Using DSLs for Developing Enterprise Systems

Margus Freudenthal
Cybernetica AS/University of Tartu
Aleksandri 8a
Tartu 51004, Estonia
margus@cyber.ee

ABSTRACT

This paper investigates the suitability of contemporary DSL tools in the context of enterprise software development. The main focus is on integration issues between the DSL tool, the DSL implementation and the rest of the enterprise system. The paper examines different scenarios for integrating DSLs into the enterprise systems. A number of criteria for evaluating DSL tools are then extracted from these scenarios. These criteria are then used to evaluate five industry-strength DSL tools.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.2.6 [**Software Engineering**]: Programming Environments—*programming workbench*

General Terms

Languages, Design

Keywords

Domain-Specific Languages

1. INTRODUCTION

Background and Motivation

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [16]. Classical examples of DSLs are Unix Makefiles (build scripts), regular expressions (specifying text patterns), HTML (describing text layout), and GraphViz (describing graphs). This paper investigates practical issues connected to using DSLs for developing enterprise information systems.

Before we can get into specifics, we need to define what is an "enterprise information system" (EIS). Finding the authoritative definition for this term is quite difficult (if not impossible), but there seems to be a rough consensus that the EIS is a system for integrating and coordinating business processes of a (usually large) organisation. From the technical point of view, the enterprise systems are characterised by the following properties.

- They are usually implemented using programming languages such as Java, C#, a 4GL, or a database language (PL/SQL). The enterprise systems also make extensive use of frameworks (such as JEE and .NET) and middleware (application servers, enterprise service buses etc.).
- They may consist of a set of interconnected modules that are built upon a common architecture or on top of a packaged enterprise system such as an Enterprise Resource Planning (ERP) system. Often there may be a benefit in applying the Software Product Lines [11] approach.
- They tend to be shallow but wide. The application code in an EIS itself does not have to be technically complex because the complex parts are implemented in the frameworks and application servers. Instead, most of the application code deals with implementing the concepts, rules and processes of the organisation.

The frameworks and middleware that are used to build the EIS usually contain different graphical or XML-based DSLs for configuring the components. These DSLs are often called horizontal or technical DSLs. Additionally, there are vertical or business DSLs that are concentrated on encoding business logic for a specific business domain such as taxation, banking, or medical domains. Taking advantage of the vertical DSLs is usually technically more complex because there may not exist a suitable DSL (with suitable implementation) for this particular application, and the EIS developer herself must create the necessary DSLs. This paper focuses on the second, more active type of the DSL use where the DSLs and DSL-supporting components are developed together with the EIS.

If one adopts the DSL-based development process, the costs of developing a new DSL must be low otherwise it is more economical to code the business logic in a general-purpose programming language. The DSL development can be made more efficient by taking advantage of good tool support. There exists a reasonable body of tools to assist in various aspects of creating DSLs. Examples of these tools are

parser generators, code generators, transformation systems, and IDE generators. The main question posed in this paper is: are the existing DSL tools suitable for use in enterprise software development?

Scope of this Work

Since the terms DSL and DSL tool can be used in quite a wide sense, we will first make matters more concrete by narrowing down the range of DSLs and DSL tools analysed in this paper.

DSLs can take quite different forms and be implemented in quite different ways (see [15] for a taxonomy). This paper focuses on the development model where the enterprise system is built following the language-oriented programming method, described by Ward in [17]. With this method, a component is divided into two parts. The first part contains the functionality of the component, described in a DSL. The second part contains the implementation of the DSL. This approach can be applied recursively – the implementation of the DSL can be written using another, lower level DSL. This model of development assumes that the DSLs are able to communicate with each other and that the DSL implementations are integrated into the system to allow fine-grained control over which parts of the system are implemented using DSLs.

Language-oriented programming is a general method that can be realised by means of DSLs embedded in a general-purpose programming language (internal DSL). In this way, the various DSLs can be integrated by means of the host programming language. However, mainstream general-purpose programming languages commonly used for developing enterprise systems do not offer features needed for creating high-quality internal DSLs (e.g. unobtrusive syntax, ability to define new control structures). In addition, internal DSLs are often difficult to use for non-programmers because the details of the host language often interfere with the DSL (for example, in the case of error messages). Therefore, this paper focuses on external DSLs that are implemented using especially crafted parser and that can offer syntax and semantics that does not depend on the host language. When building an external DSL, one can use either textual or graphical syntax. Both have different strengths and weaknesses and choosing between them is mostly a matter of taste and availability of tools, competence, etc. Existing tools for developing textual DSLs have different characteristics compared to those for developing visual DSLs. It would be beyond the scope of one paper to analyse both categories of toolsets. Accordingly, in this paper we focus on textual DSLs.

DSL tools are considered to be software programs that simplify some aspects of creating DSLs (such as parser generators, code generators, and IDE generators) and that are advertised as DSL tools. Many software tools (including all the general-purpose programming languages) can, in principle, be used to implement DSLs. Parser libraries in high-level languages, such as Haskell or Prolog, can achieve the result comparable to using specialised parser generators [8]. In order to compare the tools on a more equal footing, this paper only looks at tools that are specifically aimed at developing DSLs.

Previous work

Published comparisons of DSL/DSM tools [14, 12, 10] have mostly discussed available functionality and ease of use. In particular, the set of tools included in comparison by Pfeiffer and Pichler [14] is similar to this paper. However, this paper differs from the Pfeiffer and Pichler paper in two important aspects. First, this paper only reviews tools that are suitable for industrial-strength software development. This excludes pure research prototypes and inactive open source projects. Additionally, this comparison includes tools that do not provide IDE functionality but can nevertheless be used to create a working DSL implementation. Second, Pfeiffer and Pichler are mainly concerned by general characteristics of the tools (type of metamodel, type of code generator, etc.), whereas this paper targets the technical properties that are important when using the DSL tools for enterprise development.

Den Haan [3] describes the development model and the roles that are involved in DSL-based software development. This model is reused in this paper as the basis for terminology related to roles in the DSL-based software development.

2. INTEGRATING DSLS INTO EIS

In general, all the functionality of an enterprise information system cannot be expressed by a single DSL program. The EIS contains several concerns (persistence, distribution, business logic) that each are best handled using a separate DSL. For example, the Java Enterprise platform includes several XML-based technical DSLs. Additionally, different parts of business logic (workflows, state machines, verification rules) are often best expressed using different DSLs. Following that logic, a DSL-based EIS would typically consist of components written in different DSLs and glued together by code manually written in a general-purpose language, such as Java. Therefore, one important question is how the DSL code can be called from the rest of the EIS. This section lists different scenarios for integrating the DSL implementation into the EIS.

The biggest factor influencing the options for integrating DSL code and "glue" code is whether the DSL is interpreted or compiled. In this paper, the line between interpretation and compilation is drawn according to whether the DSL interpreter is part of the application code or part of the environment (language runtime, hardware)[1].

The integration options listed in this section mostly differ in the point of time when the DSL program is packaged/compiled and loaded into the system. From the DSL user's point of view the main difference between the options is the versioning model of the DSL program. For example, if the DSL program is packaged with the rest of the source code of the system then the DSL program should be versioned together with the rest of the source code.

[1]For example, an application can be written in the Python language and the DSL program translated to Python (or Python bytecode), which is then loaded into Python runtime. We consider this approach to be compilation, because the code is interpreted by the environment (Python runtime). However, if the DSL program is translated to a form that is interpreted by the Python code in the application itself, then we consider it to be interpretation.

There are two principal ways of deploying DSL programs that use interpretation.

I.1. DSL program can be packaged with the application source code as a text resource. For example, this option is used in the Java Enterprise platform for various configuration and manifest files. With this approach, the life-cycle (deployment schedule, versioning policy) of the DSL program will match the life-cycle of the other application code. Changing the DSL program involves redeploying of the whole application. Therefore, this approach is mainly suitable for technical DSL programs that change with the application code.

I.2. DSL program can be loaded at runtime and stored in a file or a database. The program can either be in the source form, or it can be compiled to some kind of bytecode. The application can also provide environment for editing, testing and debugging the DSL programs. Because changing the DSL program does not involve changing or redeploying the rest of the application code, this approach is suitable for non-technical DSL programs that capture fast-changing business requirements.

For deploying compiled DSLs, there are three options for packaging and loading the DSL programs (see figure 1).

C.1. The DSL program can be compiled during the build process of the system. The compiled DSL program is then packaged and deployed together with the rest of the system. Processes for changing and deploying the DSL program are exactly the same as for changing and deploying rest of the application code. The DSL program thus becomes integral component of the system that is integrated with other, possibly hand-written components. This approach is suitable for technical DSLs (essentially programming at higher level of abstraction).

C.2. The DSL program can be compiled separately and loaded into the system at run time (e.g. as a dynamically loaded plugin). The DSL program can be managed separately from the application code and can be used for describing business logic that changes more often than the application code. This option requires creating a special tool or a build system that is able to compile the DSL program and package it for deployment.

C.3. The DSL program can be loaded into the running application and compiled by the application code. The compiling can be done in several steps, for example by first generating Java source code and then invoking the Java compiler. The compiled DSL program is stored in a file or a database and loaded at run time. Using this option requires packaging the DSL compiler with the application software. As with the previous option, the life-cycle of the DSL program is not tied to the life-cycle of the application code.

The main difference between the last two options is the environment where the DSL compiler is run. The option C.3 can be somewhat more complicated to implement because the runtime environment must also contain all the development tools and libraries needed for compiling the DSL programs. Also, invoking the DSL compiler from the application code can be complicated and resource-consuming. However, embedding the DSL compiler into the application software can offer several benefits. First, it lowers the requirements for the working environment of the business engineer[2]. If the application software includes a web-based DSL editing tool, the business engineer can create DSL programs using only a web browser. Second, if the DSL program comes from untrusted sources (e.g., end users customising their user experience) then it is possible to analyse the DSL program before compiling to ensure it meets the requirements. If the DSL program is compiled before loading into the system, then the analysis becomes more difficult as it becomes necessary to reverse engineer the compiled code.

3. EVALUATION CRITERIA

This section presents the criteria that will be used in the next section for evaluating the DSL tools. The main question is, whether the DSL tools support the integration scenarios described in the previous section. Each criterion is given a short code that will be used to refer to it in later text.

[sep_wb] If the DSL tool contains an IDE builder, then it is possible to use the generated DSL IDE separately from the DSL tool. The user of the DSL IDE cannot change the language definition. Fulfilling this requirement makes it possible to separate the work of of the language engineer/transformation specialist[3] and the business engineer.

[build] It is possible to integrate the DSL compiler into the build process of the system. In particular, this means that the DSL compiler must be able to operate in non-interactive environments, e.g. when called by a build script or when used in an automated build server. Fulfilling this requirement makes it possible to implement scenario C.1.

[system] It is possible to integrate the DSL compiler into the system. In particular, this means that the DSL compiler can be deployed as a library that can directly called from the application code. This criterion is stricter than the criterion [build]. In principle, if the tool can be invoked from the build system, then it is possible to invoke it from the system as an external program. However, the criterion [system] means that the DSL compiler can be invoked without the overhead of creating another process. Fulfilling this requirement makes it possible to implement scenario C.2.

[vcs] The DSL tool supports version control of the DSL programs. If the DSL uses human-readable

[2]Business engineer is the person who describes solutions to business problems with DSL program. See [3] for a full taxonomy of the roles in DSL-oriented development.

[3]The language engineer creates the language description and the transformation specialist creates the language implementation.

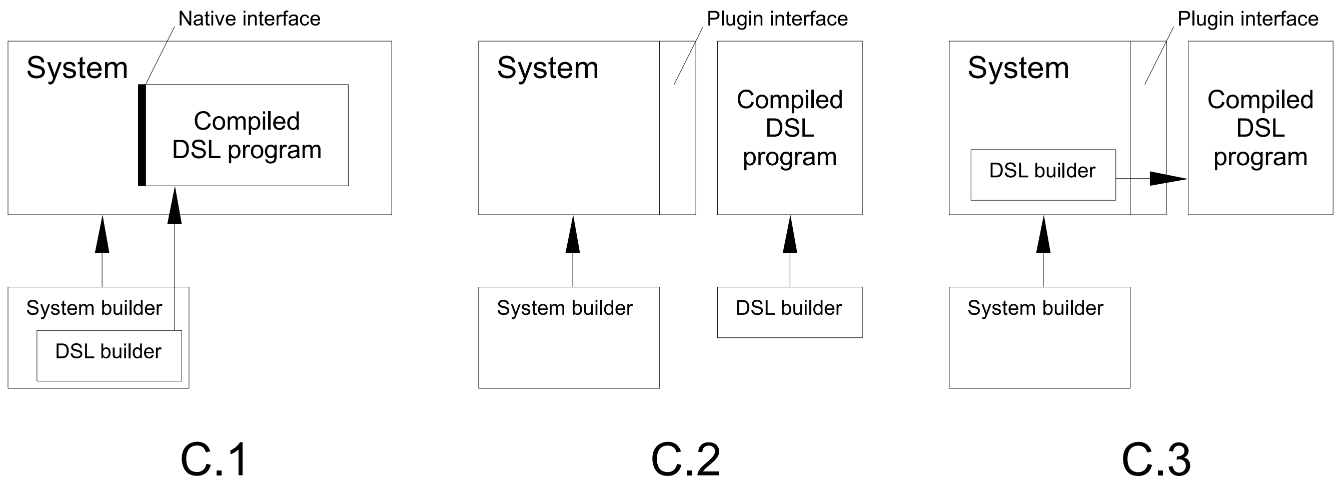

Figure 1: Options for deploying compiled DLSs

textual syntax, this is very easy because storing and merging text files is supported by all the popular version control systems. However, if the DSL program is stored in some internal format (e.g. serialised XML representation), then the DSL tool must include explicit support for version control. Fulfilling this requirement makes it possible to implement scenarios I.1 and C.1.

[custom] The DSL tool supports creating customisable DSL implementations that contain basic functionality and offer extension points where application-specific customisations can be applied. Examples of these customisations are set of objects that can be manipulated by the language and operations that can be applied to the objects. Customisation can also be achieved if it is possible to divide the language implementation into smaller modules or fragments and then reuse these fragments in different combinations. Creating customisable DSLs and composing a DSL from several language modules may be needed if the EIS is built as a software product line.

4. EVALUATION RESULTS

This section evaluates five DSL tools with respect to the criteria defined in section 3.

Because this paper focuses on tools for industrial software development, we restrict our evaluation to tools that are reasonably mature, are supported by a community or a company and are available to the public. These criteria are elaborated below.

- Maturity. The tool should have a history of practical use for several years. It should have a stable current release that can be used for developing production-level code. There should be cases of using the tool in several production systems (this requirement excludes pure research tools that have been developed and used only once or twice). The tool should be documented reasonably well and work without crashing or malfunctioning.
- Support. The tool should currently be in active maintenance and new releases (or bug fixes) should appear periodically, in reasonable time intervals. There should be an active user community that supports the tool via mailing lists or user forums. Alternatively, there should be a company providing support for the users of the tool.
- Availability. The tool should be available to general public under a reasonable licence. Open source licences are preferable because they lower the risks caused by e.g. insufficient documentation or community support.

Furthermore, to ensure that the selected DSL tools are representative, we classified existing DSL tools into three categories and selected one or two tools in a given category. The categories and selected tools are described below.

Standalone Tools

This group contains tools that do not depend on any IDE. In general, they contain a parser generator and some kind of code generation facility (e.g. a template engine). Because they do not depend on an IDE, the [sep_wb] criterion is not applicable for them. As standalone tools, they can easily be called from the build system (criterion [build]). Because DSL programs are stored as plain text files, they can be version-controlled using any VCS, thus satisfying criterion [vcs].

ANTLR and StringTemplate

ANTLR[4] [13] is primarily a parser generator. It takes as input a description of a DSL's syntax and produces a parser that recognises the DSL. By adding actions to the parser, it is possible to build a translator or an interpreter. ANTLR offers facilities for parsing the DSL program into its abstract

[4]Available online at `http://www.antlr.org/`

syntax tree and then performing operations on the tree (using tree grammars to navigate the tree). StringTemplate[5] is a templating engine that works well with ANTLR and can be used for generating code from the DSL program. Together with ANTLR, it is possible to create a complete implementation (parser and code generator) for a simple DSL.

ANTLR generates code and provides APIs for several popular programming languages. Therefore, in most cases it is possible to integrate the ANTLR-based compiler or interpreter directly with the target system, thus satisfying the [system] criterion. ANTLR's support for customisable languages (criterion [custom]) is quite limited. There is a mechanism for grammar inheritance, but this only supports some specific cases of reuse and parametrisation.

Stratego/XT

Stratego/XT[6] [1] is a language and a tool set for program transformation. The toolkit provides facilities for defining the concrete syntax of the DSL (parser generator) and applying transformations to the abstract representation of the DSL program. The transformations are expressed in terms of rewriting rules that can be controlled by programmable strategies. The Stratego/XT toolkit can be used for implementing interpreters, compilers or program transformation tools (type checker, model checker, etc.). In addition, there is an IDE creation toolkit Spoofax/IMP[7] [9] (based on the IMP toolkit discussed below) that can generate Eclipse plugins for Stratego-based DSLs.

The Stratego/XT toolkit is written in C and generates native code. Integration into the EIS therefore means invoking an external program or using a foreign function interface. Integration as an external program is acceptable for invoking a compiler, but can add considerable overhead when invoking a frequently-called interpreter. Stratego has Java-based runtime that can run Stratego programs in the Java virtual machine, although with severely reduced performance. Therefore, the Stratego does not fully support the [system] criterion. On the other hand, Stratego allows creating highly modular and customisable DSL implementations and therefore satisfies the [custom] criterion.

Eclipse-Based Tools

This group contains tools that include Eclipse-based IDE builders. In general, the DSL implementation is deployed as an Eclipse plugin, which can be distributed separately from the DSL tool. This satisfies the [sep_wb] criterion. Both tools in this group store DSL programs as plain text files, therefore fulfilling the [vcs] criterion.

Xtext

Xtext[8] [7] is an Eclipse-based framework for creating textual DSLs. It integrates with the Eclipse Modelling Framework (EMF) and uses the Ecore metamodel for describing the abstract representation of the DSL program. Based on the grammar description Xtext creates the parser, the metamodel and the Eclipse-based IDE. The generated IDE can be customised (e.g. by adding checks or specifying non-default rules for outlining a DSL program) by writing Java code. Xtext includes templating engine Xpand that can be used for simple code generation tasks.

Xtext is quite tightly integrated with the Eclipse and EMF in particular. It is possible to build an Xtext-based DSL from a command line and satisfy the [build] criterion. However, the amount of dependencies to various Eclipse components can make it difficult to integrate an Xtext-based language implementation into an EIS, although the parser and the code generator can be called from the Java. Therefore, Xtext fulfils the [system] criterion partially. Xtext provides support for modular grammars and the language services can also be modular, thus satisfying the [custom] criterion.

IMP

IMP[9] [2] is primarily an Eclipse-based IDE builder. It comes with the LPG parser generator, but can also use other parsers. When compared to Xtext, the IMP offers more flexibility in creating a DSL implementation. The cost of this flexibility is increased complexity: compared to the Xtext, the IMP requires more programming to create an IDE for a simple DSL.

The IMP does not offer much support for creating non-visual parts of a DSL implementation and therefore does impose significant restrictions. The LPG parser generator can generate Java code that can be directly called by the system. Thus, IMP satisfies both the [build] and the [system] criteria. The IDE generated by the IMP can be made quite customisable, but the LPG parser generator does not support creating modular grammars. Therefore, the IMP fulfils the [custom] criterion partially.

Language Workbenches

Language workbenches [5] represent a different approach to DSL development than the previous groups. This approach uses a mix of techniques from the textual and the graphical DSLs. The DSL program is displayed on screen as text, but the user directly edits the abstract representation of the program. This technique is called projectional editing [6] and has been typically used for editing graphical DSLs.

Meta Programming System

The Meta Programming System[10] (MPS) [4] is not as mature as the other tools reviewed in this paper, but it serves as a good example of the language workbench approach. MPS offers tools to cover all the aspects of creating a DSL, including explicit support for defining constraints and type systems.

MPS does not allow deploying the DSL IDE separately from the main workbench. However, it is possible to export the DSL as a read-only package to prevent its accidental modification by the business engineer. Because the deployed language is not well encapsulated, MPS does not fully satisfy the [sep_wb] criterion.

[5] Available online at `http://www.stringtemplate.org/`
[6] Available online at `http://strategoxt.org/`
[7] Available on-line at `http://strategoxt.org/Stratego/Spoofax-IMP`
[8] Available online at `http://www.eclipse.org/Xtext/`
[9] Available online at `http://www.eclipse.org/imp/`
[10] Available online at `http://www.jetbrains.com/mps/`

Feature	Stratego	ANTLR	Xtext	IMP	MPS
sep_wb	N/A	N/A	Yes	Yes	Partial
build	Yes	Yes	Yes	Yes	No
system	Partial	Yes	Partial	Yes	No
vcs	N/A	N/A	N/A	N/A	Yes
custom	Yes	Partial	Yes	Partial	Yes

Table 1: Tool comparison matrix

A DSL implementation (even the type checker and code generator) created with MPS is tightly integrated with the MPS's graphical workbench and the underlying IntelliJ IDEA. According to a comment from an MPS developer[11], this is a limitation of the current architecture of MPS and there are no plans to change it. Because of this, a MPS-based DSL compiler cannot be invoked from the command line or from the system, and MPS does not satisfy the criteria [build] and [system].

MPS provides excellent support for creating modular and extensible DSLs. It is possible to extend a DSL with new constructs or to combine several DSLs. The MPS therefore satisfies the criterion [custom]. Unlike the previous tools, the MPS stores DSL programs as structured XML files. It therefore needs explicit support for version control operations. MPS inherits VCS support from the IntelliJ IDEA, which supports all the major VCS systems, thus satisfying [vcs].

5. CONCLUSIONS

Table 1 summarises the evaluation results. When looking at functionality, the Xtext and the MPS are the two tools that cover all the aspects of creating a DSL: parsing, code generation, validation, and IDE generation. However, with these tools it is difficult to implement the integration scenarios listed in section 2. The non-visual parts of these tools (parser, code generator, program validator) depend on the IDE framework. On the one hand, this makes the IDE part simpler and more powerful because the DSL program is expressed in a format most suitable for implementing the IDE services. On the other hand, this architecture makes it difficult to use tool in combination with other tools or outside the original IDE environment.

The other evaluated tools have a more specific focus. ANTLR and Stratego focus on parsing and code generation; IMP is mainly an IDE generator. Although Stratego can be used with the Spoofax/IMP IDE generator and IMP comes bundled with the LPG parser generator, the interface between the visual and non-visual parts of the DSL implementation is clearly defined and the non-visual part does not have dependencies that would prevent it from functioning as a component of a larger system.

To sum up the evaluation results, the users of DSL tools seem to have a choice between a conveniently packaged end-to-end solution and the ability to integrate parts of the DSL toolchain into an EIS. This is unfortunate because technically there are no strong reasons why the two cannot be combined. The author wishes to make a point that a DSL tool is often not an end in itself but rather a part in a larger system. Thus, the developers of DSL tools should consider making the tools modular so that they can easily be combined with other tools and embedded into larger systems.

The main contributions of this paper are: (i) the discussion of scenarios of how a DSL can be integrated into an EIS; (ii) list of criteria for evaluating whether a DSL tool can support the integration scenarios; and (iii) evaluation of representatives of different types of DSL tools against these criteria. The paper provides a perspective about what is required in order to successfully use DSLs in developing enterprise information systems. This is useful to enterprise developers for selecting a suitable tool; and to DSL tool developers for making their tools usable in the context of EIS.

This work has looked at a subset of all available DSL tools. First, we have considered only tools that are specifically targeted for DSL creation, as opposed to general-purpose programming languages that can also be used to create DSL implementations. Second, we have looked at external DSLs that use textual syntax. Third, there was a preference for open source tools that have substantial amount of freely available documentation. One of the goals of our future research is to lift these restrictions and verify the conclusion of this paper against more classes of DSL tools, including visual DSL tools.

[11] `http://www.jetbrains.net/devnet/thread/283315`

Acknowledgements

The author wishes to thank Marlon Dumas and the anonymous reviewers for their feedback on earlier versions of this paper. This research was supported by the Estonian Doctoral School in Information and Communication Technology.

6. REFERENCES

[1] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008. Special issue on Experimental Systems and Tools.

[2] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton, Jr. Imp: a meta-tooling platform for creating language-specific ides in eclipse. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 485–488, New York, NY, USA, 2007. ACM.

[3] Johan den Haan. Roles in Model Driven Engineering. `http://www.theenterprisearchitect.eu/archive/2009/02/04/roles-in-model-driven-engineering`, 4 February 2009.

[4] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`, November 2004.

[5] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? `http://www.martinfowler.com/articles/languageWorkbench.html`, May 2005.

[6] Martin Fowler. Projectional Editing. `http://www.martinfowler.com/bliki/ProjectionalEditing.html`, January 2008.

[7] Peter Friese, Sven Efftinge, and Jan Köhnlein. Build your own textual DSL with Tools from the Eclipse Modeling Project. `http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html`, 18 April 2008.

[8] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functctional Programming*, 8(4):437–444, 1998.

[9] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Domain-specific languages for composable editor plugins. In T. Ekman and J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, April 2009.

[10] Benoît Langlois, Consuela elena Jitia, and Eric Jouenne. DSL Classification. In *The 7th OOPSLA Workshop on Domain-Specific Modeling*, October 2008.

[11] Linda M. Northrop and Paul C. Clements. A Framework for Software Product Line Practice, Version 5.0. Technical report, Software Engineering Institute, July 2007.

[12] Turhan Özgür. Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling in the context of Model-Driven Development. Master's thesis, Blekinge Institute of Technology, 22 January 2007.

[13] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[14] Michael Pfeiffer and Josef Pichler. A Comparison of Tool Support for Textual Domain-Specific Languages. In *The 8th OOPSLA Workshop on Domain-Specific Modeling*, 19 October 2008.

[15] T. Sloane, M. Mernik, and J. Heering. When and how to develop domain-specific languages. Technical Report SEN-E0309, CWI, 2003.

[16] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[17] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.