a software project can improve 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 observed in (Mohagheghi and Dehlen, 2008), but it is not supported by any quantitative evidence. A similar observation was made by Motorola in (Baker et al., 2005), which states that it is sometimes faster and sometimes slower to find the root cause of a software defect when 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.

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 (Voelter, 2009). However, people still interact with generated code in several ways. For example, 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 it is clear what the code is doing. Secondly, manual code reviews of generated code are part of the development practice in many places to ensure correctness of the code and its generators (Broy et al., 2012; Mooij et al., 2013). Since code generators generate code in a structured way, this means that the confidence in their correctness is increased over time. This argument is consistent with a best practice stated in (Voelter, 2009). Lastly, for certification of safety-critical software in e.g., the automotive and avionics domains (RTCA, Inc., 2012), it may furthermore be more cost-efficient to manually inspect the code than to qualify the code generator, which is very expensive and time-consuming.

4 APPROACH TO INVESTIGATE MITIGATION OF PAINS

This section explains the organization of the practical investigation of the pains for an MBE approach based on DSLs in an industrial case study. We start by motivating our choice of modeling technology, before presenting the case study. Lastly, we present our approach to investigate pain-mitigation techniques.

4.1 Modeling Technology

The potential pains of MBE are investigated by means of DSLs, since the discussion in Section 3 suggests that it has the potential to successfully mitigate many of the chosen pains. It is also a technology that is already used within the partner company, and we have many years of experience of transferring it to industry and applying it, e.g., (Kurtev et al., 2017; Mooij et al., 2016). There are many approaches (Mernik et al., 2005) and tools (Erdweg et al., 2015) for developing DSLs. This work uses Xtext as DSL development tool. Xtext is a mature language workbench that has been around for more than a decade and has high coverage in terms of important features for language development (Erdweg et al., 2015). Xtext is additionally an open-source tool, which is available as a plugin for the Eclipse IDE. 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 (Bettini, 2016).

4.2 Case Study

A suitable case study is needed to investigate pain-mitigation techniques. We start by presenting the general context of our case study, being 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, such as missiles, guns, and countermeasures.

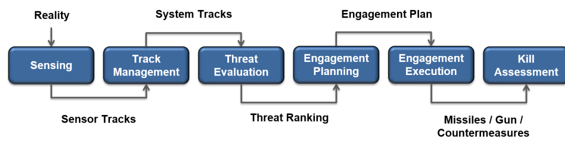

Figure 1: Overview of the engagement chain

The engagement chain consists of a number of steps that execute periodically, e.g., every few seconds. In the first step, surveillance radars are observing their environment 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 detected set of system tracks are sent to the threat evaluation process, which determines the type of threats and produces a ranking that indicates which threat is considered more dangerous. A list of ranked threats is then sent to the engagement planning process, which determines the combinations of sensors and effectors that should be used against each hostile threat and at what time, i.e., planning in both time and space. Depending on the choice of planning algorithm, it may plan engagements of threats strictly following threat ranking, or it may plan more flexibly using the ranking as a guideline. The generated engagement plan is then executed, followed by a kill assessment process that determines whether the threat has been neutralized or should be considered for reengagements.

For the purpose of our case study, we have selected the Threat Ranking component, which is the final step of the Threat Evaluation process. In essence, the Threat Ranking component gets a list of hostile threats and produces an ordered list of threats indicating the priority with which they should be considered for engagement by the engagement planner. This ranking can be produced in many ways using a wide range of different criteria that can be expressed as instances of a DSL. Threat Ranking is considered a suitable choice for our case study as it is a relatively small component, yet with sufficient variability to be interesting to model using a DSL. The size of the component is beneficial as it can be modeled with limited time, allowing multiple iterations of development, another best practice from (Voelter, 2009).

4.3 Organization of Investigation

The chosen modeling approach is applied to three phases of development with different target environments: 1) early design space exploration to identify candidate system configurations, where high-level simulation models of components are used to provide quick approximate results, 2) detailed performance estimation using high-fidelity simulation models to get accurate results for a single or a few candidate systems, and 3) execution of production code in the actual Combat Management System. Note that the Threat Ranking component runs a relatively simple algorithm that can be fully implemented in all three environments. This means that unlike e.g., a radar model, the different implementations of the Threat Ranking component do not have different levels of abstraction. The expectation is hence that all implementations of the algorithm should output the same ranking, given the same input and environment.

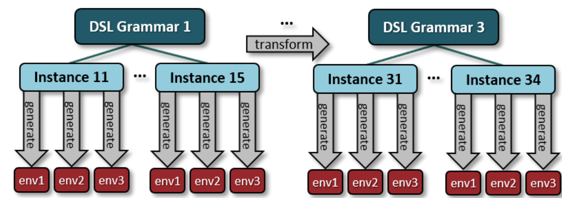

Figure 2: Overview of approach

To capture the impact of an evolving language, we choose to design and implement the Threat Ranking DSL in three steps. First, we create a baseline grammar with basic Threat Ranking concepts. We then extend this grammar twice and introduce additional concepts. More details about each of the three DSL grammars is provided in Section 5. An overview of our approach is shown in Figure 2, which shows several versions of the grammar, each with a number of model instances that have to be mapped to each of the three environments.

5 THREAT RANKING DSL

This section presents the Threat Ranking DSL developed in this work. As previously mentioned in Section 4.3, we first define a baseline grammar (referred to as Grammar 1) with basic Threat Ranking concepts. This grammar is then extended twice, resulting in Grammars 2 and 3, respectively, adding new concepts that increases the range of algorithms that can be expressed in the language.

Note that for reasons of confidentiality, the Threat Ranking process modeled in this work does not immediately correspond to any Threat Ranking component produced by the company, but captures the general spirit and complexity of such a component. Also remember that the goal of this work is to investigate pain-mitigation techniques for the selected pains and not to design the perfect DSL. The DSL presented in this section is hence only means to achieve the goal.

Next, we proceed by presenting each of the three grammars in more detail. Note that this is an overview and not a complete reference manual for the language.

```
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
```

Figure 3: An example instance using Grammar 3 of the Threat Ranking DSL

The following descriptions hence only cover key concepts and not the full expressivity of the DSL.

5.1 Design Rationale

In terms of the classification of DSL development patterns in (Mernik et al., 2005), this work used informal domain analysis, primarily based on discussions with relevant domain experts, to identify a suitable domain model. The design of the language was initially informal and followed the language invention pattern, i.e., a new DSL was designed from scratch. The design process primarily involved making a number of example instances demonstrating relevant use-cases at a suitable level of abstraction. This was succeeded by a formal design phase where the concrete syntax of the language was specified in Backus-Naur form (BNF), which is the starting point for DSL design in Xtext. 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 (Mernik et al., 2005). A design decision of the language is that it should read well as text to make it easy for domain experts to understand and discuss. This means some extra keywords have been added to make it read better, at expense of slightly longer specifications. This is not expected to be an issue as specifications are quite short.

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. This can be observed in the example instance shown in Figure 3. The example is an instance of Grammar 3 of the Threat Ranking DSL and will be discussed throughout this section.

5.2 Grammar 1 - Basic DSL Concepts

As seen in Figure 3, 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, which is commonly considered a best practice (Voelter, 2009; Karsai et al., 2014; Wile, 2004)

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 costly mistakes. 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 3 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 choosing any of 9 pre-defined tiebreaker metrics, including the distance from the threat to the own ship and the time to reach the closest point of approach (assuming a predicted trajectory). For each metric, it is possible to indicate whether a higher or a lower value is more dangerous. If this parameter is omitted, the default setting is that a lower value is more dangerous, since this intuitively holds for all pre-defined tiebreaker metrics except the speed of the threat. It is also possible to omit the tiebreaker metric altogether, in which case ties are broken in an unspecified way.

5.3 Grammar 2 - Threat Properties and Custom Metrics

The second DSL grammar extends the first in two ways. First, it adds additional static threat properties, such as the specified keep-out range and the estimated lethality of the threat. This means a number of new keywords were added to the language that removes the need for hard coding important values for properties in the DSL instance, as done with the keep-out range of 1 km for the type ANY in the example instance in Figure 3. Instead, the keywords can be used directly in the language, see e.g., the use of the keyword keepOutRange further down in the example, and the correct values are automatically provided to the simulation models or code when they are (re-)generated. This ensures that important properties can be modified in a single place in the generator, making it easier to accommodate changes.

The second addition allows custom tiebreaker metrics to be defined, vastly increasing the possibilities for how to rank threats with the same threat level. The example instance in Figure 3 defines a new metric as a weighted combination of the specified keep out range and the lethality of the threat type. For this metric, a higher value is more dangerous, as indicated in the definition of the tiebreaker metric.

5.4 Grammar 3 - High-value Units

The third grammar adds 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. The introduction of the objective means that the ranking process is generalized from considering every threat and its relation to the own ship to consider the relation to a number of reference tracks. If the objective is to protect the own ship, then the own ship is used as reference track to compute e.g., the time to reach the closest point of approach. Similarly, if the objective is to protect an HVU, then the HVU is used as reference track. In case the objective is to protect both, then they are both used as reference tracks, resulting in a tiebreaker metric for each reference. These metrics are then weighed to arrive at a final ranking.

6 RESULTS OF INVESTIGATION

This section explains the pain-mitigation techniques employed and lessons learned from applying an MBE approach using DSLs to the case study with the goal of addressing the six selected pains. We proceed by discussing each of the selected pains in turn.

6.1 Different Models of a Component and Different Grammars (Pain 7)

Variations within components are specified completely within the DSL and do not include the use of feature models. In fact, the custom metrics in our DSL can specify arbitrarily complex expressions containing kinematic information (e.g., speeds and distances) and other threat properties, which 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.

In our evolving DSL grammars, previously discussed in Section 5, we ensured that new concepts are optional and have default values when required if they are not specified. This is mentioned as a best practice for evolvable languages in (Voelter, 2009). This design implies that an instance of Grammar 1 is also a valid instance of Grammar 2 and Grammar 3. For example, an instance of Grammar 1 does not use any of extra static properties that are specified in Grammar 2, e.g., keep-out range and lethality, and is hence unaffected by their addition when the grammar is extended to Grammar 2. Similarly, it does not define an optional custom metric, but uses one of the pre-defined tiebreaker metrics from Grammar 1, which are still available in later versions of the grammar. Grammar 3 does introduce and require the concept of objective to produce a ranking, but uses a default setting to protect the own ship if it is not specified, which was the standard behavior in earlier versions of the DSL. In this way, the behavior of instances specified under Grammar 1 and Grammar 2 still behave as expected in the extended Grammar 3.

While extending a grammar with optional components and default settings is simple and convenient, it may not be possible, or even desirable, in all cases.

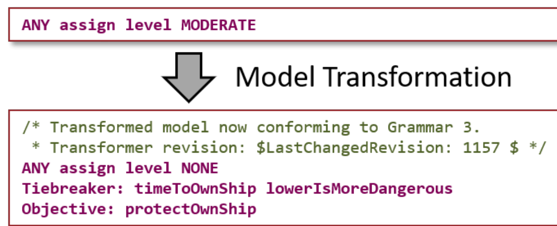

Figure 4: A minimal model is transformed to be compatible with Grammar 3 of the Threat Ranking DSL.

For example, this would never allow the concrete syntax of existing constructs to be modified and improved, which may be very limiting for a DSL that needs to evolve over long periods of time. A general solution to address this problem is to use model transformations that transform models of an earlier version of the grammar to a newer version. Implementing a model transformation is no different from implementing a generator that produces a simulation model or production code, since they are all examples of text generation. In this particular case, the generated text is just an instance of a newer version of the DSL. A simple example of a model transformation implemented in this work is shown in Figure 4. This transformation accepts instances of Grammar 1, Grammar 2 or Grammar 3 and outputs an instance compliant with Grammar 3, where all default settings are made explicit. For example, it explicitly writes out that the default tiebreaker metric in the implementation is the time to reach the own ship, where a lower value is considered more dangerous. Note that the transformed model clearly indicates which version of the grammar it is compliant with and which source code revision of the transformer was used to create it. This helps managing the evolution of the language (Voelter, 2009).

6.2 Consistency between Model and Realization (Pain 8)

In the proposed DSL-based approach to MBE, the model is the sole source of truth from which both simulations models and code are generated, following the best practice from (Smith et al., 2007). This was achieved by implementing code generators for the relevant languages and simulation models are hence always consistent with each other and with the production code. We do currently not any generate documentation, 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 and m2doc. However, we leave this as future work.

6.3 Ensuring Model Quality (Pain 10)

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 (Smith et al., 2007). 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 complains if the syntax of an instance does not comply 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 current version of the grammar. 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 4.2, without having to run the simulator. The generated report is based on a single given list of threats to be ordered. Realistic lists of threats are easily obtained by recording inputs to the Threat Rank-

Analysis of custom metric:

Weights: smallNumber := 0.000001
Expression: timeToOwnShip * timeToKOR + keepOutRangeViolated * smallNumber / speed

Ranking by custom metric (lower is more dangerous):

1) [1.37] 5-MISSILE
2) [2.07] 3-MISSILE
3) [2.08] 1-MISSILE
4) [2.29] 4-MISSILE
5) [2.56] 2-MISSILE

Example: 5-MISSILE

Parameters:
CPADistance : 48.30 m
altitude : 19.86 m
speed : 799.93 m/s
timeToKOR : 22.82 s
timeToOwnShip : 0.06 s

Substituted: 0.06 * 22.82 + 0.0 * 0.000001 / 799.93

Evaluated: 1.37

Figure 5: Generated analysis showing result of applying a custom metric to a particular set of threats.

ing components during simulation of threat scenarios. The report, shown in Figure 5, shows 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.

6.4 Relating Results to Versions of Models and Tools (Pain 14)

In the broader picture of a code base and tools that evolve over time, results of a simulation or an execution do not only depend on the inputs, such as the scenario and own ship configuration, but also on the source code revision and the version of simulators and other tools. This is problematic if questions ever arise over how a particular result was obtained. To be able to trace and reproduce results, it is important to keep track of which versions of what tools and source code were used to create them. This also means that previously used versions of tools must be archived after update in case they need to be used again.

To address this issue with the high-level simulation environment, we keep the source code of the simulator, the DSL grammars, instances of the language, and the scenario under version management using Subversion. Once results of a simulation are created, the revision of source code of the generator is stored in an accompanying file. This file also contains the version numbers of the simulator and other relevant tools. The file is stored together with the used ship configuration file and Threat Ranking DSL instance. Each generated artifact is furthermore annotated with the version of the generator that created it to improve traceability and debugging. Together, these pieces of information ensure that each deterministic simulation can be reproduced and that there is a clear link between results and the set up that was used to create them. Similar measures need to be taken for the production platform and any other environments that are used, but this is left as future work.

6.5 Quality of Generated Code (Pain 12)

Section 6.3 previously mentioned how we ensure model quality. This section considers the quality of generated code, which relates to quality of models, since our simulation models are code in either a general-purpose programming language or in an executable modeling language. To clearly distinguish the scope of these two pains, Section 6.3 considers ensuring quality at the level of DSL instances, before any generation, and this discusses the quality of generated artifacts, such as simulation models or production code.

As previously mentioned in Section 4.3, generated simulation models and production code should always produce the same ranking, given the same input. An important issue is hence how to validate 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 pain is to remove the differences in environment and execute all implementations in a single framework. 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 in semantics of the generators by extensive regression testing through comparison of results, following the recommendation in (Voelter, 2010). 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.

In addition to component-level testing, we also 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, are consistent across implementations and do not unexpectedly change during development. Note that comparing results from multiple implementations does not 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. 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 (RTCA, Inc., 2012).

Manual validation is tedious and time-consuming labor. To reduce this effort, validation 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 (Voelter, 2009), the DSL instances used for testing have been designed in such a way that they exercise as many constructs of the language 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 language 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 tool that can be used with any Xtext project to enable continuous integration. A description of this method and an example project is available online[1]. To automate all aspects of testing, we have set up a Jenkins Automation server that checks out the latest version of the code after each commit, builds the compiler and runs all tests. This enables defects to be caught early, improving phase containment of defects, and ensures that only the latest changes must be reviewed and debugged.

6.6 No Continuity in the Development Process (Pain 1)

When developing code and generators for the different environments, we noted three types of differences: 1) *architectural differences*, e.g., in the high-level simulation environment, the result of the tiebreaker was only used to determine the order of threats with the same threat level, while in the high-fidelity environment it is an integral part of a complete real-world scenario, 2) *differences in implementation*, e.g., different coordinate systems to represent positions or support for different threat types, 3) *differences in abstraction*, e.g., the high-fidelity simulation environment uses more advanced algorithms for track prediction. These types of differences are hardly surprising since different environments were designed by different people at different times for different purposes.

The differences in the first two categories were all minor and could be overcome by an adaptation layer that integrates the generated code into the respective environments. For differences in the last category, we had to think carefully if any of the differences affected the design of the language or the generators. In the case of Threat Ranking, we found that the differences in abstraction could be safely encapsulated in the respective environment. For example, it does not matter which type of track prediction is used to determine threat properties, such as the closest point of approach or the time to reach the own ship, as each environment can do this with their respective algorithms, just like before the introduction of DSLs. The generator only assumes that there is a way to access these properties in all environments, which was possible in our case.

Although we could bridge the differences and use a single Threat Ranking model for different environments corresponding to different stages of design, adapters and workarounds are not an ideal solution in the long term. We recommend carefully aligning concepts, architecture and implementation between environments if model-based engineering should be used as a continuous process in all stages of design. Ideally, we recommend using a single framework, as shown in Figure 6. The envisioned framework can be used in all phases of development and allows a flexible combination of abstract models, detailed models, code, and hardware. The upper layer in Figure 6 shows the generic infrastructure with three key components: 1) the simulation framework, 2) a repository of components, and 3) the reference architecture of the product. In the bottom layer, this infrastructure is used to develop a product.

This envisioned framework allows virtual prototypes to be quickly assembled by re-using existing models from the component repository and developing new models as necessary. The framework generates high-level simulation models of software (SW) and hardware (HW) components from the selected models, allowing the complete system to be simulated. This enables the company to reach an agreement on the ship configuration at an early stage of the design process. The benefit of this is that it reduces the specification phase of the system (and thereby design time) and reduces the risk of changes to the system once the development has started.

Once a system has been specified, it is incrementally refined to use higher fidelity models (SW+/HW+). The higher fidelity simulation models generated from these models provide increasingly accurate performance metrics than in earlier phases,

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

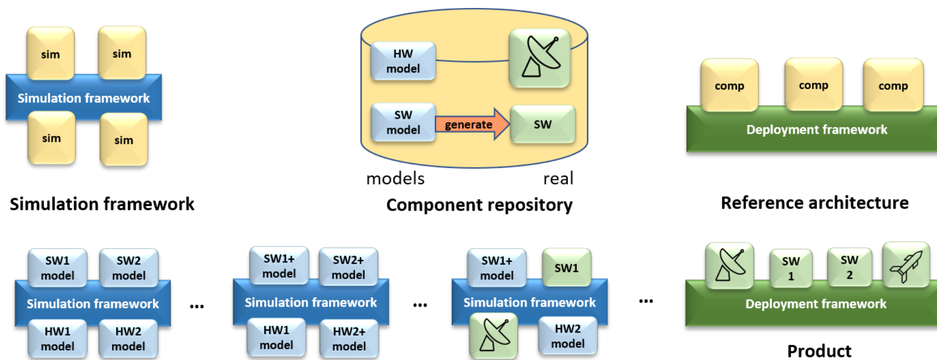

Figure 6: Vision of a single framework for all development phases.

although at the cost of longer development times of new models and increased simulation times. These simulations validate that the system that was defined truly delivers the required performance. As the actual software and hardware components are being developed, simulation models are gradually replaced using a software-in-the-loop and hardware-in-the-loop approach, respectively, allowing incremental verification of developed components before the entire system is developed. Once all components are developed, they are deployed using a deployment framework based on the reference architecture.

The main benefits of this model-based vision are that it provides a continuous and incremental development process with a high level of reuse between systems and development phases. This addresses the problems from Section 1 by: 1) reducing development time by increasing productivity through re-use, improved communication, lower defect rates, and better phase-containment of defects, and 2) improves evolvability by enabling changes to be made at model level and allowing artifacts of the system to be regenerated.

7 OPEN ISSUES

After presenting the results of our investigation, this section proceeds by defining a number of open issues that arose during this work. These issues may serve as suggestions for future research in this area.

1. **Ensuring semantic consistency of generators by construction.** Currently, we manually implement the semantics of the DSL in each generator and validate their consistency by comparing the outputs of the generated algorithms when testing in a common environment. While this worked reasonably well in our case, it means that each semantic change needs to be implemented in each generator. 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.
2. **Validation of implementations at different levels of abstraction.** In our case study, all implementations are at the same level of abstraction. This allowed us to expect the same output from all implementation given an identical environment, which served as a base for our validation approach. In the general case, simulation models in different frameworks and production code will have different abstraction levels, which will require a different validation method. A possible solution may be to specify tolerances for differences between results, if these are known.
3. **Techniques to develop a single simulation framework that can be used throughout the development chain.** While the vision in Figure 6 seems like a promising way forward, it is not clear how to technically realize it. Perhaps industry standard frameworks for co-simulation, such as High-level Architecture (HLA) (SISO, 2010), could be an important building block to ensure interoperability between models at different levels and implementations in hardware and software?

8 CONCLUSIONS

This paper discusses the first steps towards transferring an approach to Model-based engineering (MBE) and domain-specific languages (DSLs) to a company in the defense domain. The goal of this approach is to reduce design-time and improve evolvability by establishing continuity and reuse between different stages of design, such as early design space exploration, detailed performance estimation, and product implementation. The approach achieves this

goal by using domain-specific models as the sole truth to (re-)generate simulation models and product code, thereby enabling quick changes while ensuring mutual consistency. However, any new technology comes with both pains and gains and this work presents an investigation into the pains associated with this approach and how they can be mitigated.

A list of 14 technical pains related to MBE representative of our industrial partners is presented and six selected pains are further discussed in the context of industrial practice. A case study of a Threat Ranking component in a Combat Management System is then carried out to experience the selected pains and propose techniques to mitigate or eliminate them. The results of our investigation have convinced the company that the approach is feasible for simple components, which has allowed us to continue the work with a more complex case study. This will allow us to address more of the specified pains, as well as the three presented open issues.

REFERENCES

Baker, P., Loh, S., and Weil, F. (2005). Model-driven engineering in a large industrial context - Motorola case study. *Model Driven Engineering Languages and Systems*, pages 476–491.

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Beuche, D., Papajewski, H., and Schröder-Preikschat, W. (2004). Variability management with feature models. *Science of Computer Programming*, 53(3):333–352.

Broy, M., Kirstan, S., Krcmar, H., Schätz, B., and Zimmermann, J. (2012). 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*, pages 343–369.

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

Goulão, M., Amaral, V., and Mernik, M. (2016). Quality in model-driven engineering: a tertiary study. *Software Quality Journal*, 3(24):601–633.

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

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

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., and Völkel, S. (2014). Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378*.

Kurtev, I., Schuts, M., Hooman, J., and Swagerman, D.-J. (2017). Integrating interface modeling and analysis in an industrial setting. In *MODELSWARD*, pages 345–352.

Mellegård, N., Ferwerda, A., Lind, K., Heldal, R., and Chaudron, M. R. (2016). Impact of introducing domain-specific modelling in software maintenance: An industrial case study. *IEEE Transactions on Software Engineering*, 42(3):245–260.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344.

Mohagheghi, P. and Dehlen, V. (2008). Where is the proof? – a review of experiences from applying MDE in industry. *Lecture Notes in Computer Science*, 5095:432–443.

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

Mooij, A. J., Joy, M. M., Eggen, G., Janson, P., and Rădulescu, A. (2016). Industrial software rejuvenation using open-source parsers. In *International Conference on Theory and Practice of Model Transformations*, pages 157–172. Springer.

RTCA, Inc. (2012). *RTCA/DO-178C*. U.S. Dept. of Transportation, Federal Aviation Administration.

SISO, S. (2010). IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38.

Smith, P. F., Prabhu, S. M., and Friedman, J. (2007). Best practices for establishing a model-based design culture. Technical report, SAE Technical Paper.

Torchiano, M., Tomassetti, F., Ricca, F., Tiso, A., and Reggio, G. (2013). Relevance, benefits, and problems of software modelling and model driven techniques – a survey in the Italian industry. *Journal of Systems and Software*, 86(8):2110–2126.

Vetro, A., Bohm, W., and Torchiano, M. (2015). On the benefits and barriers when adopting software modelling and model driven techniques – an external, differentiated replication. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–4. IEEE.

Voelter, M. (2009). Best practices for DSLs and model-driven development. *Journal of Object Technology*, 8(6):79–102.

Voelter, M. (2010). Architecture as language. *IEEE Software*, 27(2):56–64.

Voelter, M. and Visser, E. (2011). Product line engineering using domain-specific languages. In *Software Product Line Conference (SPLC), 15th International*, pages 70–79. IEEE.

Whittle, J., Hutchinson, J., and Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85.

Wile, D. (2004). Lessons learned from real DSL experiments. *Sci. Comput. Program.*, 51(3):265–290.