Concepts for Model-Driven Design and Evolution of Domain-Specific Languages

Uwe Zdun
Department of Information Systems
Vienna University of Economics
Vienna, Austria
zdun@acm.org

ABSTRACT

Recently a number of concepts, such as software factories, model-driven software development, and language-oriented programming, advocate the use of model-driven domain-specific languages to express domain models. In contrast to many traditional uses of domain-specific languages, this approach leverages the systematic use of domain-specific languages. In this position paper, we identify open issues in this approach, arising when the domain-specific languages are complex or constantly evolving. We introduce the concept of a domain-specific language product line, which provides a common, tailorable language infrastructure to enable the rapid, model-driven creation and evolution of domain-specific languages. Moreover, we propose our vision to use a domain-specific language product line as a conceptual glue between the many concepts and approaches used inside a software factory.

1. INTRODUCTION

Domain-specific languages (DSL) are usually little languages that are particularly expressive in a certain problem domain. There is a recent interest in using DSLs in approaches such as software factories [11], model-driven software development (MDSD) [17, 19], model-driven architecture [15], and language-oriented programming [4] (see also [7]). These approaches use DSLs as domain modeling languages, i.e. languages which represent the abstractions familiar to the domain experts. A DSL either may have a textual or graphical representation (concrete syntax). This concrete syntax is mapped to a language model (abstract syntax), which describes the modeling elements with which models in the respective domain can be created. The semantics of the models are usually defined by model transformation and code generation. Consequently, the DSL itself is mainly defined using models (and meta-models). In the following, we refer to this approach using the term *model-driven DSL*.

Creating a model-driven DSL usually means to perform three steps [7]:

1. A *language model* is defined to formally specify the *abstract syntax* of the DSL. Often the language model is derived from a formal meta-model, similar to the UML 2 meta-model. Also other language models are possible: for instance, in [11] context free grammars are discussed as an alternative, but the meta-model approach is advocated. If a UML-like meta-model is used, UML profiles with stereotypes, tagged values, and constraints (e.g. in OCL) are used to formally model the extensions to the UML needed for defining the DSL.
2. A suitable *concrete syntax* is defined, e.g. graphical symbols or a textual syntax, which is used by users of the DSL. Because the concrete syntax represents the concepts in the abstract syntax, usually there is a correspondence between concrete and abstract syntax elements.
3. A generator translates the DSL into an executable representation. To do this, it must map the elements of the concrete syntax to instances of the abstract syntax, confirming to the formal language model of the DSL. From the abstract syntax, code in the target programming language is generated. Maybe different model transformation steps are happening in between. This can be done using a variety of model transformation and code generation techniques. In practice the generator must be able to define the *semantics* of the DSL.

Let us consider an example; a visual DSL for a workflow designer should be defined:

- Firstly, a UML 2 based language model is defined to represent the abstract syntax, which specifies how the various language elements of the workflow DSL interact. This model represents each language element as a class. That is, the workflow DSL's language model contains classes for workflow elements like tasks, transitions, joins, splits, etc. UML relationships are used to model the potential interactions of language elements. For instance, tasks can be connected to each other using transitions, so there is an association between these two classes. Finally, UML stereotypes and OCL constraints are defined to model those aspects of the workflow DSL that cannot be expressed in native UML. For instance, an OCL constraint is provided so that an "end task" is not allowed to have outgoing transitions. All workflow DSL classes are stereotyped as "DSLElement" to indicate the membership in a DSL to the generator.
- Secondly, as a concrete syntax, visual symbols are defined for each of the language elements of the DSL. The abstract syntax model in UML is annotated with tagged values indicating the names of the visual elements, so that the generator can map the concrete syntax elements to abstract syntax elements.
- Thirdly, the code generation to the target programming language Java is defined. Each concrete syntax element is mapped to an instance of the respective abstract syntax class, and it is checked that no constraint is violated. Predefined hand-crafted Java code and generator templates define the semantics of the DSL's abstract syntax elements. The generator composes the generated Java code and the hand-crafted Java code using the generation instructions in

the generator templates (e.g. following the patterns from [20]).

Traditionally, DSLs are often defined in an ad hoc fashion. Examples are small, declarative languages for domain abstractions as they have been developed in many research projects, little languages in the Unix world, domain-specific language extensions to Lisp defined using Lisp macros, and domain-specific language extensions in many scripting languages. The model-driven DSL approach, in contrast, introduces a systematic process to define DSLs. Also, it reuses the generator infrastructure and the predefined meta-models for the definition of DSLs. Finally, the approach provides a clear concept to combine object-oriented models with DSL concepts.

There are, however, a few problems, when the model-driven DSL approach is applied for creating more complex DSLs, multiple DSLs are needed, or the DSLs need to be constantly evolved. First of all, the UML-like meta-models are not really designed for creating languages, and thus many UML stereotypes and OCL constraints are needed, which are rather cumbersome to use. Language evolution of the DSL means to modify all the artifacts mentioned above, which hinders constant language evolution. If more than one DSL is needed, the reuse between the language realizations is rather limited. For instance, it would be helpful to reuse abstractions for complex language-related tasks, like complex parsing solutions or control structures in the language (like loops, blocks, exception handling, etc). Finally, there are languages used within the model generator (e.g. the generator templates' language in the example above), and there is no reason why these languages should not be defined like all other DSLs. So far only some of these issues have been brought up as research issues in the MDSD literature [21], but no integral solution has been proposed.

In this paper, we propose an approach to extend the concepts of model-driven DSLs to overcome the problems mentioned above. In particular, the goal is to offer concepts to develop DSLs using a *DSL product line*. That is, the DSLs themselves are developed using the same processes and principles like all other products in a product line or software factory. We achieve this goal by combining the concepts of model-driven DSLs with our earlier research work on language engineering of tailorable languages (see [26]). This work essentially integrates concepts of tools suitable for rapid language development (such as interpreters, dynamic languages, homoiconic languages, macro languages, etc.) with concepts from object-oriented design, meta programming, language integration, and language embedding. A tailorable language provides a language infrastructure that can be used to realize all the concepts of model-driven DSLs and combines them with rapid language development to overcome the problems sketched above. In this paper, we sketch how a tailorable language can be used to create and evolve DSLs as products of a DSL product line.

Please note that this position paper reports on a work in progress. All concepts discussed in Section 2 have been developed as prototypes and used in a few projects (industry and research projects). The integral view proposed in this paper (see especially Section 3), however, has not yet been applied on a broader scale. Also, there is still a need for further integrative concepts and the development of prototypes (see Section 4 for a discussion of future work).

2. DSL PRODUCT LINES

In this section, we describe the elements of our concepts of DSL product lines. Please note that the concepts described in the Subsections 2.1-2.3 are in our view mandatory, whereas the concepts in Subsections 2.4-2.6 are highly valuable but optional. In Subsection 2.7 we present an example configuration of a DSL product line.

2.1 DSL language infrastructure and target programming language

Product lines distinguish common assets provided by the product line to all products, and variable assets, which are product-specific variations that are bound to predefined variation points in the common assets. We use the term *DSL language infrastructure* to refer to the common implementation assets of a DSL product line, i.e. those components that are reused across all DSLs derived as products from the DSL product line. For instance, in our work we use an interpreter and the common parts of the language model as components in the language infrastructure, but depending on the implementation other kinds of components like compilers, byte-code engineering libraries, parsers, etc. might also be part of the language infrastructure.

Above, we have distinguished the DSLs and the *target programming language*. It might be non-obvious why we cannot either use an extension of the target programming language as a language infrastructure for the DSLs or, alternatively, the language infrastructure itself as the target programming language. In many projects, the target programming language is given by outer influences. For instance, in many of our recent projects Java or C were used, mostly because of business decisions or for reasons of legacy software integration. We believe such requirements are quite typical. These languages, however, do not directly support the rapid definition of domain-specific language extensions; therefore, DSLs are usually defined on top of more powerful language infrastructures: in the examples above we have mentioned model-driven environments for model-driven DSLs, and scripting languages and Lisp macros for traditional DSLs. These more powerful language infrastructures, in turn, are selected only in seldom cases as target programming languages. Please note that our concepts are also applicable in case the language infrastructure and target programming language are identical, but because this does happen only rarely, we make the distinction here.

One important property of the language infrastructure is *embeddability* in the target programming language. "Embeddable" means that the embedded language infrastructure is used like a library of the embedding language; i.e. the embedding language controls the embedded language infrastructure. Operations in the embedded language infrastructure can be invoked from the embedding language, and the embedded language can call back into the embedding language. Please note that many existing languages support embeddability, and thus concepts and implementations can be reused. For instance, most scripting language, such as Tcl, Python, Ruby, and Perl, are embeddable in C/C++, some of them also in Java. There are many other languages written in Java. The Java Native Interface supports embedding C programs in Java. In other words, the problem of language embedding is well understood, and many successful examples of embeddable languages exist, which can be reused.

2.2 Kinds of DSLs

To be able to use one language infrastructure for all DSLs in a DSL product line, we first must delimit the kinds of DSLs that are in focus of this paper. We distinguish two kinds of DSLs needed in a model-driven DSL approach:

- DSLs for the application domain, as they are proposed in software factory concepts [11] and MDSD approaches [17, 19] alike. For instance, the workflow DSL in the example in the previous section is a DSLs for the application domain.

- DSLs for the domain of code generation and model transformation, e.g. template languages, model transformation languages, glue languages, etc. used inside of the generation architecture, offering no application domain related language elements. For instance, the generator templates in the example in the previous section provide a DSL for code generation.

The reason why we add the domain of code generation and model transformation to the targeted DSLs is that we want to reach the goal that the same language infrastructure can be used for *all* languages used in models, transformations, and applications. In other approaches, such as [17, 19], a special template language is used for models and transformations, and the DSLs for the application domain are developed from scratch using models and code generation. Both kinds of languages have different language concepts, and the developers must learn all the different languages. Our goal is to provide one tailorable language concept as part of the language infrastructure instead, from which all abstract syntaxes of DSLs can be derived. Thus all languages have similar syntactic concepts (i.e. they are easier to learn for developers), and, at the same time, common implementations, e.g. for parsing and common control structures, can be reused across all DSLs.

DSLs for the domain of code generation and model transformation are always translated into the target programming language at generation time. This is often also the case for DSLs for the application domain, but there are also DSLs for the application domain which are interpreted at runtime. Just consider the workflow DSL used as an example above: its concrete syntax is visible to the user, and thus its language infrastructure plus all DSL-specific extensions need to be deployed with the product.

We can thus distinguish two kinds of DSLs for the application domain:

- A DSL used only for modeling purposes before generation-time. The generator transforms the code in this DSL to code in the target programming language.

- A DSL that is deployed and used for domain-specific modeling in the runtime system of a product; the DSL must be interpreted and mapped to implementations of abstract syntax elements at runtime. The implementations of abstract syntax elements are generated by the generator from the abstract syntax specification.

2.3 On the role of interpretation

An integral approach for building DSLs should be able to support all kinds of DSLs introduced in the previous section. An important insight is that this is only possible when the DSL language infrastructure supports some kind of interpreter, because without runtime interpretation it is not possible to build DSLs which are deployed and used at runtime. Please note, even when using the purely model-driven DSL approach sketched in Section 1, in fact, a simple, hand-crafted interpreter for the workflow DSL is generated and deployed to the product.

Of course, it is possible to use only hand-crafted interpreters and only for that purpose. We argue against this practice for two reasons: firstly, hand-crafting interpreters is tedious and the resulting interpreters are usually much less powerful than existing interpreters. Thus, instead, we propose to reuse an existing interpreter as part of the language infrastructure. Secondly, an interpreter is also a useful tool to create and evolve the DSLs which are only used at generation time. Finally, if the same interpreter is used as a language infrastructure for all DSLs, we ensure a consistent syntax across all DSLs.

That means, for DSLs which are deployed and used at runtime, the interpreter is used as an execution environment for the DSL code, and maps the DSL invocations to the abstract syntax of the language. This mapping is performed using the split object approach, introduced in Section 2.5. For the DSLs that are interpreted at generation time, the interpreter is used as a special-purpose generator, which creates other models and/or code in the target programming language. The generated models and code are composed with the models and code generated by other generators.

2.4 A tailorable language as a foundation for a DSL product line

An important aspect has not been discussed so far: if one language infrastructure should be the foundation of a variety of different DSLs, how can this language infrastructure effectively support all different language concepts and at the same time allow for rapid creation and evolution of DSLs?

As a solution to this problem we propose to use the *tailorable language* concept (see also [26]) as a foundation. Language tailorability means that a language is designed to by adapted to new language concepts. We propose to define a dynamically interpreted language that is flexibly adaptable (tailorable) to the context in which it is used. The language representation (syntax) and the interpretation of that language (its semantics) can be tailored to each of the required DSLs, but the general language infrastructure can be reused for all DSLs. The goal is to be able to rapidly design and change (i.e. evolve) language concepts for the DSLs, even though different DSLs might have quite diverse syntaxes and semantics. Moreover, the mapping to the target programming language's concepts is equally important.

As argued above, we propose to make an existing, interpreted language tailorable, in order to be able to reuse its interpreter. A second important language characteristic is that the language should be able to evaluate data, provided in the language, as code. Such languages are called *homoiconic languages* [13]. This language property is illustrated by the simple example in Lisp below, in which we first assign to `a` a program fragment which assigns 1 to `b`. Later we evaluate the program fragment that `a` refers to using `eval`. The consequence is that the code in `a` is executed and `b` is set to 1.

```
(setf a '(setf b 1))
; ...
; some time later
; ...
(eval a)
```

Besides the Lisp family of languages, many other languages are homoiconic, including Tcl, Perl, Prolog, and Smalltalk. This language property enables us to change the language meaning and representation rapidly (at runtime if necessary), and lets us experiment with languages easily.

In addition, we require a language with a *dynamic object system* to tailor the language to the DSL concepts. In a dynamic object system all relationships of an object, including class and superclass relationships, might possibly be changed at any time. Together with the homoiconic property, we are able to redefine the language's syntax and semantics to the needs of the DSL. In our prototype of a tailorable language, Frag [25], one object can have multiple, dynamic classes, which can be composed as mixins [3]. Many other dynamic programming techniques, such as dynamic aspects [2, 1], meta-objects [14], or meta-classes [6], can be used equivalently to add behavior to classes dynamically and transparently. Moreover, a number of patterns are described to implement dynamic object systems in more static languages, such as Object System Layer [9] or Type Object [12].

Reflection allows us to query the runtime state of the object system at any time. We require two kinds of reflective properties in our tailorable language:

- *Structural reflection (introspection):* If we are able to change the language meaning and structure (at runtime), an important goal is to be able to find out the current composition of the objects and classes at any time. Therefore, we need to offer *introspection options* for each language element. Examples are introspection options for type relationships, superclass relationships, associations, and aggregations. This feature is used as a "memory" of how the object system is tailored. Any other structural reflection technique can be used equivalently.

- *Control flow reflection:* Structural reflection is only one part of the current context of an object. The second part is the behavior or control flow context in which the object is invoked. The behavior context is in many interpreted languages handled by a callstack. Thus, the callstack should be fully accessible from within the language. This way we can find out, for instance, which component has called which other component. This feature is used to allow for conditional composition and configuration behavior on basis of the current control flow. Any other techniques that allows for access to the current control flow of the invocations can be used equivalently. For instance, some AOP approaches allow us to intercept the control flow.

We call a language (and its architecture) supporting the properties sketched above a *tailorable language*. These language features are useful to create and evolve DSLs. To illustrate this let us consider an example: a DSL for method transformations that can operate on models written in other DSLs (see [23] for a detailed description of this DSL which is an extension of our prototype language Frag [25]). In this DSL the interpreter can use structural reflection to find and introspect all method implementations. We can use the homoiconic language property to redefine method definitions at runtime. Structural reflection is also used to quantify over class and method names. For instance, wildcards like "`*`" can be used in the introspection options. This way we can specify method transformations like: "transform all methods having the name `Connection*`". Control flow reflection is useful to adapt the code that is introduced by method transformations. This way, we are able to achieve similar quantifications over objects as offered by the joinpoint language in AOP approaches.

In summary, tailorability enables us to rapidly adapt a language to support new language features and reuse many functionalities of the language infrastructure. Even though using a tailorable language is not mandatory for developing a DSL product line, it significantly eases the creation and evolution of DSLs.

2.5 Language bindings by generated split objects

Above we have proposed to embed the language infrastructure of the DSL product line in the target programming language. A consequence of embedding the DSLs is that we need bindings between the code in the language infrastructure of the DSL product line and the target programming language. We propose to generate these bindings. The first questions that arises is: where are the best binding points? Here, we propose to use the abstract syntax of the DSLs as a starting point. Our idea is to reflect each abstract syntax element of a DSL in the target programming language, and make both representations of the abstract syntax element be able to communicate with each other.

We use the Split Object pattern [22, 24] to realize this goal: a split object is an object that exists in two languages, but is treated like a single instance. This way we intercept invocations that are not implemented in the DSL, and forward them automatically into the target programming language. To realize this feature we use a *dynamic dispatcher* in the tailorable language (DSL), and a program generator in the target programming language. In our prototypes, we use *aspect-oriented programming* with AspectJ to intercept invocations in Java, and the generator SWIG [18] to integrate split objects for C/C++. Any other technique that is capable of intercepting all invocations to an object can also be used to realize the Split Object pattern (see [22]).

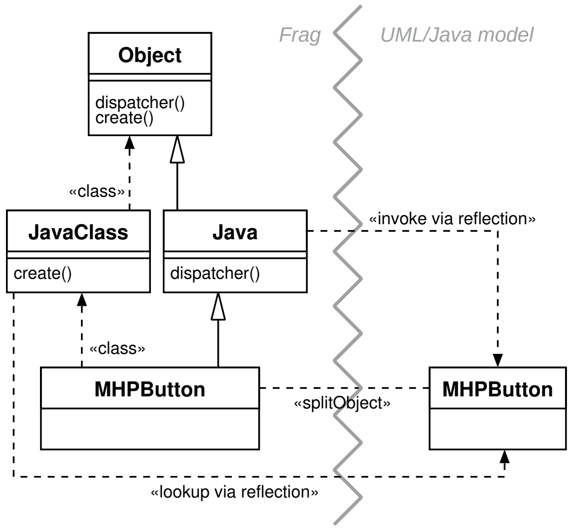

Figure 1: Split object structure example

Please note that the tailorability property of the DSL is important for language integration via split objects. Split object classes in the DSL and the target programming language mimic each others class hierarchy. Moreover, a correspondence of class structures and class names between the two languages of a split object class hierarchy enables generation of split objects as glues between the DSL and other models. To be able to do this, the DSL must be able to represent the class and object concepts of the target programming language: we tailor the DSL to support the features and constraints of the target programming language's class and object concepts.

Let us consider an example from [26]: a DSL is used for configuring GUI elements for the MHP. The MHP specification [5] is a generic set of APIs for digital content broadcast applications, interaction via a return channel, and internet access on an MHP terminal. Typical MHP terminals are for instance digital set-top boxes, integrated digital TV sets, and multimedia PCs. To provide for configurability of the user interface, we define a DSL which is mapped to the model of the platform and thus to the resulting Java classes.

Figure 1 shows a split object `MHPButton`, which is an abstract syntax element of the DSL defined as an extension of Frag [25]. The Frag class has a counterpart in Java, also named `MHPButton`. When an invocation cannot be dispatched for the DSL class, the Java class `MHPButton` is automatically looked up via Java Reflection, because the Frag class `MHPButton` has the class `JavaClass` as a class and `Java` as a superclass. These two classes are provided by Frag's Java split object framework. All methods that are not found for the Frag half of the split object are automatically forwarded to the Java half by the `dispatcher` method, which is automatically invoked. The invocations to the Frag method `create` are forwarded to the respective Java constructor, and automatically a split object half is created in Java for each Frag object derived from MHPButton. All details of forwarding invocations, such as type conversion, are automatically performed in the background.

2.6 Generating other artifacts

In our concept, the interpreter is used as a generator for language-related models, model transformations, and their implementations. Split objects are generated as bindings between DSLs and other models (and their generated implementations). Sometimes other generation tasks are required.

For instance, the language models for abstract syntaxes are in our work mostly defined in a textual form because this enable rapid changes to the language models. But sometimes graphical representations are needed as well (e.g. for documentation purposes). In these cases we generate the graphical model representations using automatic graph drawing algorithms from textual model representations (as described in [16]).

2.7 Example: Generation architecture configuration

Figure 2 shows an example of a generation architecture configuration with interpreters. The goal is to create Java code. The main Java models are represented in UML and transformed via a UML-to-Java generator to Java classes. The outputs of this generator are used by the DSL interpreter and generator. In addition, hand-crafted Java code (combined with the generated Java classes following the patterns from [20]) is added by the developers.

There are three DSLs: two are from the domain of code generation and model transformation (template language and model transformation language), and one is a DSL from the application domain. The latter is the only DSL that is deployed to the product, the two former DSLs are used only inside the generator architecture.

In addition to the generated and hand-crafted Java code, the DSL interpreter produces split object bindings between the application domain DSL and the Java classes, a deployable interpreter for the application domain DSL, the deployable application domain DSL code (embedded in Java), and glue code to lets everything work smoothly together.

Finally, the DSLs' language models, describing the abstract syntax, are sent to an UML figure generator (like [16]) to produce UML diagrams for documenting the predefined DSL models with UML 2.

3. USING THE CONCEPTS IN SOFTWARE FACTORIES

Software factories [11] were one of the approaches which we have identified in Section 1 as an approach that could benefit from DSL product lines. The software factory concept proposes the automated creation of software and the integration of approaches from various disciplines, such as model-driven software development (MDSD), software product lines, software architecture, patterns, aspect-oriented software development (AOSD), component-based software development (CBSD), service-oriented architectures (SOA), and others. More precisely, a software factory uses MDSD to create a software product line in which the other named concepts are combined.

The integral approach taken by software factories leads to another important motivation for using our DSL product line approach. One important aspect to software factories is that the concept reduces the complexity and increases the changeability by combining approaches from different disciplines. But this creates a new kind of complexity: developers must master all the different concepts used in a software factory, such as (different kinds of) components, patterns, aspects, services, models, generators, etc. Often multiple paths for integrating these artifacts into one product line exist. We argue an explicit, intuitive, and easy-to-use concept for developing DSLs in terms of a DSL product line can be used to design DSLs for integration of these concepts. The DSL product line's language infrastructure provides *one* integrated DSL concept with which all other concepts can be glued and configured. In other words, a DSL product line, following our concepts, could be used as a "conceptual glue" between the different kinds of models and artifacts.

The consistent use of a DSL product line throughout the software factory solves a number of other issues: for instance, the common architectural foundation can be used to provide traceability between the different glued artifacts (e.g. as a central indirection, the interpreters of the DSLs can record relevant trace links). Also, other model representations, e.g. for documentation purposes, can easily be built.

4. CONCLUSION

In this position paper, we have combined our earlier research work on tailorable languages (see [26]), split objects (see [24]), and the use of high-level languages in product line architectures (see [10, 8]) with the concepts of model-driven development of DSLs. Our approach offers the consistent use of one language framework for all kinds of DSLs in a DSL product line identified in Section 2.2. Our approach allows for a more rapid design and evolution of domain-specific languages, without sacrificing the benefits of model-driven DSLs. Finally, our approach enables the reuse of language-related generation code, integration models, and powerful language-manipulation features offered by the interpreter.

So far, we have implemented all concepts described in Section 2 as prototypes and defined a number of DSLs using the concepts in different projects. Our plan is to further develop and integrate the concepts and prototypes. In Section 3, we proposed to use the DSL product line as a conceptual glue that reduces the conceptual complexity of a software factory. This, however, is still work in progress.

Even though our concept entails some novel concepts and some novel combinations of concepts, most of the technological components are broadly available. For instance, there are many languages that can easily be extended to serve as a tailorable language, many interpreters offer generative features, existing generators can be used to generate split object bindings (like SWIG [18]) or UML model visualizations (like [16]), and so on. Our concepts require the developer to combine the most suitable tools to a DSL product line. The concepts can be applied with any model-driven approach, but they make especially sense in the context of a larger product lines or software factories, because the time and effort for creating and evolving a DSL language infrastructure should not be underestimated, but it is likely to be small in comparison to the whole product line or software factory.

5. REFERENCES

[1] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, March 2004. ACM Press.

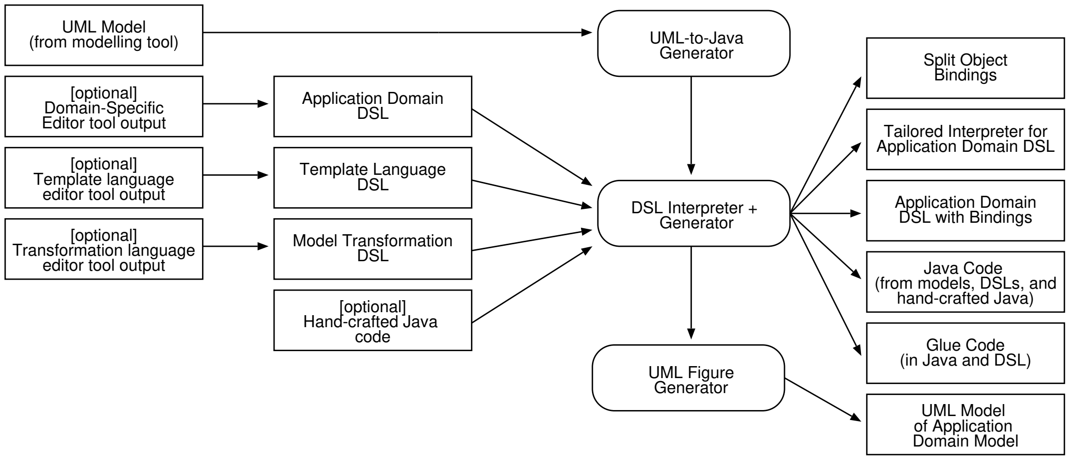

Figure 2: Example of a generation architecture configuration

[2] J. Boner and A. Vasseur. AspectWerkz. http://aspectwerkz.codehaus.org, 2004.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming (OOPSLA/ECOOP'90)*, volume 25 of *SIGPLAN Notices*, pages 303–311, October 1990.

[4] S. Dmitriev. Language oriented programming: The next programming paradigm. Onboard Magazine, http://www.onboard.jetbrains.com/is1/articles/04/10/lop/index.html, October 2004.

[5] ETSI. MHP specification 1.0.1. ETSI standard TS101-812, October 2001.

[6] I. Forman and S. Danforth. *Putting Metaclasses to Work – A new Dimension to Object-Oriented Programming*. Addison-Wesley, 1999.

[7] M. Fowler. Language workbenches: The killer-app for domain specific languages? http://www.martinfowler.com/articles/languageWorkbench.html, June 2005.

[8] M. Goedicke, C. Koellmann, and U. Zdun. Designing runtime variation points in product line architectures: Three cases. *Science of Computer Programming*, 53(3):353–380, 2004.

[9] M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceedings of 5th European Conference on Pattern Languages of Programs (EuroPlop 2000)*, Irsee, Germany, July 2000.

[10] M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(1):1–30, 2002.

[11] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. J. Wiley and Sons Ltd., 2004.

[12] R. Johnson and B. Woolf. Type object. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[13] A. Kay. *The Reactive Engine*. PhD thesis, University of Utah, 1969.

[14] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[15] OMG. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.

[16] D. Spinellis. On the declarative specification of models. *IEEE Software*, 20(2):96–95, 2003.

[17] T. Stahl and M. Voelter. *Modellgetriebene Software Entwicklung*. D.Punkt, 2005.

[18] Swig Project. Simplified wrapper and interface generator. http://www.swig.org/, 2003.

[19] M. Voelter. Model-driven software development tutorial. http://www.voelter.de/services/mdsd-tutorial.html, 2005.

[20] M. Voelter and J. Bettin. Patterns for model-driven software-development. In *Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPlop 2004)*, Irsee, Germany, July 2004.

[21] H. Wada. Modeling turnpike: a model-driven framework for domain-specific software development. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems – Doctoral Symposium*, Montego Bay, Jamaica, Oct 2005.

[22] U. Zdun. Some patterns of component and language integration. In *Proceedings of 9th European Conference on Pattern Languages of Programs (EuroPLoP 2004)*, Irsee, Germany, July 2004.

[23] U. Zdun. Supporting incremental and experimental software evolution by runtime method transformations. *Science of Computer Programming*, 52(1–3):131–163, 2004.

[24] U. Zdun. Using split objects for maintenance and reengineering tasks. In *8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 105–114, Tampere, Finland, Mar 2004.

[25] U. Zdun. Frag. http://frag.sourceforge.net/, 2005.

[26] U. Zdun. Tailorable language for behavioral composition and configuration of software components. *Computer Languages, Systems and Structures: An International Journal, Elsevier (accepted for publication)*, 2005.