2013 IEEE 37th Annual Computer Software and Applications Conference Workshops

Gaining Industrial Confidence for the Introduction of Domain-Specific Languages

Arjan J. Mooij
Embedded Systems Innovation by TNO (ESI)
Eindhoven, The Netherlands
Email: arjan.mooij@tno.nl

Jozef Hooman
ESI & Radboud University
Eindhoven & Nijmegen, The Netherlands
Email: jozef.hooman@tno.nl

Rob Albers
Philips Healthcare
Best, The Netherlands
Email: r.albers@philips.com

***Abstract*—Domain-Specific Languages (DSLs) receive attention as the possible next abstraction step in programming. Despite the benefits of using DSLs, in the industry there is also some reluctance against their introduction in product development. We address a number of issues that are important to gain industrial confidence for the introduction of DSLs. These include the available tools, the quality of generated code, and the incorporation in the industrial workflow. Our observations are based on an industrial study project at Philips Healthcare, especially concerning the development of a DSL for collision prevention. We also relate our experiences to the literature.**

I. Introduction

Domain-Specific Languages (DSLs) currently receive attention as the possible next abstraction step in programming that brings software development closer to the domain requirements. As described in [14], by trading generality for expressiveness in a limited domain, DSLs offer substantial gains in ease of use compared with general-purpose programming languages in their domain of application.

Surveys like [13], [5] illustrate that DSLs have been studied for some decades, but their use is not widespread in industry. Modern implementation technologies like EMF [20] seem to boost the applicability of DSLs in industrial practice, as witnessed by reports like [15], [22].

A lot of value of the DSL approach is obtained by code generation; see also the best practice "Use models to generate production code" from [18]. However, the development of code generators requires quite some effort. Since product development becomes dependent on the generated code, we need to show how this effort can be limited and how the quality of the generated code can be guaranteed.

In general, the introduction of DSLs in product development is not for free. For instance, [14] observes that "DSL development is hard, requiring both domain knowledge and language development expertise. Few people have both. Not surprisingly, the decision to develop a DSL is often postponed indefinitely." In this paper, our aim is to answer the question:

"How to get confidence for the introduction of DSLs in industry?"

According to our experience, confidence has to be obtained in several directions, leading to the following questions:

1) Which tools support the development and the use of DSLs? How practical are they in use?
2) How to get started with the introduction of DSLs? How to deal with the risks associated with new technology? How to limit the initial investment?
3) How to guarantee the quality of the generated code? How relevant is the structure and readability of the generated code?
4) How to embed DSLs in the existing industrial workflow? How to integrate the code generation with the existing software build process?

We address these questions based on our experiences with the industrial introduction of DSLs. Moreover, we relate our work to the literature and indicate where we follow or deviate from published guidelines, such as [24], [14], [10], [23].

In particular, we report about an industrial study project at Philips Healthcare aimed at redesigning some of the collision prevention components. Their medical imaging systems contain a number of moving parts, such as a table and various pieces of imaging equipment. Safety of these systems includes the avoidance of collisions between these heavy physical objects and humans, such as patient and medical staff.

The goal is to re-develop the collision prevention components in order to facilitate systematic reuse of safety-critical software across product configurations and medical applications. After analysing the proposed reference architecture in a similar way as described in [9], we have developed a DSL for collision prevention. This development includes the definition of a textual language, validation, code generation, and finally testing on physical systems.

Overview: In Section II we introduce the typical ingredients of a DSL and address the choice of tool support. We illustrate our choice using the study case at Philips Healthcare. Section III discusses our approach to the introduction of DSLs in industry. Questions concerning the quality of the generated code are discussed in Section IV. Section V explains our experiences with the support for debugging, logging, and monitoring, which are important for incorporation in the industrial workflow. Concluding remarks and a sketch of future work can be found in Section VI.

II. DSL Ingredients and Tool Support

When starting to develop a DSL in practice, one of the first questions is what tools to use. This concerns tools for both

978-0-7695-4987-3/13 $26.00 © 2013 IEEE
DOI 10.1109/COMPSACW.2013.83

the users and the developers of the DSL, as is also indicated by the guideline “Tooling Matters!” from [23].

There are two typical kinds of languages, namely, textual and graphical. The guideline “Adopt existing notations domain experts use” from [10] advocates to reuse notation from the domain itself. The guidelines “Decide carefully whether to use graphical or textual realization” from [10] and “Graphical vs. Textual Notation” from [23] compare these main types of languages. An important observation is that textual languages and their editors are faster to develop.

In the context of our collision prevention case, no existing notations were in use for specifying the collision prevention logic, apart from the implementation code. Our intended language users are familiar with textual programming languages. Moreover, they even prefer a textual language, as this simplifies the integration with their tools for comparing and merging files. Therefore we have focused on textual languages instead of graphical languages.

Based on industrial application reports like [15], [22], we have decided to focus on the Eclipse Modeling Framework (EMF, [20]). EMF is supported by a large collection of open source tools, including the tools EMFText [19], Xtext [4], Xtend [3], Epsilon [2] and Acceleo [1]. These tools can be used for creating editors, parsers, validators and code generators. Two main tooling approaches can be distinguished:

1) one integrated toolset (e.g., Xtext/Xtend);
2) a combination of several specialized tools.

In the following subsections we discuss some observations from our informal experiments with all these tools. It is not our goal to perform a systematic tool comparison. For our industrial study case we have decided to use Xtext/Xtend, but our results are not restricted to this toolset.

An overview of the typical ingredients of a DSL is depicted in Fig. 1. This overview addresses both the meta level, where the language infrastructure is defined, and the instance level, which is used by engineers when developing a particular application. The ingredients are explained in the following subsections.

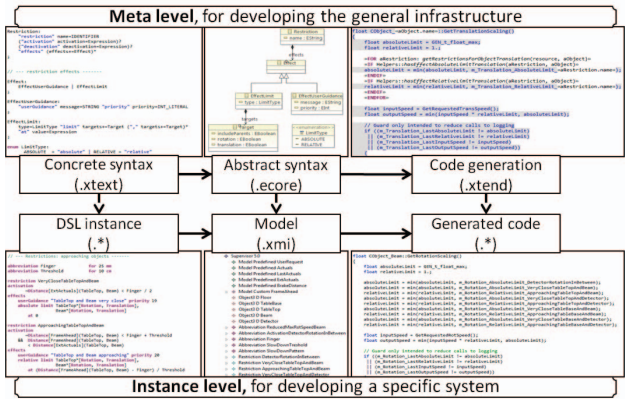

Figure 1. Overview of DSL ingredients

A. Abstract Syntax and Concrete Syntax

To create an editor and a parser for a textual language, at the meta level two artefacts are used:

- Abstract syntax (or meta-model): describing the domain concepts and their relations, typically in terms of a kind of UML class diagram. In our study case, the collision prevention component restricts the movements and informs the user, leading to concepts such as *Restriction*, *EffectLimit* and *EffectUserGuidance*; Fig. 2 depicts a small part of our abstract syntax.

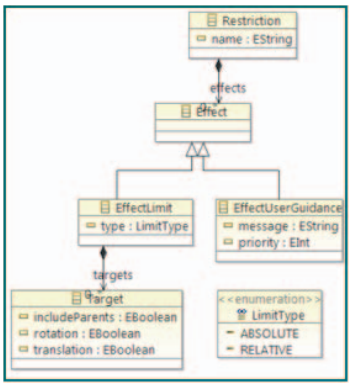

Figure 2. Abstract syntax

- Concrete syntax (or language): describing the representation of an instance of the abstract syntax, typically in terms of grammar rules. Fig. 3 shows a small part of the concrete syntax of our study case.

```
Abbreviation:
    "abbreviation" name=IDENTIFIER "for" expression=Expression
;

// --- restrictions -------

Restriction:
    "restriction" name=IDENTIFIER
    ("activation" activation=Expression)?
    ("deactivation" deactivation=Expression)?
    "effects" (effects+=Effect)*
;

// --- restriction effects -------

Effect:
```

Figure 3. Concrete syntax

These syntaxes are usually separated to manage the complexity [6], separating concept definition from their representation. Nevertheless, the structure of the abstract and concrete syntax overlap significantly [12] and they need to be defined consistently; see also guideline “Align abstract and concrete syntax” from [10]. As argued in [11], the abstract syntax is the central element which functions as pivot between concrete syntax and semantics.

EMF has built-in support for creating an abstract syntax, namely, a tree editor and a graphical editor (Ecore tools). For creating a concrete syntax, we have considered the tools Xtext and EMFText. These tools are based on ANTLR [16], which generates LL(*) parsers. Based on the syntaxes, these tools

generate a basic language environment, which includes Java classes for the concepts in the abstract syntax and an Eclipse-based editor using the concrete syntax.

The Extended Backus-Naur Form (EBNF) notation is the standard way of defining grammars, but both tools include some own extensions. In terms of [17], we consider the extensions from XText (such as assigned actions) more solution oriented, and the extensions from EMFText (such as operator precedence annotations) more problem oriented.

Petter Graff posted three blog [7] entries comparing Xtext and EMFText. He observes that their functionality is very similar, but there is more documentation for Xtext than for EMFText. He also observes some differences in the underlying language development process. Both tools can be used with a given abstract syntax, whereas Xtext can also derive an abstract syntax from a concrete syntax. As a consequence, the natural development process using EMFText starts with an abstract syntax, whereas using Xtext it can also start with a concrete syntax.

Given the very strict relation in these EMF-based tools between abstract and concrete textual syntax, we consider that a separate abstract syntax leads to more modelling efforts, more modelling formalisms (that need to be learnt), and more consistency issues. Hence, our choice is to use Xtext.

B. Instances of the DSL

Based on the definitions at the meta level, a language infrastructure is generated, including a dedicated editor for the defined DSL. It recognizes the domain specific keywords (as defined in the concrete syntax) and provides content assist functionality.

Fig. 4 contains an example instance of our collision prevention DSL. It uses custom language elements that correspond to domain concepts, including keywords such as "restriction" and "limit". Restrictions have an activation condition that depends on the distance between certain pairs of objects. Active restrictions lead to a certain effect, such as displaying user guidance messages and imposing speed limits on object movements.

```
// --- Restrictions: approaching objects -------

abbreviation Finger             for 25 mm
abbreviation Threshold          for 10 cm

restriction VeryCloseTableTopAndBeam
activation
      ~Distance[ExtActuals](TableTop, Beam) < Finger / 2
effects
    userGuidance "TableTop and Beam very close" priority 19
    absolute limit TableTop*[Rotation, Translation],
                   Beam*[Rotation, Translation]
       at 0

restriction ApproachingTableTopAndBeam
activation
      ~Distance[FrameAhead](TableTop, Beam) < Finger + Threshold
    &&  Distance[FrameAhead](TableTop, Beam)
       < Distance[ExtActuals](TableTop, Beam)
effects
    userGuidance "TableTop and Beam approaching" priority 20
    relative limit TableTop*[Rotation, Translation],
                   Beam*[Rotation, Translation]
       at (Distance[FrameAhead](TableTop, Beam) - Finger) / Threshold
```

Figure 4. Instance of our DSL

In addition, our language has some features that make it more convenient and less error-prone to use:

- *Shorthands*: notations such as "TableTop*" denote object TableTop and all objects it depends on. This is compact, but probably not self-explanatory; see also guideline "Balance compactness and comprehensibility" from [10].
- *Units*: programming languages usually focus on the types of numbers, but for specifications we use (optional) measurement units such as "mm" and "cm". This information was added based on earlier experiences with faults due to a misunderstanding of the units used.

C. Model Validation and Code Generation

At the instance level, code generation is used to generate source code from the model. Model validation is used for early fault detection, which saves development time by reporting problems (if any) before code generation. We consider two kinds of constraints for model validation:

- domain-specific constraints (e.g., certain relations must be acyclic);
- assumptions from the code generators (e.g., non-implemented features).

At the meta-level, model validation and code generation are defined using similar kinds of languages; in both cases the main goal is to get the right data from the model. Acceleo is a pragmatic implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL) standard; see [1]. Acceleo files are interpreted during code generation. The Xtend language is based on the Java language, and Xtend files are compiled into Java source code.

We have observed that after a small change in the DSL the Acceleo code generators always overwrite all generated files (also if there are no changes). For programming environments such as Visual Studio this leads to a time-consuming rebuild of all files. We have also observed that Xtend code generators only overwrite generated files that have changed, which avoids Visual Studio to rebuild them unnecessarily.

III. Introduction of DSLs in Industry

Since DSLs are new for the involved business unit of Philips Healthcare, we have to deal with the risks associated with the introduction of a new technology. This includes training the developers, but in [21] it is stressed that "only one or two developers need to master language and code generator development. The other developers simply use the modeling languages these experts have created. This division of labor enables the best developers to package their experience, empowering other members of the development team."

Still it is important to limit the initial investment and show the value of DSLs as soon as possible. To avoid a "big bang" approach, we have initially used the existing legacy code to implement some functionality by means of a DSL. In this way, it is possible to quickly demonstrate the generated code on a physical system. In our study case we could also quickly experiment with various collision prevention strategies expressed using the DSL.

Section III-A discusses the importance of the architecture and the requirements for developing a good DSL. In Section III-B we explain how we have incrementally identified the essential domain concepts. Section III-C discusses our choice to minimize the number of meta-models to reduce the initial investment.

A. Architecture and Requirements

Developing a DSL based on legacy code has the danger that the code generated from the DSL may need to mimic the existing code artefacts. In particular, there is a risk that the essential domain concepts may get lost in the implementation decisions of the legacy system. To obtain a good DSL, it is crucial that also the software architecture and the details of the requirements can be discussed.

For our study case, we have extensively discussed the possibility to separate various pieces of functionality. This is useful for reducing the complexity of the DSLs. Phrased differently, in the words of [21], "focusing only on a narrow area of interest ... The narrower the domain, the easier it becomes to build a good, high-level language and make generators produce first class code."

In our experience, the systematic approach imposed by DSLs helps to focus on general solutions (within the domain) instead of ad-hoc solutions. It helps to focus on identifying the essential domain concepts that can be reused systematically, instead of introducing all kinds of exceptions.

Given a new architecture with clear responsibilities for software layers and well-defined interfaces, a migration path can be defined where the code generated by the DSL gradually replaces larger parts of the legacy code. In this way, the risks are reduced and confidence in the DSL can be obtained gradually.

B. Incrementally Identify the Essential Concepts

When starting the development of a DSL, it is often difficult to decide which concepts should and should not be in the meta-model and the language. Initially the focus for the concepts is often on structure, and hence there is a risk that different developers have a different semantics of these concepts in mind. When inexperienced in developing DSLs, there is the additional problem of getting used to thinking at the meta level. In such situations, identifying a full meta-model before implementing any code generators is a big risk.

To develop the meta-model and the language, our approach is to quickly develop an incremental series of DSLs, including the code generators that give semantics to the concepts in the DSL. As updating the code generators requires a significant amount of work, we initially aim for pragmatic but not necessarily optimal implementations. During later development phases, once the language and meta-model are (more-or-less) fixed, optimizations can be performed by experienced programmers in the target programming language.

In our experience, this is an effective way to get fast feedback from domain experts. Demonstrating generated code on physical systems also helps to gain confidence in DSLs. This approach corresponds to the guideline "Iterate!" from [23], and the following advice from [21]: "The best approach for building DSLs is incremental. ... This iterative process minimizes risks, allows early testing, and smoothes the path of organizational change: moving from coding to modeling."

In general, we are reluctant towards introducing additional concepts; see also guideline "Reflect only the necessary domain concepts" from [10], and guideline "API should be minimal without imposing undue inconvenience" from [8]. We typically start with a very simple DSL that is close to the required external software interfaces, and offers minimal functionality. Then we proceed by determining what essential functionality is missing, and how this can be specified effectively. We continuously try to avoid introducing ad-hoc concepts, in order to keep the DSL simple, and the workload manageable.

In contrast, we are generous towards syntactic requests that increase the acceptance by users of the languages; see also guideline "Notation, notation, notation" from [23]. Examples of such syntactic requests include changes in keywords, or the order of attributes. Such requests can be implemented quickly, but we have noticed that we need to be careful with homonyms, which have different meanings to different people. In our collision prevention DSL, we have therefore avoided the term "movement", and we have introduced an explicit keyword "activation" to distinguish between "restriction" and "active restriction".

C. Small Number of Meta-Models

Model-based development, and in particular domain-specific languages, often consider multiple meta-models. However, as developing, maintaining, and documenting them also requires significant effort, we aim for a small number of meta-models, primarily the source meta-model. Below, we mention two reasons for the introduction of additional meta-models and explain why we did not use them.

Firstly, a meta-model could be introduced for the target language (like C or C++) itself. Instead, we expect that the complexity of a transformation step to the target meta-model is comparable to a generation step to the target language. Moreover, the generation step looks more intuitive for experienced developers in the target language. Generation also gives more control over the layout in the target language.

Secondly, extra meta-models could be used to split a generation (or transformation) step into several smaller steps. Instead, we control the complexity of the generation step using structured sets of auxiliary functions that on-the-fly perform transformations. The drawbacks include that the intermediate results cannot easily be inspected, and that it naturally leads to a tendency to recompute certain data from the model multiple times.

IV. Quality of Generated Code

The generated code becomes part of the final product and hence we must address its quality. To gain acceptance and confidence in both the generated code and the code generator, we

aim to generate human-readable code. This also provides an exit-strategy if DSL technology would be abandoned at some point in time, for example, because of future developments, or because of time pressure. Thus the development can continue manually based on the generated code.

The guideline "Care about generated code" from [23] also identifies the importance of adhering to the same standards as manually written code. On the other hand, sometimes we adapt the code structure to the DSL structure, in order to keep the code generator simple.

To manage the complexity of the code generators, we have defined a set of auxiliary functions to obtain information from the model efficiently. These functions can then be used in the code templates. This facilitates distribution of work, as developing the auxiliary functions is a different skill than developing code templates for the target language. It also stimulates that all code templates rely on the same definitions of the important domain-specific relations, which is good for their reliability. The guideline "Care about templates" from [23] also proposes the use of modularization techniques, and in particular to extract complex expressions in functions called by the templates.

A. Separating Constant and Variable Fragments

We distinguish two kinds of code fragments: constant and variable. Variable code fragments are affected by the DSL instance, whereas constant code fragments are not influenced by the DSL instance. When building a code generator, it is very tempting to mix constant and variable fragments.

This typically leads to a lot of code duplication, but (initially) this may not be considered as a major issue, given that the code is generated. It may even be considered beneficial for the statistics on the amount of generated code.

Nevertheless, this mixing has serious disadvantages:

- constant fragments are duplicated, which leads to unnecessarily long compilation times, and errors (if any) that are reported multiple times;
- constant fragments are likely to be developed using template editors, which are not as supportive as modern editors for traditional (plain) code.

For instance, an earlier version of our code generator resulted in a lot of duplication in the code that was generated for each restriction. In an improved version of the code generator we made this structure explicit using inheritance. That is, we use one general super-class for restrictions containing all generic/constant fragments, and let all specific restrictions inherit from this class.

In this way, the structure of the generated code is similar to typical manual code, where variable parameters are also separated from constant mechanisms. We therefore aim to separate constant and variable fragments, and aim to develop constant fragments outside the code generator. In particular we prefer to limit the amount of generated code by using external libraries with reusable functionality.

B. Testing the Code Generator

Domain-specific code generators can be seen as custom software tools. Since product development depends on these tools, the question is how to ensure the quality of the code generator. Note, however, that the code produced by the code generators can be treated in the same way as manually written code. Assuming that the generated code is tested thoroughly, there is no strong business need to test the code generator itself explicitly.

Code generators produce their code in a very structured way, and by using them more often and testing the generated code, the confidence in the code generators will grow. Moreover, the use of code templates makes it easy to ensure consistency of the pieces of code, and to adhere to coding standards. Furthermore, code generators focus on a structured translation from domain requirements to implementation concepts. Once implemented, this translation can be reused systematically in a more reliable way than manual coding.

V. Debugging, Logging, and Monitoring

Before introducing code generators in product development, we need to determine how to support debugging, logging, and monitoring. As the code is generated using code templates, it is easy to weave in some extra code for logging and monitoring. The main challenge is to relate the generated code back to the abstraction level of the DSL instance.

We distinguish between debugging the code generator and debugging a language instance, although in both cases the generated code plays a role. When debugging the code generator, we can assume familiarity with the details of the DSL and the structure of the generated code. To support the debugger, it is important that the generated code and the code generator are human-readable and well-structured. Then we can rely on traditional means of debugging the generated code and the code generator.

When debugging a language instance, we cannot assume that the developer is (or wants to be) aware of the details of the design and the execution architecture of the generated code. In these cases we need mappings between the DSL instance and the generated code. Regarding debugging, [25] distinguishes two aspects:

- debugging actions, e.g., breakpoints, step, suspend, and resume;
- debugging views, e.g., on variables and control state.

Typically a single DSL construct is translated into several constructs in a target language. Hence debug actions in terms of a DSL need to be translated to the target language, and the values of variables need to be translated from the target language to the DSL.One way is to integrate the DSL environment with a traditional development environment, e.g., MetaEdit+ with Visual Studio. Open issue "Model Debugging" from [23] expresses a preference for such a general technical solution instead of a hand-constructed specific solution.

For DSLs that are operational and state-based, it may be useful to map small steps in the DSL to big steps in the generated code. For declarative DSLs like our collision prevention

DSL this is not so natural. Our collision prevention DSL is a declarative language with two natural big computation steps, related to:

- a user command with a new movement request;
- a periodic evaluation of all requests and restrictions.

The places in the code that initiate the DSL behaviour for these big computation steps, typically a single method call per big computation step, are logical places for breakpoints and for performing "step over" actions.

For debugging views that inspect the internal state of the generated code, we have used various monitor screens at the abstraction level of the DSL. These include the values of the used input sensors, the evaluation of restrictions, and the produced output to the motors. In addition, part of this information and the incoming and outgoing method calls are stored in logs. This is fully integrated with the other logging and monitoring facilities in the product software. As an extra benefit over a general technical solution, this product-integrated solution can even be used when the system is running outside the development environment.

VI. Concluding Remarks

To obtain industrial confidence for the introduction of DSLs, we have developed a prototype DSL and made a number of decisions, which we have related to the literature. Thus we address our initial questions from Section I as follows:

1) We have developed a textual DSL using the EMF-based toolset Xtext/Xtend, for which only a small number of artefacts need to be created at the meta level.
2) To effectively implement a software component using a DSL, we have focused on clearly defined pieces of functionality, where some variability is to be expected. To limit the initial investment, we have minimized the number of meta-models and incrementally identified the essential concepts. Already in early stages we have developed the first code generators, in order to reduce risks, to obtain executable results and to get early feedback.
3) To get confidence in both the code generator and the generated code, we have generated human-readable code without a lot of code duplication. This also avoids the danger of a technology lock-in as product development can always continue manually based on the generated code alone. To structure the code generator, we have separated the functions that inspect the model from the code generation templates.
4) To integrate DSLs with the existing industrial workflow, a strategy is needed for debugging, logging, and monitoring. On the tooling side, unnecessary rebuilds of unchanged files should be avoided, and code generation must be possible from the command line for inclusion in the software build process.

The DSL approach separates domain models for specific cases from code generators for the general case. This separation facilitates a change to another target language, to another coding standard, or to another logging and tracing strategy. In addition to generating code, also analysis models can be generated; see also best practice "Rule of Two" from [18]. As a part of future work, we aim to generate analysis models for more advanced validation, including safety and performance analysis.

Acknowledgment

This publication was supported by the Allegio project, as part of the Dutch national program COMMIT. The authors like to thank Hans Driessen for his valuable comments during the execution of this work.

References

[1] Acceleo. http://www.eclipse.org/acceleo/ (2012), version 3.1.3
[2] Epsilon. http://www.eclipse.org/epsilon/ (2012), version 0.9.1
[3] Xtend. http://www.eclipse.org/xtend/ (2012), version 2.3
[4] Xtext. http://www.eclipse.org/Xtext/ (2012), version 2.3
[5] van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: an annotated bibliography. SIGPLAN Notices 35(6), 26–36 (2000)
[6] Fondement, F., Baar, T.: Making metamodels aware of concrete syntax. In: Proceedings of ECMDA-FA'05. LNCS, vol. 3748, pp. 190–204 (2005)
[7] Graff, P.: Xtext or EMFText... that is the question. http://pettergraff.blogspot.nl/2009/11/xtext-or-emftext-that-is-question.html (Nov 2009)
[8] Henning, M.: API design matters. ACM Queue 5(4), 24–36 (2007)
[9] Hooman, J., Mooij, A.J., van Wezep, H.: Early fault detection in industry using models at various abstraction levels. In: Proceedings of IFM'12. LNCS, vol. 7321, pp. 262–282 (2012)
[10] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schneider, M., Völkel, S.: Design guidelines for Domain Specific Languages. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09). pp. 7–13 (2009)
[11] Kleppe, A.: A language description is more than a metamodel. In: Proceedings of SLE'07 (2007)
[12] Krahn, H., Rumpe, B., Völkel, S.: Integrated definition of abstract and concrete syntax for textual languages. In: Proceedings of MoDELS'07. LNCS, vol. 4735, pp. 286–300 (2007)
[13] Krueger, C.W.: Software reuse. ACM Computing Surveys 24(2), 131–183 (1992)
[14] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
[15] Nagy, I., Cleophas, L.G., van den Brand, M., Engelen, L., Raulea, L., Mithun, E.X.L.: VPDSL: A DSL for software in the loop simulations covering material flow. In: Proceedings of ICECCS'12. pp. 318–327. IEEE (2012)
[16] Parr, T.J., Quong, R.W.: ANTLR: A predicated-LL(k) parser generator. Software: Practice and Experience 25(7), 789–810 (1995)
[17] Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
[18] Smith, P., Prabhu, S., Friedman, J.: Best practices for establishing a model-based design culture. SAE Technical Paper 2007-01-0777 (2007)
[19] Software Technology Group, TU Dresden: EMFText. http://www.emftext.org/ (2011), version 1.4.0
[20] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. Pearson Education (2008)
[21] Tolvanen, J.P.: Domain-specific modeling for full code generation. Software Tech News (STN) 12(4), 4–7 (2010)
[22] Verriet, J., Liang, H.L., Hamberg, R., Wijngaarden, B.: Model-driven development of logistic systems using domain-specific tooling. In: Complex Systems Design & Management. pp. 165–176. Springer (2013)
[23] Voelter, M.: Best practices for DSLs and model-driven development. Journal of Object Technology 8(6), 79–102 (2009)
[24] Wile, D.S.: Lessons learned from real DSL experiments. Science of Computer Programming 51(3), 265–290 (2004)
[25] Wu, H., Gray, J., Mernik, M.: Grammar-driven generation of domain-specific language debuggers. Software: Practice and Experience 38(10), 1073–1103 (2008)