A DSL Toolkit for Deferring Architectural Decisions in DSL-Based Software Design

Uwe Zdun
Information Systems Institute
Vienna University of Technology
Vienna, Austria
zdun@infosys.tuwien.ac.at

A number of mature toolkits and language workbenches for DSL-based design have been proposed, making DSL-based design attractive for many projects. These toolkits preselect many architectural decision options. However, in many cases it would be beneficial for DSL-based design to decide for the DSL's architecture later on in a DSL project, once the requirements and the domain have been sufficiently understood. We propose a language and a number of DSLs for DSL-based design and development that combine important benefits of different DSL toolkits in a unique way. Our approach specifically targets at deferring architectural decisions in DSL-based design. As a consequence, the architect can choose, even late in a DSL project, for options such as whether to provide the DSL as one or more external or embedded DSLs and whether or not to use an explicit language model.

1 Introduction

Developing custom software for a specific target domain is a challenging task. In this context, Domain-specific Languages (DSLs) receive a constantly growing attention in recent years [Fow05, Hud96, MHS05]. DSLs have gained popularity through the advance of so-called language workbenches [Fow05]. Language workbenches take the well known language-oriented programming approach and use Integrated Development Environment (IDE) tooling to make it a viable approach. Examples of language workbenches are Model-driven Development (MDD) tools [SV06] such as XText [Eff08], software factories [GS04], MetaCase [KT08], OSLO [Mic08], and MPS [Dmi04]. In contrast to simple language-oriented programming approaches, focusing mainly on parser development and related activities, language workbenches support many aspects of DSL design and development, including for example the transformation of DSL code to a platform or the development of language editors for the DSL. In addition to language workbenches, some other DSL toolkits exist that support many aspects of DSL design and development. For example, DSLs in dynamic languages [Fre06] are typically developed as embedded DSLs that can use the existing tooling and platforms of the dynamic language.

With the DSL toolkits becoming more mature, DSLs are not only used in niches anymore, but receive widespread adoption. For many architects, DSLs are an interesting approach that can potentially be used in projects where domain knowledge is hard to capture. DSLs can help here, as they come with the promise to enable involving domain experts more closely in the software design activities.

Architecture can be seen as a set of design decisions rather than (only) a set of components and connectors [JB05]. In this view, the architect needs to understand which architectural decisions are influenced by the choice of an approach or a technology. The above named toolkits require architects to make important architectural decisions about the DSLs in a project early on – when the DSL toolkit is chosen. Example decisions are: using an embedded or external DSL approach, using an implicit or explicit language model (and which kind of explicit language model should be used), how to define the static semantics of the DSL, how to integrate the DSL with the host language, and the choice between generation, transformation, and interpretation for defining the DSL's execution semantics. Often significant changes or even a complete reimplementation of the DSL in another technology would be necessary to change some of these architectural decisions later on, once a DSL implementation has been completed. For example, in one of our projects it has been decided during the early DSL design steps to implement an external DSL in a software technical domain using the Microsoft DSL tools (see [GS04]), mainly because of the attractiveness of the tooling. The choice for using a graphical, external DSL was hence made implicitly by the choice for the tool. In later stages of the DSL design process, the project learned that the graphical, external DSL approach was not highly suitable for the daily work tasks of the software engineers working with the DSL. To change the decision to a textual, embedded DSL, however, meant to reimplement the entire DSL prototype in a different technology.

In our experiences, it is often necessary to focus in early stages of DSL projects on getting the domain abstractions right. Work collaboratively conducted by the domain experts and the software designers is needed, and approaches, such as rapid application development, experimenting with DSL ideas, and agile methods with many iterative feedback cycles, are highly useful in DSL-based design and development. The dilemma is: On the one hand, toolkit-specific architectural decisions, such as the ones named before, should be deferred till the domain abstractions and requirements are sufficiently understood, but, on the other hand, experimenting with the DSL requires some realistic DSL toolkit that preselects many options for the architectural decisions listed above.

Of course, it is possible to solve this dilemma by developing a throw-away prototype that is replaced by another technology, once a first sufficient understanding of the domain has been reached. In contrast, in this paper, we present a study investigating on the feasibility of the idea to provide a DSL language toolkit that supports incrementally evolving a DSL and still allows DSL architects to change fundamental architectural decisions later on.

We present a language called Frag (available from [Zdu10]) as a toolkit for DSL-based design and development (focusing mainly on textual DSLs). In this language, we can either use any kind of Frag objects to implement the DSL or base the DSL on an explicit language model – similar to models in model-driven DSL approaches (see [SV06]). In our approach, we combine the external DSL and the embedded DSL approaches in a unique way: For each language model, Frag provides an intuitive embedded DSL syntax. In addition, one or more external DSLs can be added and mapped to Frag, using a flexible rule-based parsing approach that is embedded in the Frag language. In Frag, developers can define the execution semantics of the DSL using interpreted code, transformation templates, or dynamic language features. Static semantics can be given using custom code, an OCL-style constraint language, or transformation templates. The use of the host language in the DSL can be forbidden, or it can be supported using various language extension or specialization techniques. Frag is designed so that these alternative choices for the architectural decisions on DSL-based design can be changed during a DSL project with a mostly local change impact.

All specifications of DSLs in Frag are provided using DSLs that are embedded in Frag (which is itself embedded in Java; hence every Frag DSL can be used in Java applications). Such embedded DSLs are provided for specifying language models, constraints, and parser specifications (including a lexical scanner and mapping specification DSL). That is, the Frag approach for specifying DSLs follows the DSL approach itself.

As our contribution is mainly a novel combination of known concepts from existing tools, our insights are not limited to Frag. By following our approach, with foreseeable effort, other DSL toolkits can be modified or adapted to defer specific architectural decisions. The overall goal is to let the architect decide about fundamental architectural decisions in DSL-based design, and not the chosen technology. This is particularly relevant for the analysis, design, and implementation parts of the DSL development process. Our approach aims at reduced costs through lower efforts required in cases major design decisions on the DSL must be changed in later stages of the development process. This might also lead to a higher quality of the DSL, as more time can be spent on domain analysis and design of the DSL, and less time must be spent on adapting the DSL development to the DSL tool used.

This paper is structured as follows. First, in Section 2, we provide background on DSL concepts and the major architectural decisions that need to be made during a DSL project. Next, in Section 3, we present our approach to DSL development. In Section 4 we provide an extensive example on all aspects of our approach. Next, in Section 5 we use this example to compare to a set of related approaches and especially the decision making in these other approaches in detail. Section 6 reports on experiences in various DSL projects in which we have used our approach. In addition, we measure the costs of our approach in terms of a performance and scalability evaluation for each of the major parts of the Frag framework. Finally, we conclude.

2 DSL Concepts and Architectural Decisions

A *domain-specific language* is a tailor-made language that does only provide abstractions suitable for one particular problem domain. In recent years, domain-specific languages received a constantly growing attention, especially in the area of model-driven development [GS04, KT08, Sch06, Sel03, SV06]. The basic idea of DSLs, however, already has a long history (see, e.g., [Ben86, HB88]). In the Unix context, for example, DSLs have a tradition as so called “little languages” or “mini languages” (see [Ben86, Ray03]), and in the context of languages such as Common Lisp the development of “embedded languages” is promoted (see [Gra93]). Other popular examples of DSLs are LaTeX for Typesetting [Lam94, Lat08], HTML for hypertext web pages [PAC+02], the (extended) Backus-Naur Form (BNF) for syntax specification [ISO96], or SQL for database queries and manipulation [CB74, ISO03].

DSLs can be defined for technical users, such as software developers or architects, as well as non-technical users, such as business analysts, managers, biologists, and so on. In either case, the DSL should make the domain-knowledge explicit, and help the DSL users to accomplish their work tasks. For example, if the DSL is built for supporting business analysts who need to describe compliance concerns in the business processes of an organization, then the DSL could for instance offer abstractions to describe compliance requirements, risks, and controls, and support linking them to the existing business process descriptions. In addition, the DSL implementation should support the business analyst in his daily work tasks, for example by automating certain recurring work steps, such as generating a compliance documentation for audit purposes. In general, we can distinguish two different styles of DSLs (see, e.g., [Fow05]) and choosing one of them is an important architectural decision in DSL-based design:

- An *embedded DSL* (also called *internal DSL*) is an extension to an existing programming language and uses the syntactic elements of the underlying language, hosting the DSL. Typical examples of embedded DSLs are the DSLs provided in the Ruby on Rails framework, the Ruby DSLs described in [Fre06], and the Haskel example presented in [Hud96].
- An *external DSL* is defined in a different format than the intended host language(s) and can use all kinds of syntactical elements. Typical examples of textual external DSLs are DSLs developed using openArchitectureWare's XText framework [Ope08] or Microsofts' OSLO framework [Mic08], and many examples of graphical DSLs such as the domain-specific modeling languages presented in [KT08].

From a user's point of view, the DSL consists of *language elements*. The core definition of these language elements is provided by the *language model* that specifies the concepts of the DSL's target domain. For instance, if the target domain is Service-oriented Architecture (SOA), then elements such as service, process, and business object are part of the language model. If the target domain is core banking, then elements such as account, bond, and customer are part of the language model. The MDD paradigm advocates to specify the language model (usually called DSL metamodel) explicitly using a modeling language such as UML or EMF. Other implementation options for DSLs, such as embedded DSLs based on scripting languages or dynamic languages [Fre06], do not imply an explicit language model – it is often implemented and hidden in the DSL implementation source code. In addition, there are also other options for specifying the language model than meta-models, such as DTDs or XML Schema for XML-parser-based DSLs [Fow08] or context-free grammars for DSLs based on parser generators [Par07]. If and how an explicit language model is supported is a second central architectural decision.

In order to use the DSL, we must define its semantics, too. For many DSLs, static semantics are defined through constraints for the language model, as well as program code defining the execution semantics of the DSL. In the MDD literature, the language model and the static semantics of the DSL together are called the *abstract syntax* of the DSL [GS04]. The architect must decide how static semantics are implemented: Besides constraint languages offered by some of the model-driven toolkits, expressing them in transformations, in custom code, or not at all are options for this decision.

In addition to the abstract syntax, each DSL needs a *concrete syntax*, offered to the user of the DSL. Each DSL can have multiple concrete syntaxes. We can distinguish textual syntaxes, graphical syntaxes, form-based syntaxes, table-based syntax, and so on. The architect must decide which kinds of concrete syntaxes are supported by the DSL. The architectural decision for one or more concrete syntaxes is usually very important, as the concrete syntax is the main user experience of the DSL. This paper mainly focuses on textual concrete syntaxes. Our approach allows DSL developers to add additional other kinds of syntaxes on top of the textual syntax, in the same way as virtually all other textual DSL approaches; hence, in this paper, we do not look deeper into this architectural decision.

Transformations are defined to transform DSL code written in a concrete syntax to another model representation or to code that can be executed on a specific *platform*. In general, a transformation is a directive that defines how one (model) format is to be transformed to another (model) format. In different approaches, the transformations are realized in different ways: In the model-driven approaches, a generator performs the transformations. In scripting and dynamic languages, the concrete syntax code is mapped to directives of the interpreter. A parser generator creates a parser that triggers instructions in a programming language. In all approaches, a programming language is eventually used to execute the DSL and is used as the target of the transformations. This programming language is called the *host language*. The architect can choose among these different options for defining the execution semantics of the DSL.

Sometimes it is useful to use some host language constructs in the DSL. Consider for examples using loops to iterate over statements or substitutions to realize variables. Reuse of host language features is helpful, as reimplementing such features for a DSL can be a substantial effort. But on the other hand the accessibility of host language features can be confusing for non-technical stakeholders using the DSL. Hence, it must be decided whether the use of the host language should be allowed or forbidden in the DSL and to which extent it should be allowed. Usually, it is allowed in embedded DSLs (called language extension in [Spi01]) and forbidden in external DSLs, because these combinations are easy to implement. But also other combinations are possible, like an (e.g., external) DSL that offers only a limited set of host language features of the host language (called piggyback in [Spi01]) or language specialization [Spi01], which describes the creation of a DSL by omitting features of the host language.

The decisions named so far are central architectural decisions on DSL design and will be considered in the rest of this paper. They are summarized in Table 1. Please note that many other architectural decisions on DSL design exist. For example, related to the decisions on the concrete syntax and transformations is the decision if and how syntax extensions and evolution is supported. If more than one related DSLs are developed, architects must decide how the different DSLs are integrated. For making the DSL usable it is often necessary to provide custom error messages that are understandable to the user of the DSL. The architect must decide which error messages are supported and how/when they are triggered. Frag has a positive influence on some of these other decisions, too. For example, many parts of Frag allow developers to easily place custom error messages. However, in this paper we only focus on the decisions summarized in Table 1, as the options for these decision mainly distinguish Frag's approach from the related work.

In an ideal world, the architect could decide for all options of these architectural decisions while gaining experience with the DSL as it is evolving. Unfortunately, most existing DSL toolkits do not leave many of these choices open. That is, the decision for a particular toolkit preselects many other decision options: Once significant design and development with a DSL toolkit has been conducted, it is hard to change the other decisions made by the DSL toolkit designers, if they turn out to be inappropriate. In the next two sections, we illustrate our idea to provide a DSL toolkit that leaves these decisions open – first in general and then using a detailed example.

Table 1: Architectural Decisions on DSL-based Design Considered in this Paper: Summary

Decision	*Alternatives*	*Discussed in*
How should the language model of the DSL be realized?	Explicit Language Model based on a meta-model (e.g., UML model, EMF model, FMF model), a DTD, an XML Schema, or a context-free grammar, Implicit Language Model (language model implemented and hidden in the DSL implementation code)	Section 3.1
Should an external or embedded DSL be developed?	Embedded DSL, External DSL, Hybrid DSL (combining the two former alternatives)	Section 3.2
How should the DSL's execution semantics be defined?	Generator-based Transformations using Transformation Templates or Transformation Rules, Interpretation in a Dynamic Language or Scripting Language, parser that triggers instructions in a programming language	Section 3.3
How should the DSL be integrated with the host language?	Host Language Use Forbidden, Language Extension, Piggyback, Language Specialization	Section 3.4
How should the static semantics for the language model be realized?	Using a Constraint Languages, Using Transformations, Using Custom Code, Not At All	Section 3.5

3 DSL-based Design in Frag

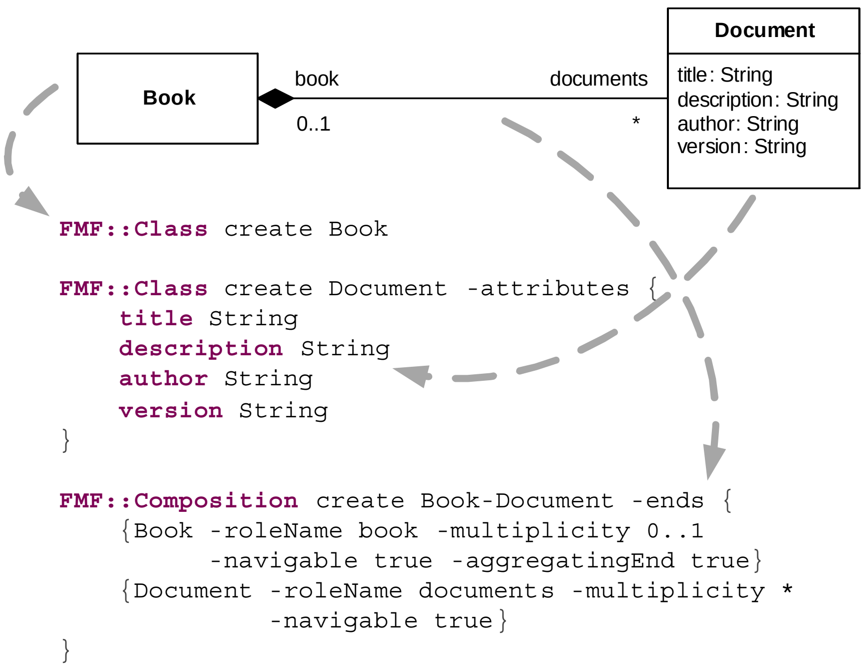
Frag [Zdu10] is a dynamic programming language implemented in Java. Its main goal is the rapid definition of DSLs based on Frag and processing of these languages. Frag is a full-fledged OO language, but in this paper we will only introduce and use its specific features for DSL-based design and development. In this section, we will explain the various steps that must be performed to define a DSL and illustrate the architectural decisions and their alternatives that can be deferred by using Frag. In the next section, we give a detailed example.

3.1 Specifying and Instantiating a DSL Language Model

Regarding the architectural decision "How should the language model of the DSL be realized?" Frag supports two options:

- Because Frag is a full-fledged programming language, a DSL architecture in Frag does not have to be based on an explicit language model. We can also use a structure of ordinary programming language objects to represent the DSL language model – akin to many DSLs in dynamic languages (see [Fre06]). Frag objects can either be implemented in Frag or in Java.

- In addition, Frag implements a modeling framework called FMF that is similar to the modeling frameworks found in model-driven language workbenches [SV06, Fow05]. Using this modeling framework, an explicit language model can be defined. FMF classes and objects are ordinary Frag objects. For this reason, deciding for the FMF option does not limit decision options with regard to other architectural decisions. FMF classes can also be instantiated in Java, too.



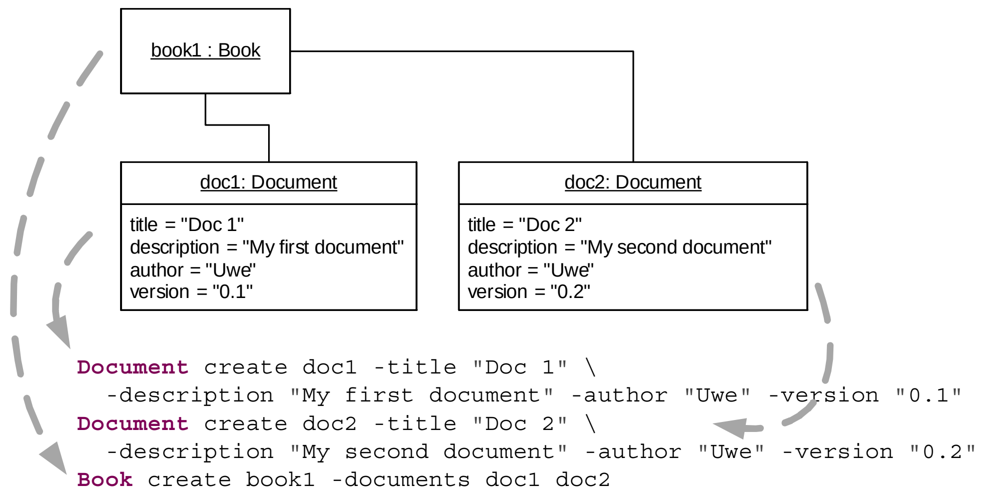
```
FMF::Class create Book

FMF::Class create Document -attributes {
    title String
    description String
    author String
    version String
}

FMF::Composition create Book-Document -ends {
    {Book -roleName book -multiplicity 0..1
          -navigable true -aggregatingEnd true}
    {Document -roleName documents -multiplicity *
              -navigable true}
}
```

Figure 1: FMF Example

FMF defines a meta-model for language models that can be used to specify UML-style language models in Frag. The language model classes are defined using the `FMF::Class` meta-class. FMF introduces also a number of relationships between classes: Dependencies, Associations, Compositions, Aggregations, and Inheritance, as well as extensions using stereotypes, enumerations, and so on. Figure 1 illustrates how a simple UML class model, containing two classes, a few attributes, and a composition relationship, is mapped to Frag code. The translation is in most cases a one-to-one mapping. Once we have defined a language model in Frag, we can use the language model to create instances using the same syntax. Figure 2 illustrates how an instance of the UML model is mapped to Frag code.



```
Document create doc1 -title "Doc 1" \
  -description "My first document" -author "Uwe" -version "0.1"
Document create doc2 -title "Doc 2" \
  -description "My second document" -author "Uwe" -version "0.2"
Book create book1 -documents doc1 doc2
```

Figure 2: FMF Instance Example

As mentioned previously, the alternative for this decision is to use ordinary Frag (or Java) objects to realize the language model. This is illustrated using the same example in Figure 3. In this simple example, the explicit and implicit language model are still pretty similar, but we can already see the important difference that the modeling language abstractions are not explicitly present here: the code in Figure 3 contains no explicit relationships, no cardinalities, no navigability, and no typing of attributes or relationships. Methods are used to implement parts of the language model.

```
Object create Book -defaults {
      documents ""
}
Book method getDocuments {} {
      self get documents
}
Book method setDocuments {docs} {
      self set documents $docs
}
Object create Document -defaults {
    text ""
    title ""
    description ""
    author ""
    version ""
}
```

Figure 3: Implicit Language Model in Frag

In Frag, both alternatives for the decision on the language model realization can be combined with any choice for the other decisions described in Table 1. Changing this decision later on requires to change the code that refers to language model elements, but the effort is rather limited. The new kind of language model needs to be implemented. In addition some changes to parts accessing the language model might be necessary: Class names can stay the same and do not need to be changed. However, fields and relationships need to be accessed with getter/setter methods instead of association role names (this change is a straightforward search/replace task).

3.2 Parsing: Scanner and Mapping Specification

In the previous section, we have already introduced one alternative solution for the decision: "Should an external or embedded DSL be developed?" Any Frag language model (explicit or implicit) automatically offers an embedded DSL syntax. An example of this syntax is shown in Figure 2.

To support the external DSL option, too, FMF models or any other Frag object structures can be created using any parser. For instance, all kinds of parser generators or XML parsers could be used to create external concrete syntaxes for FMF models. However, to enable deferring the architectural decisions related to syntax extension and language integration, we propose a parsing approach based on the composition of lexical scanner rules, followed by a mapping of the scanner's output to the language model.

This lexical scanner approach is similar to the lexer in other parser generator approaches such as ANTLR [Par07]. The mapping described below has the same role as the parser specification in these parser generators, but offers some extensibility constructs (similar to attribute grammars [MZ05]; see below).

The main idea of the lexical scanner approach is that all parsing is performed using rules. The rules are traversed in the order in which they are defined, and if a rule applies it gets executed (i.e., the order of definition defines the priority of the rules). Rules consist of matchers and optionally tokens. A rule starts with the symbol '`%`'. If the matcher matches the scanner's input, it appends the matched input to a scanned elements list. A token is a literal appended to the rule specification. If a token is given, all elements appended by this rule to the scanned elements list get the token. Tokens can be used by a mapping specification (see below) to easier identify certain scanned elements. In addition, there are rules that ignore the matched input. These start with '`%-`'.

To foster reuse and make scanner specifications more readable, a scanner specification can also contain matcher variables, defined using the '`=`' symbol. Matcher variables are simply replaced in rules or other matcher variables. Both variable and rule definitions are ended by a semicolon.

The syntax of matchers in the scanner specification is pretty similar to other lexical parsing approaches that are based on or similar to EBNF. You can define alternatives using '`|`', sequences by concatenating elements with whitespaces, groups of elements using round brackets '`(...)`', character sets using curly braces '`{...}`', repetitions using multiplicities in square brackets '`[...]`' (containing numbers, '`..`', and '`*`'), 'not' using '`!`', and literals in quotation marks.

Frag parsers can be defined using a Java API. In contrast to many other parsing approaches, Frag also offers the option to define parsers in Frag using an embedded DSL. This way, an embedded DSL can contain a parser for an embedded sub-language. That is, deferring the related architectural decision does not only work for one DSL, but also for DSLs integrated in another DSL.

To illustrate our scanner specification approach, Figure 4 defines the syntax of embedded DSL for scanner specifications in its own syntax.

```
# Match variables
COMMENT = "#" (!( "\r\n" | "\n"))[*];
WS = " " | "\r\n" | "\n" | "\t";
RESERVED = WS | "!" | "(" | ")" | "[" | "]" | "{" | "}" | "|" | ";" | "#" | "\"";
STRING = "\"" ("\\\"" | !"\"")[*] "\"";
MULTIPLICITY = "[" (WS | {0123456789.*})[*] "]";
CHAR_SET = "\{" ("\\\}" | !"\}")[*] "\}";

# Rules
%- COMMENT;
%- WS;
% STRING "STRING";
% MULTIPLICITY "MULTIPLICITY";
% CHAR_SET "CHAR_SET";
% "!" "NOT";
% "(" "L_BRACES";
% ")" "R_BRACES";
% "|" "OR";
% ";" "END";
% (!RESERVED)[1..*] "ID";
```

Figure 4: Scanner Specification Example: Syntax Specification of the Embedded DSL for Scanner Specifications

The result of scanning a text is a tokenized list of elements. The literals at the end of '`%`' rules are the tokens assigned by the rules. The tokens can be used later on to easily interpret the scanned elements list.

To ease mapping the scanned elements lists to the language model, we provide a mapping DSL. It basically can map elements, sequences of elements, alternatives, and repetitions in the scanned elements lists. It triggers code fragments that create and fill instances of the language model with the parsed information. Elements that are matched can have a condition that must evaluate to true in order for the element to match. These conditions are similar to attributes in attribute grammars [MZ05], as they can be used to decide whether a rule applies or not.

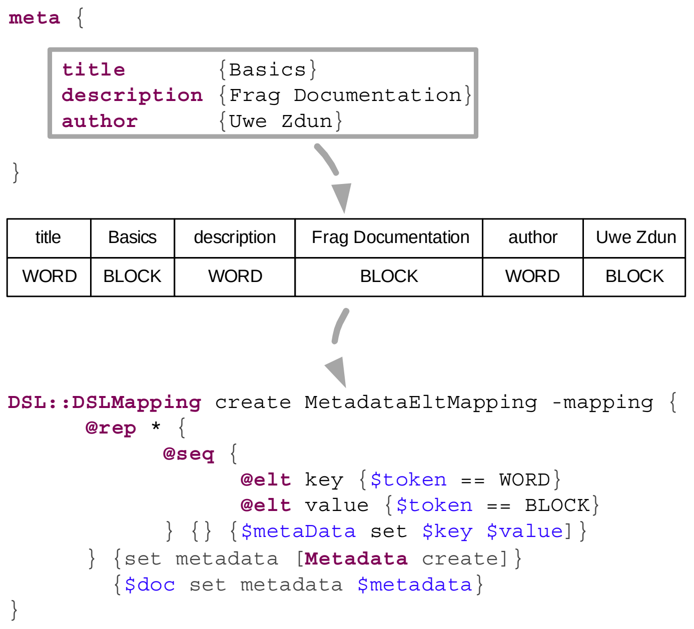
Figure 5 shows a simple DSL mapping for metadata lists that are embedded in other DSL statements. In the example, a sample metadata block is shown. It is also shown how it is parsed into a scanned elements list with tokens (`BLOCK` and `WORD`) (as created by a scanner specification). This list is the input for the mapping specification shown at the bottom of the figure. The mapping specification accepts any repetition of a sequence of two elements, where the first element has the token `WORD` and the second element has the token `BLOCK`. Before the repetition is mapped, a `Metadata` object is created. After each sequence of elements, the metadata keyword and value are set on the metadata object. Finally, when the end of the repetition is reached, the metadata object is set as the metadata for the current document.

A mapping can invoke another mapping (an example is shown in Section 4). This way a mapping can reuse and incrementally extend existing mappings. This is similar to grammar extension through inheritance in attribute grammars in the approach by Mernik and Zumer [MZ05]. However, our approach rather resembles extension through delegation than inheritance.

Our approach to define parsing and mapping specifications within the Frag language allows us to defer the architectural decision on the external vs. embedded syntax until late in the project. In Frag, always an embedded syntax is needed. To change the decision and also provide an external syntax, only the implementation effort for this external syntax is required. It is possible to only use an embedded syntax, to hide the embedded syntax and only offer the external syntax, or to use an hybrid approach and offer both syntaxes to the user. This decision has no influence on other decisions.

3.3 Executing the DSL

As Frag is a dynamic language, we can use the ordinary option for executing a DSL in a dynamic language: Using the interpreter of Frag to execute the language. In addition, as Frag is embedded in Java, we also have the option to use Java classes to implement the DSL execu-



```
meta {
    title       {Basics}
    description {Frag Documentation}
    author      {Uwe Zdun}
}
```

title	Basics	description	Frag Documentation	author	Uwe Zdun
WORD	BLOCK	WORD	BLOCK	WORD	BLOCK

```
DSL::DSLMapping create MetadataEltMapping -mapping {
      @rep * {
            @seq {
                  @elt key {$token == WORD}
                  @elt value {$token == BLOCK}
            } {} {$metaData set $key $value]}
      } {set metadata [Metadata create]}
        {$doc set metadata $metadata}
}
```

Figure 5: Example of a DSL mapping and its input

tion semantics (Frag imports Java classes automatically via reflection). For example, we can extend the `Document` object from Figure 3 with a mixin (an extensional class) that interprets the document and creates HTML code from it, as shown in Figure 6.

In addition to this obvious alternative for the decision "How should the DSL's execution semantics be defined?" we can also support other options in Frag. Frag additionally supports transformation templates, as they can be found in model-driven language workbenches. Using these transformation templates any code can be generated. For example, Figure 7 shows a transformation template generating HTML code from an FMF model. Likewise, other code in other languages or other models can be generated. The template creates the same HTML code, as created by the example in Figure 6 for the explicit language model in Figure 1.

Finally, there is a third option supported by Frag: As Frag combines a dynamic language with a DSL language model, the classes of the DSL language model can also be transformed using Frag's dynamic language features, such as reclassing Frag objects, changing superclasses, renaming objects/classes, and so on (see [Zdu10] for details).

None of these options is influenced by any of the other architectural decisions, and hence these options can be selected at any stage at which designing execution semantics for the DSL is necessary. Changing the decision means just rewriting the DSL execution code; no other parts

```
Object create DocumentHTMLMixin
DocumentHTMLMixin method transformText {} {
      # ...
}
DocumentHTMLMixin method mandatoryNotEmptyArguments {argList} {
      # ...
}
DocumentHTMLMixin method transformToHTML {} {
      self mandatoryNotEmptyArguments {title description author version}

      string build `
            <html>
                  <head>
                        <title>`[self get title]`</title>
                        <meta NAME="description" CONTENT="`[self get description]`">
                  </head>
                  <body>
                        <h1>`[self get title]`</h1>
                        <h3>`Author: [self get author]`</h3>
                        <h3>`Version: [self get version]`</h3>
                        `[self transformText]`
                  </body>
            </html>`
}
Document mixins DocumentHTMLMixin
```

Figure 6: DSL Execution Semantics Defined Using Interpreted Code

of the DSL are affected.

```
<html>
      <head>
            <title> <~ $doc title ~></title>
            <meta NAME="description" CONTENT="<~ $doc description ~>">
      </head>
      <body>
            <h1><~ $doc title ~></h1>
            <h3>Author: <~ $doc author ~></h3>
            <h3>Version: <~ $doc version ~></h3>
            <~ $doc text ~>
      </body>
</html>
```

Figure 7: Transformation template example

3.4 Integrating the DSL with the Host Language

The possible integration of the DSL with the host language is linked to the decision on an embedded vs. an external DSL: In an embedded DSL, per default you can use any host language feature. That is, the language extension pattern [Spi01] is used. In an external DSL, per default no host language feature can be used. To change the decision on the host language integration, without changing the decision on embedded vs. external DSL is usually some effort. Frag supports this change to a certain extent, though.

Frag provides a modular architecture, in which each host language command is bound to the interpreter as an object that is registered in the interpreter at startup time of the interpreter. A similar architecture can also be found in other modular interpreter architectures, such as the Tcl [Ous94] or Hecl [Wel10] interpreters. For interpreting an embedded language without certain parts of the Frag language, we can simply not register or de-register the problematic commands. This can go pretty far by using an embedded interpreter that virtually knows only the basic Frag parsing rules and the embedded DSL statements. However, this is also a considerable effort and, after all, we still have the Frag parsing rules in place. Hence, this alternative mainly makes sense in order to remove certain commands that are considered "dangerous" for domain experts. This approach is called a safe interpreter [LOW97], and it is well applicable for changing parts of the host language integration decision for embedded DSLs with low effort. That is, this approach uses the language specialization pattern [Spi01] to create a DSL by removing features from the host language.

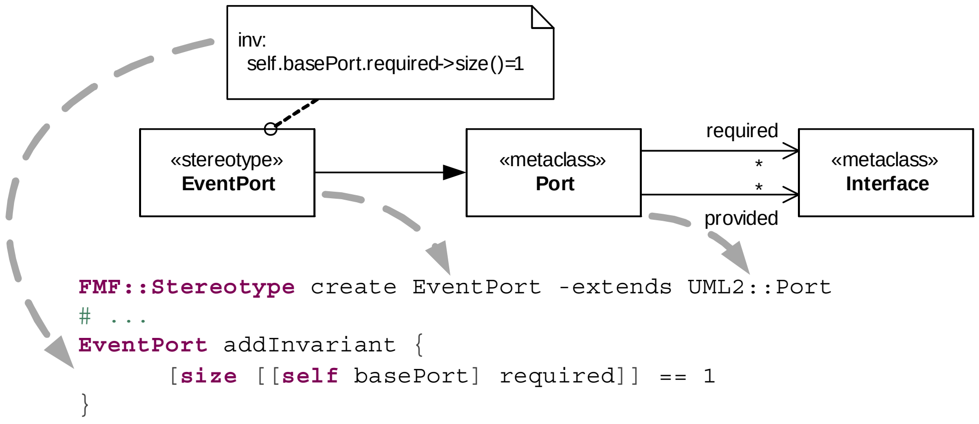
For the other case, adding host language commands to external DSLs, our approach only offers the support that can also be found in other model-driven approaches: a transformation must be used that generates or interprets the host language statement that should be added to the external DSL. For instance, if generation is used, the transformation templates explained before can be used. This realizes the piggyback pattern [Spi01]: selected host language features are provided in the DSL and will be passed untouched by the transformations. This is useful for linguistic elements such as expressions, substitutions, loops, and so on. Using the piggyback approach, these elements can be offered with small effort in an external DSL.

3.5 Static Semantics for the Language Model

Regarding the architectural decision "How should the static semantics for the language model be realized?" Frag offers a constraint language called FCL that can be used to specify OCL-style constraints for language models. As an alternative static semantics can be implemented as ordinary Frag or Java code. Also the transformation templates described before can be used.

Which of these alternatives should be chosen for realizing the static semantics of the DSL is not always obvious early during a DSL project. For example, it makes sense to use OCL-style constraints to link the DSL implementation and design. However, custom code can be more useful, if the semantics are hard to express in OCL or if the Frag/Java code should be reused in other functions of the transformation templates or in the interpreted code.

FCL is a DSL to specify OCL-style constraints in a modular fashion using the Frag syntax. Figure 8 shows an example where a `Port` meta-class is extended by an `EventPort` stereotype. We use the UML extension relationship to define the stereotype. A port can have an arbitrary number of required interfaces. The extension limits the number of required interfaces to 1, using an OCL constraint. Figure 8 shows how this simple constraint is translated to FCL. Apart from the different concrete syntaxes, FCL and OCL have very similar concepts.



```
FMF::Stereotype create EventPort -extends UML2::Port
# ...
EventPort addInvariant {
      [size [[self basePort] required]] == 1
}
```

Figure 8: FCL Example

For model-based constraints like the one above, FCL might be more appropriate, as it enables developers to translate OCL constraints of the model very easily. In contrast, some other constraints require custom code. An example, is the `mandatoryNotEmptyArguments` invocation in Figure 6. This invocation triggers a constraint that checks whether a set of arguments is existing and non-empty, otherwise a custom error message is raised. The example code is shown in Figure 9. This would be very cumbersome to express in OCL or FCL. Hence, the custom code is the better alternative in this example. The same code can also be called from or integrated in a transformation template.

As constraints are only additional code, this decision has no influence on the other architectural decisions. To change a constraint from one implementation to another only requires rewriting the constraint and making sure the constraint is triggered, when the constraints are checked.

```
DocumentHTMLMixin method mandatoryNotEmptyArguments {argList} {
      foreach arg $argList {
            if {![self varExists $arg] || [self get $arg] == ""} {
                  throw [Exception create -msg
                    "Mandatory argument '$arg' missing or empty for document: '[self]'"]
            }
      }
}
```

Figure 9: Static Semantics Constraint Implemented in Frag Code

3.6 Approach Summary

Table 2 summarizes how the architectural decisions, summarized in Table 1, are resolved using solutions for the decisions' alternatives in Frag. As we can see, Frag offers one alternative solution for each of the general alternatives summarized in Table 1. Except for the decision on the integration of the host language, none of the decisions have a dependency to other decisions that would make it to be changed later in the development process. Hence, in most cases, the only change effort that needs to be considered is the change effort required for changing the architectural decision itself. Only the decision on the integration of the host language needs to be considered together with the decision on embedded vs. external DSL, and options are presented to change it to a certain extent even if the choice goes against the default provided by an embedded or external DSL.

Frag offers support for low change effort of the decisions, too. As we can see mostly local changes to the components directly influenced by the architectural decisions are required. Only a few simple non-local changes, such as replacing field and relationship access code in the first decision in Table 2, might be required.

4 Detailed Example

In this section, we illustrate DSL-based design in Frag. We focus on the decision "Should an external or embedded DSL be developed?" in this illustration. As an external DSL in Frag is entirely based on the embedded DSL design and implementation, we usually start off with embedded DSL design and development when developing a DSL in Frag. Once the abstract syntax concepts are maturing (in a first version) and an initial test suite has been developed, we can add the external DSL artifacts – in case they are needed.

Table 2: Alternatives, Dependencies, and Change Effort Required for the Architectural Decisions in Frag

Decision	*Alternative solutions provided in Frag*	*Dependencies to other decisions*	*Change effort required*
How should the language model of the DSL be realized?	Explicit language model in FMF, implicit language model using Frag or Java objects	None	Implement new language model, search/replace of field and relationship accesses
Should an external or embedded DSL be developed?	Embedded DSL as Frag code, external DSL using Frag's parsing approach, hybrid DSL (combining the two former alternatives), using an external parser (e.g., XML parser, parser generator, ...)	None	Embedded syntax is always needed, for supporting external syntax only this syntax needs to be implemented
How should the DSL's execution semantics be defined?	Frag interpreted code, Frag transformation templates, dynamic language features	None	Only implement new DSL execution code
How should the DSL be integrated with the host language?	Forbidden (using external DSL), language extension (using embedded DSL), language specialization through removing interpreter commands, piggyback using transformations	Forbidden alternative is linked to external DSL, language extension is linked to embedded DSL	Some effort is needed to realize language specialization or piggyback alternatives, the others comes for free
How should the static semantics for the language model be realized?	Using the FCL constraint languages, using custom Frag code, using Frag transformation templates	None	Implement the new constraint and make sure it gets triggered

As an alternative, we could also have started by developing an external mockup syntax first, e.g., together with the domain experts, and hence defer the decisions on the language model design. In our experience, Frag's embedded parsing approach requires only little efforts to experiment with syntax ideas early on (i.e., before implementing the DSL). As reported in [SZ09], this has helped us in some projects in the communication with the domain experts in cases the domain concepts were highly unclear. Of course, once these syntax and language ideas mature, we need to start a language model design, because for the final DSL a close correspondence between parser and language model are necessary in any case [AP04]. In this example, however, we directly start with the language model design, as the domain concepts are roughly clear.

In this paper, we want to explore the steps for DSL-based design and development in Frag using Fowler's introductory example (see [Fow08]). This example has two main benefits: firstly, it is easy to understand, but non-trivial. Secondly, it has been realized in a number of other DSL toolkits, enabling us to compare the solutions (see Section 5). The example is about describing state machines used to implement machinery to lock and unlock secret compartments. Figure 10 shows a sample state machine presented by Fowler: Miss Grant's system. She has a secret compartment in her bedroom that is normally locked and concealed. To unlocked it for her to open, she has to close the door, open a drawer in her chest, and turn a light on. Many variations of the sequence of actions that can be carried out and the resulting behavior of the controller software exist. The example DSL should support the company, providing the controllers, so that they can install a new system with minimal effort.

To model state charts as the one in Figure 10 using the DSL, we need to be able to declare events, commands, states, and transitions between states. States contain references to actions, which should be executed when entering the state. The transitions are triggered by events, and link between a source and target state. In this example, the first step to realize this DSL in Frag is to design and implement the language model of the DSL. Figure 11 shows a language model that we have derived and adapted from Fowler's example.

We use FMF to implement the model in Frag. Figure 12 shows the language model implementation for the example (only an excerpt is shown). Various classes and associations between these classes, following the design in Figure 11, are defined. We use object names to denote identifiers for states, events, and commands. Hence, we do not need to implement the `NamedElement` class. The `isResetting` property is defined with `false` as a default value, as most events are not resetting. Apart from these two implementation details, the Frag implementation corre-

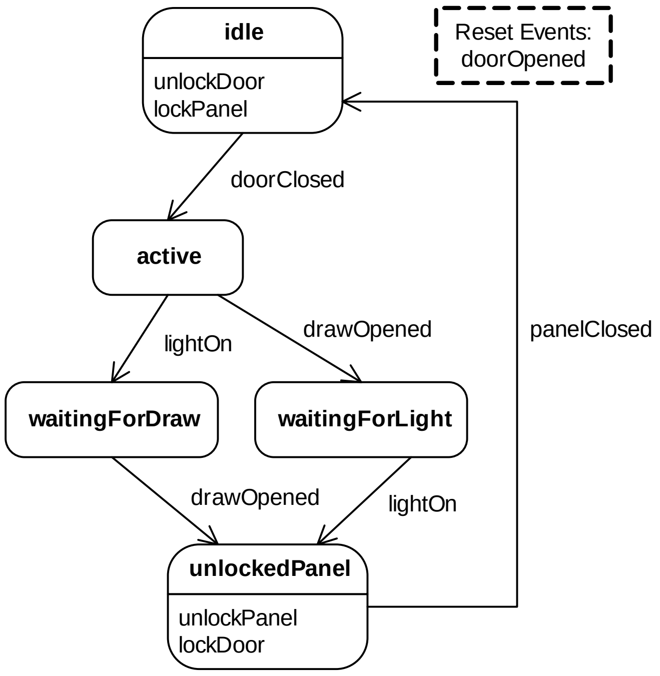


Figure 10: Fowler's example state chart (from [Fow08])

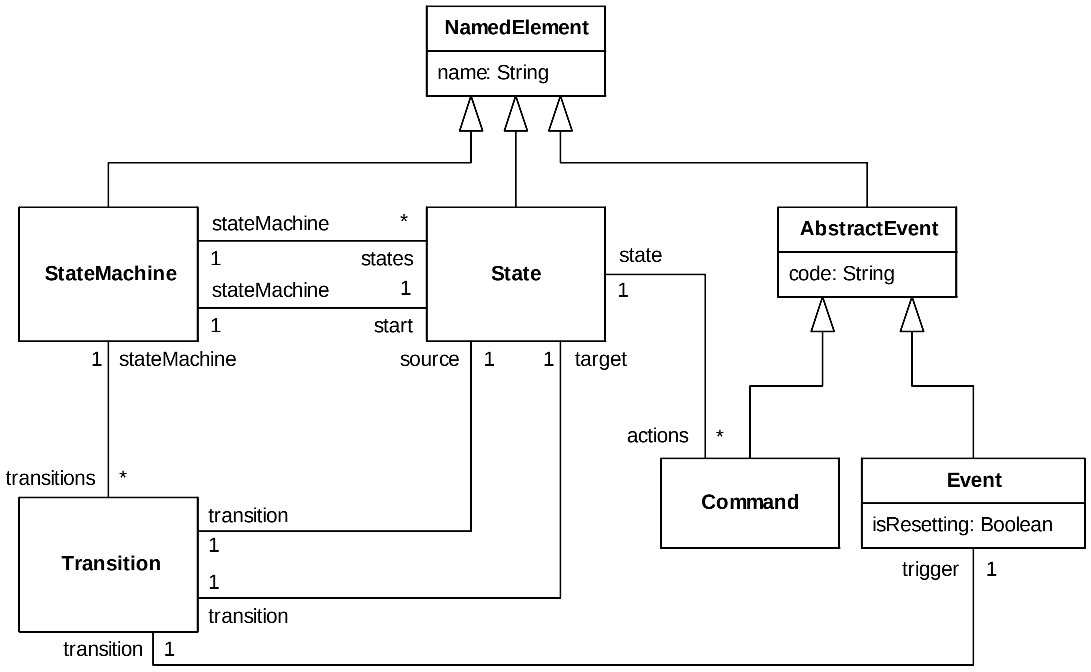


Figure 11: Language model for the state chart DSL (adapted from [Fow08])

sponds exactly to Figure 12.

```
FMF::Class create StateMachine
FMF::Class create State
FMF::Association create StateMachineStates -ends {
  {StateMachine -roleName stateMachine
    -multiplicity 1 -navigable true}
  {State -roleName states
    -multiplicity * -navigable true}
}
FMF::Association create StateMachineStart -ends {
  {StateMachine -roleName stateMachine
    -multiplicity 1 -navigable true}
  {State -roleName start
    -multiplicity 1 -navigable true}
}
FMF::Class create Transition
...
```

Figure 12: Example language model (excerpt)

Once a language model is defined, we can use the Frag syntax to create instances of the model. That is, by defining the language model, we automatically have a rich embedded Frag syntax that can be used. This syntax is the same syntax as for defining the language model. Figure 13 shows an example in which we create a model instance that implements the state chart from Figure 11.

This example uses the Frag syntax in a pretty plain way. Actually, only two language features of Frag that are not derived from the language model are used. Firstly, we use `list build` to pass lists to the state machine instance. Secondly, we use unnamed objects for transitions. Both features are used to simplify the code and do not have to be used. However, we can also go much further, and use all kinds of language constructs defined in Frag. For instance, the code in Figure 14 shows a `foreach` loop doing the same as the `Command` instantiations in Figure 13. This syntax would be useful, if some additional tasks must be performed for each `Command` that is instantiated.

The two examples show the benefits and liabilities of embedded DSL syntaxes. As a benefit, we can use the Frag language features for simplifying and shortening the code. We can use features, such as control structures and substitutions, to ease (recurring) tasks. Software developers will appreciate such features being available, as they allow them to automate tasks and make the code more readable. However, as a liability the user of the DSL must understand the Frag

```
Event create doorClosed -code D1CL
Event create drawOpened -code D2OP
Event create lightOn -code L1ON
Event create doorOpened -code D1OP -isResetting true
Event create panelClosed -code PNCL

Command create unlockPanel -code PNUL
Command create lockPanel -code PNLK
Command create lockDoor -code D1LK
Command create unlockDoor -code D1UL

StateMachine create MissGrantsSystem -states [list build
  [State create idle -actions {unlockDoor lockPanel}]
  [State create active]
  [State create waitingForLight]
  [State create waitingForDraw]
  [State create unlockedPanel
    -actions {unlockPanel lockDoor}]
] -transitions [list build
  [Transition create -source idle -target active
    -trigger doorClosed]
  [Transition create -source active
    -target waitingForLight -trigger drawOpened]
  [Transition create -source active
    -target waitingForDraw -trigger lightOn]
  [Transition create -source waitingForLight
    -target unlockedPanel -trigger lightOn]
  [Transition create -source waitingForDraw
    -target unlockedPanel -trigger drawOpened]
  [Transition create -source unlockedPanel
    -target idle -trigger panelClosed]
] -start idle
```

Figure 13: Example of the embedded DSL syntax

features. Even if users use only a limited subset of Frag, they must be aware that they are using an embedded DSL. For non-technical users there is the danger that they accidentally use Frag features and/or receive (for them) hard to understand error messages.

The important point to stress here is that in Frag the toolkit does not make a decision on the use of Frag language elements or the usability of error messages. As a combination of all options is provided, the architect can decide for the most appropriate choices for these decisions.

```
foreach {cmdName code} {
      unlockPanel PNUL
      lockPanel PNLK
      lockDoor D1LK
      unlockDoor D1UL
} {
      Command create $cmdName -code $code
}
```

Figure 14: Using a `foreach` loop in the embedded DSL syntax

One benefit of the Frag approach is that we can first start-off by defining an embedded DSL, which is useful for developing, testing, and evolving the language model, and then add the external DSL artifacts. This architectural decision can be deferred until the first DSL prototype has been realized. Consider we want to make the example DSL usable for non-technical stakeholders later on in our project. Fowler suggests the external syntax for the DSL that is shown in Figure 15.

```
events
  doorClosed  D1CL
  drawOpened  D2OP
  lightOn     L1ON
  doorOpened  D1OP
  panelClosed PNCL
end
resetEvents
  doorOpened
end
commands
  unlockPanel PNUL
  lockPanel   PNLK
  lockDoor    D1LK
  unlockDoor  D1UL
end
state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end
state active
  drawOpened => waitingForLight
  lightOn    => waitingForDraw
end
state waitingForLight
  lightOn => unlockedPanel
end
state waitingForDraw
  drawOpened => unlockedPanel
end
state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

Figure 15: External DSL syntax (from [Fow08])

To use this syntax for our example DSL, we must define a lexical scanner and mapping specification. The syntax of the scanner specification is shown in Figure 16. It ignores whitespaces (WS). A WORD is a repetition of something that is not a whitespace or curly braces. Three kinds of elements are added to the scanned elements list using rules: non-nested curly braces blocks consisting of words or whitespaces with the token BLOCK, arrows with the token ARROW, and words with the token WORD.

```
WS = " " | "\r\n" | "\n" | "\t";
WORD = (! (WS | "}" | "{"))[1..*];

%- WS;
% ("{" (WORD | WS)[*] "}") "BLOCK";
% "=>" "ARROW";
% WORD "WORD";
```

Figure 16: Syntax specification of the external DSL

The main mapping defined in Figure 17 accepts any repetition of the four block types (events, resetEvents, commands, and state) used in Fowlers example. For each of the block types, the mapping first requires the keyword, then any number of elements that are not “end”, and finally the element “end”. Each block’s contents are assembled in a variable `body`.

The interpretation of the blocks is handled by specialized mappings, such as the one shown in Figure 18, triggered in the `switch` statement (depending on the block’s keyword). For example, the mapping specification in Figure 18 defines how resetting events blocks are handled. Basically these blocks consist of a sequence of elements, and each element must be an event. If an element is not an event, we raise a custom error message. Otherwise we modify the event object to be a resetting event. All other block mappings similarly provide mappings of the scanned elements to the language model.

In this example we can see some of the main benefits of external DSLs in Frag. We can provide a custom syntax, check all elements semantically either during the mapping or in later stages of working with the models, and provide custom error messages that are useful for the DSL user. Hence, we can provide a much saver and easier way to use DSLs for non-technical stakeholders than with the embedded DSL approach.

The benefit of Frag’s parsing and mapping approach in this example is that it is easy to change scanner rules and mappings, we define the grammar and mapping inside the Frag interpreter without need for further tools, custom error messages can easily be embedded, and mappings

```
DSL::DSLMapping create MainMapping -mapping {
  @rep * {
    @seq {
      @elt id {$id == "events" || $id == "resetEvents" ||
               $id == "commands" || $id == "state"}
      @rep * {
        @elt e {$e != "end"} {append body $e}
      } {set body ""} {
        switch $id {
          events {
            KeyCodeBlockMapping map $body "set class Event"
          }
          resetEvents {
            ResetEventsBlockMapping map $body
          }
          commands {
            KeyCodeBlockMapping map $body "set class Command"
          }
          state {
            set transitionCode [string append $transitionCode "\n"
              [StateBlockMapping map $body "set smObj $smObj"]]
          }
        }
      }
      @elt end {$end == "end"}
    }
  } {set transitionCode ""} {
    eval $transitionCode
  }
}
```

Figure 17: Main DSL mapping for the external DSL

```
DSL::DSLMapping create ResetEventsBlockMapping -mapping {
  @rep * {
    @seq {
      @elt event {} {
        if {![interp isObject $event] || ![$event isType Event]} {
          throw [ErrorException create -msg "object $event is not an event"]
        }
        $event isResetting true
      }
    }
  }
}
```

Figure 18: DSL mapping for resetting events block

can be directed to different language models. This flexibility enables us to defer the related decisions until the design of the external syntax.

5 Comparison to Related Work

5.1 Related Work on DSL Toolkits

In the literature, we find a number of other implementations of Fowler's example. Fowler himself present 3 realizations [Fow08]: a Java API implementation, an external XML syntax, and an external syntax implemented using the ANTLR parser generator. All 3 realizations completely exclude many architectural decision options as they support only a part of the DSL-based design and development tasks. All architectural decision options that are supported are preselected: Only an external syntax or the Java-API is supported, and all 3 realizations support only a limited version of an explicit language model.

Efftinge presents an implementation of Fowler's example using XText [Eff08]. XText is a model-driven DSL approach that introduces a grammar language based on ANTLR. It is part of the openArchitectureWare generator. In contrast to ANTLR, it offers rather limited means for syntax specification (more sophisticated parse rules must be specified as "native" ANTLR rules). But within these limits many custom syntaxes can be defined. That is, XText supports only a limited number of external syntaxes well. XText supports an explicit language model, defined using EMF (this decision option is also preselected). A unique benefit of XText, due to its rather limited grammar language, is that it can generate the language model from the grammar specification. XText provides languages for defining constraints and custom error messages; here, the architect is free to choose.

The AMMA (Atlas Model Management Architecture) platform supports the model-driven definition of DSLs [JBK06]. Its architecture is similar to XText and openArchitectureWare, but some details differ. For building DSLs, a meta-model based on the KM3 (the Kernel Meta-meta-model) must be built. In contrast to XText, concrete syntaxes and DSL semantics can be represented either as models or transformations. TCS (Textual Concrete Syntax) is provided as a DSL that is used to specify the concrete syntax of DSLs. TCS and XText are the two initial contributions for TMF, the Eclipse Textual Modeling Framework [Ecl10].

JetBrains MPS system [Dmi04] provides a another similar approach to XText. A main difference is that the concrete syntax is based on forms that are filled with the MPS tool. The language model is used to define the content of the forms. These decisions are preselected by the tool and cannot be changed.

Corneliussen [Cor08] illustrates how to implement Fowler's example using Microsoft's new modeling platform, code-named Oslo [Mic08]. Oslo uses MSchema to define schemas for data instances (this is more comparable to XSD and DTD rather than an EMF model). MGraph is an exchange format for capturing concrete data instances. MGrammar is a grammar language that transforms a textual DSL to MGraph. To use this output, a programmatic mapping (e.g., using C#) is needed. The error messages are standard errors by the parser. That is, many major architectural decision options are preselected by the technology.

In a dynamic language, such as Ruby, developing embedded DSLs is supported by some of the language's constructs [Fre06]. This approach is pretty comparable to developing embedded DSLs in object-oriented, functional languages such as Scala [Dub06]. The integration with the host language is hence quite good. The depth of host language integration can be selected by the architect, but it is no option to exclude the use of the host language entirely. Usually DSL statements work on Ruby objects directly, without an explicit language model. Like the model-driven approaches, Frag supports building an explicit language model for the DSL, which eases DSL modeling and understanding. As a dynamic language, Frag can also support building DSLs without an explicit language model, as in Ruby DSLs. The grammar and the mapping to Ruby is defined in a purely programmatic fashion; other options are possible but require hand-crafting. A few choices are left open for the architect: For instance, custom error messages can be placed anywhere in the Ruby code. Many options for integrating different Ruby DSLs exist. Frag support the options of dynamic language DSLs plus language model options and the extensible syntax of external DSL approaches. In a dynamic language based DSL, the syntax can be changed only as far as possible in the host language.

In comparison to Frag, all other approaches focus either on embedded or external DSLs. Frag is the only approach that supports both approaches. Converge's approach [Tra08a, Tra08b] is to provide DSL blocks that contain DSL code. Hence, Converge combines embedded or external DSLs, too. The DSL code is written in a different syntax than the host language. The DSL syntax is specified using a declarative grammar language. Custom error messages can be defined. In Converge, however, not both approaches can be used, but DSLs defined using the specification techniques for external DSLs are embedded in a host language using a construct called DSL blocks. Ghosh demonstrates that Scala's parser combinators can be used to realize a similar solution for external DSL grammars [Gho08], but Scala does not support the DSL block feature.

One of the big benefits of the Frag approach is that it provides host language integration in both cases, external and embedded DSLs. None of the other external DSL approaches support both architectural decision options. That is, if programmatic tasks are needed in the DSL development or use, such as scripting or checking DSL code, we cannot use the same syntax for this code as for defining the DSL, but must use a different language such as the host language.

For specifying the semantics of the DSL, the other approaches either use custom code in case of the language-oriented approaches, or transformations (execution semantics) and constraint languages (static semantics) in case of the model-driven approaches (which sometimes can trigger custom code). Hence, the choices for specifying execution and static semantics are also made by the DSL tool chosen and are rather hard to change. In contrast, Frag offers both choices for both kinds of semantics.

Custom error messages are very important for defining DSLs for end users, as they should not have to deal with a parser's output or similar cryptic error messages. Reuse by non-technical users is hardly possible without error messages that are meaningful from a domain-oriented point of view. Most approaches support custom error messages: Some are easier to use than others, tough. In contrast to Frag, in some approaches the choices where a custom error message can be triggered are more limited.

In conclusion, we can assess that Frag provides a unique combination of the most important benefits of a number of DSL toolkits. In contrast to the existing DSL toolkits it leaves many architectural decisions in DSL-based design open as long as possible by supporting a particular combination of most of the options provided by the other approaches.

5.2 Related Work on Reusable Knowledge in DSL Development

While the related work presented in the previous section is directly related to the approach proposed in this article, a number of related approaches focus on reusable design and implementation knowledge on DSLs. Hence, these approaches are related to the notion of architectural decisions on DSLs, proposed in this article.

Several authors introduced patterns and pattern languages that can be applied in DSL development. This includes patterns for the design and implementation of DSLs [Spi01] and patterns for evolving frameworks into DSLs [RJ96, RJ97]. A pattern is a time-proven solution to a recurring design problem. The patterns do not only describe how a DSL is developed, but also why it is developed in a specific way. The patterns for the design and implementation of DSLs by Spinellis [Spi01] have been used by Mernik et al. [MHS05], who provide a survey of decision factors for the decision, analysis, design, and implementation phases of DSL development. These decision factors can be considered during the DSL development.

The alternatives of our architectural decisions can be seen as (candidate) patterns for DSL design. Our performance and scalability measures, presented in Section 6, provided some reasoning for making informed architectural decisions. The patterns as such, however, provide no support for deferring the decision for or against them.

Similar to patterns, lessons learned have been used as a vehicle to convey best practices of DSL development. For example, Wile reports on twelve lessons learned from three DSL experiments [Wil04]. For each lesson he introduces a respective rule of thumb and gives an overview of the experiences that are the origin of the corresponding rule. Luoma et al. [LKT04] conducted a study including 23 industrial projects for the definition of domain-specific (graphical) modeling languages. A number of DSLs are systematically compared. These approaches have in common with our approach that they are based on DSL project experiences (see Section 6). Like the patterns, the lessons learned describe specific design considerations during DSL design and are hence comparable to the architectural decisions in our work. Again, they do not focus on deferring decisions though.

Both the patterns and lessons learned on DSLs do not provide tool or language support as provided in our approach.

One contribution of our work is that it makes different options for the architectural decisions on DSLs comparable within one framework. For example, in Section 6 we compare different approaches to DSL implementation, such as external vs. embedded DSLs, in the Frag framework. Only a few studies exist so far that systematically compare different options for the DSL implementation. Kosar et al. [KLBM08], for instance, compare 10 DSL implementation approaches with regard to implementation effort and end-user effort. In contrast to our work, Kosar et al. focus on technical options for DSL implementation, such as source-to-source, macro processing, embedded, interpreter, and compiler generator. In this context, they also provide quantitative evidence for design knowledge on the decision for external vs. embedded DSLs that can be generalized and reused in other contexts.

6 Experiences and Performance/Scalability Evaluation

We and others have applied our DSL approach and the Frag prototypes in a number of projects. These experiences are summarized in Table 3. The table includes the three DSLs, we built to realize our approach, that have been introduced in this paper: the Frag mapping DSL, the Frag rule-based parsing DSL, and the Frag templating DSL. In addition, a number of other DSL projects in which the author was involved are listed, along with the author's roles in these projects. In general, these experiences all supported the claims made in this paper. For example, in most of the external DSLs we were able to start-off with an internal syntax (that is, we built an embedded DSL as a prototype first), and then revised this decision later in time.

For the three Frag internal DSLs, as well as for Frag itself, we provide tool support in form of Eclipse Editor plugins, supporting syntax highlighting, launching, error markers, outline views, preferences, and so on. It is our plan to develop a product line for automatically deriving editors for all our Frag DSLs.

An important aspect is: at what price come the benefits of our approach? In general, the approach mainly has an impact on the performance and the scalability of systems using it. To be able to follow a systematic approach to analyze the performance and scalability impacts of the various parts of our approach, we analyzed the smaller DSLs from Table 3 to understand how a realistic small-scale setting of the elements (model elements, model instances, external syntax elements, and template elements) looks like. Next, we scaled the setting up to more elements using the factors 1, 2, 5, 10, 50, and 100. We measured the performance for each setting with each scale factor to analyze how scalable the approach is. We measured the performance on a rather weak desktop machine, as our approach will usually need to run on the local machine of developers. The machine had an Intel Core2 Duo CPU, 1.60 GHz processor with 1.96 GB RAM and was running under Windows XP. We used the Java JRE 1.6.0_07 and Frag 0.8 running on it. We have run each measurement 100 times and calculated both the average in microseconds as well as the standard deviation of the results. The results are shown in Table 4. To have a baseline for assessing the scalability, we compared to a hypothetical linear scaling, calculated

Table 3: Overview of DSL Projects

Domain	*Technologies Used*	*DSL Type*	*Target*	*Project Type*	*Our Role(s)*	*Project Duration*
Bibliography management	Frag, HTML, Latex	Embedded, External	Code generation	Research project	DSL Designer, Developer	3 month
DSL editor specification	Eclipse, Java, Frag	Embedded	DSL Interpretation	Research project	Architect, Observer	1 year
Frag mapping DSL	Java, Frag	Embedded	DSL Interpretation	Research project	DSL Designer, Developer	2 years
Frag rule-based parsing DSL	Java, Frag	Embedded, External	DSL Interpretation	Research project	DSL Designer, Developer	2 years
Frag templating DSL	Java, Frag	External	DSL Interpretation	Research project	DSL Designer, Developer	1 year
Multimedia home platform	Frag, Java MHP platform	Embedded	DSL Interpretation	Industry project prototype	DSL Designer, Developer, Observer	1 year
QoS specification	Frag, Apache CXF, Java	Embedded, External	DSL Interpretation	Case study	Architect, Observer	1 year
Software architectural knowledge	Frag, HTML, Latex	Embedded	Code generation	Research project	DSL Designer, Developer	3 month
Software documentation	Frag, HTML, Latex	Embedded, External	Code generation	Research project	DSL Designer, Developer	3 month

by multiplying the average result for factor 1 with the higher factors. This comparison is shown in Figures 19 and 20.

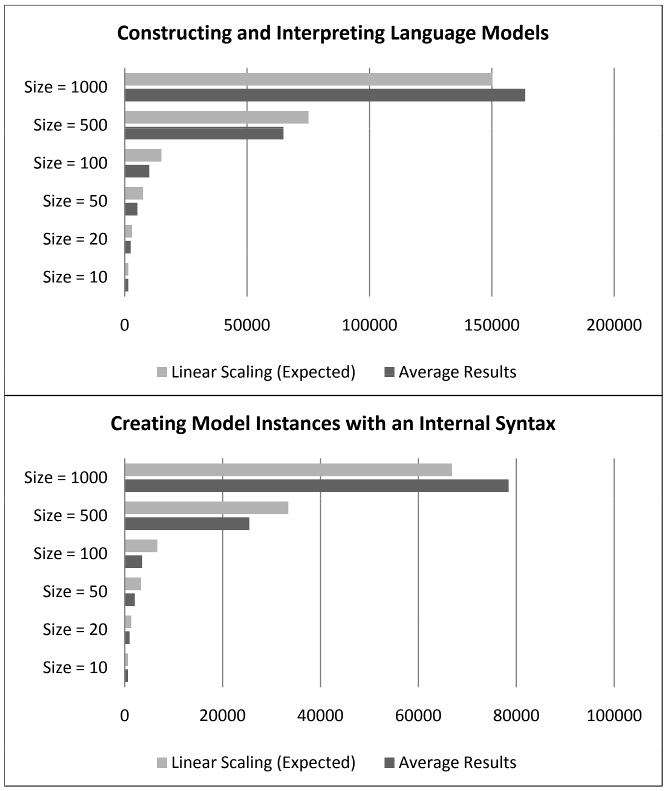


Figure 19: Comparing the Average Results to Expected Linear Scaling Result (1)

We first measured the performance and scalability of constructing and interpreting language models. As the language models in Frag have been expressed in terms of special programming language objects, the numbers should be pretty comparable to other dynamic languages running in virtual machines (if they would implement a modeling framework like FMF, too). Hence, these numbers can serve as a baseline for comparing to the other performance measures that are specific to our approach. As can be seen, in the first block of Table 4, we have measured the performance for language models of various sizes. The smallest interpreted language model had 10 elements: 4 classes, 2 inheritance relationships, 1 aggregation relationship, and 3 fields. This distribution of model elements was typical in some smaller DSL examples we have analyzed. Using the factors above, we have scaled up the measurements to 1000 language model elements of the same distribution. As can be seen, even very large language models with 1000 elements can be created in less than 0,17 seconds, meaning that the performance is acceptable for typical use cases such as MDD. Only for high-performance use cases needing for example the models at runtime and very large models, the performance can be unacceptable. The average performance scales almost linearly (see Figures 19 and 20).

However, for small models there is a rather great standard deviation, which gets better for larger models, but is still high. We think this high standard deviation is mainly due to running on a virtual machine and garbage collection of Java and Frag. We see similar standard deviations for all other measurements as well. We think this is a general drawback of running our approach on virtual machine architectures.

Secondly, we measured creating model instances with an internal syntax. We used the language model of a small DSL with 44 language model elements as the foundation for these measurements. We created an model instance with 1 instance object, 8 fields, and 1 relationship. Again, this was a typical distribution from our DSL experiences. This measurement was also scaled up to a 1000 elements. It showed a similar scaling behavior as in the first measurement: first a little better than linear scaling and then getting a little worse than linear scaling. We think that both the performance and scalability is acceptable for most typical use cases as well.

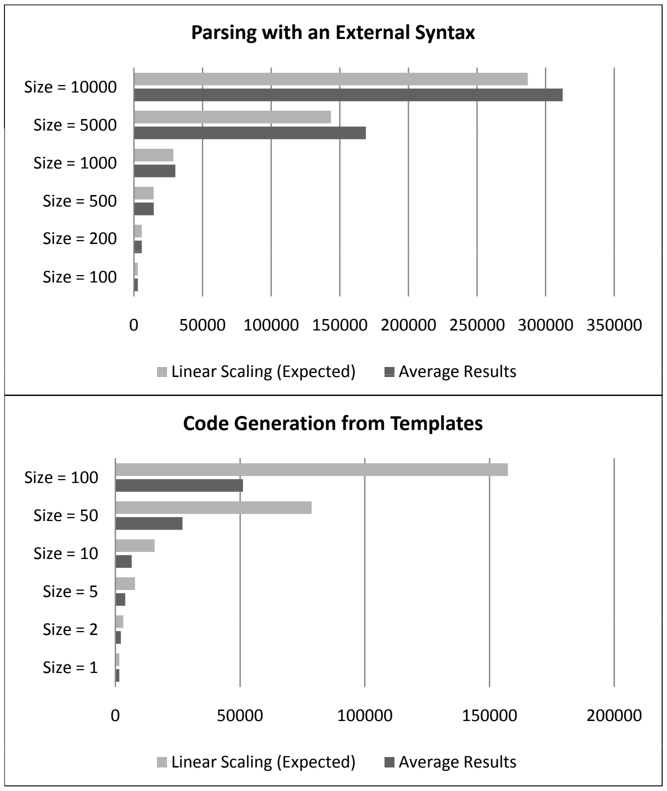


Figure 20: Comparing the Average Results to Expected Linear Scaling Result (2)

Thirdly, we measured parsing with an external syntax. In this case we used a larger starting size of 100 language elements, as usually more than 1 language element is used to depict a concept in a DSL. We used 100 language elements to start off and scaled up to 10000 language elements. Again, we used the language model with 44 model elements. We used a small scanner specification with the syntax specification shown in Figure 16 (without the arrow rule). A mapping specification with 10 mapping elements was used. A simple syntax that is mapping the language elements one-to-one to the language model was used. As shown in Table 4, the distribution of results is roughly comparable to the previous measurements and hence acceptable as well. However, it is important to note that this measurement has some open variables: an overly complex grammar or mapping specification can easily impede the performance significantly. In addition, the performance for parsing an external syntax must be added to the creation of model instances. That is, the penalty of an external syntax is very roughly leading to a performance decrease by the factor 2 (or even more). This can be a problem for very large instance models.

Fourthly, we measured the times for template generation. It must be noted that these are just very early results, as our template engine is still in an early stage of development. We have measured the performance of the instance model, explained before, with 10 instances, 80 fields, and 10 relationships. We generated from templates with 1 to 100 template elements (template directive to generate code). Normally, templates only have a few directives, hence we focused on these smaller template element sizes. As can be seen the template engine scales much better than linearly. We think this is due to the initialization time needed for setting up the template engine to parse and map the template code, before the actual transformation of the language model instances can happen. This initialization time requires a constant overhead that is more significant for templates with only one or a few template elements. But still the performance for these small element sizes is acceptable. Please note that template generation is also – like an external syntax – an additional cost. Often more than one template is needed to generate all code from a set of language model elements. Hence, templating can have a significant performance impact. However, templating usually happens at design time. The alternative, interpretation of the DSL code, has an performance impact on the system while it runs.

In summary, even though these numbers are specific to our Frag implementation, they show that it is possible to create an implementation of a DSL toolkit that allows us to defer architectural decisions of DSL design, even on a virtual machine architecture, with acceptable performance and scalability impacts. For a very few use cases, such as very large models at runtime, the performance impacts might not be acceptable, though. However, surely our prototype implementation can be improved and/or optimized for such cases, if needed. An important lesson learned is that it is important to measure the performance impact and scalability when making architectural decisions about DSL design, as the combination of decisions can lead to significant performance or scalability problems. The numbers presented in this section are hence an important contribution to making these architectural decisions, as they give rough estimates for the performance and scalability impacts that can be expect. They can also be used to direct optimization efforts.

7 Conclusion

In this paper we have introduced an approach to DSL-based design that combines ideas from various related works in a unique way. Our main goal was to overcome some liabilities directly or indirectly linked to implicitly making architectural decisions very early in a DSL project as a consequence of selecting a DSL toolkit. Frag supports deferring these architectural decisions by the following means: it provides a combination of the dynamic programming language approach to DSL-based design and the modeling framework approach known from model-driven language workbenches; it provides a flexible rule-based, embedded parsing approach; it support transformation, generation, and interpretation as options to define the execution semantics of DSLs; it makes all DSL artifacts modular entities that can be selected or deselected at any stage during DSL-based design. This way we can support deferring the architectural decisions in DSL-based design till after the domain abstractions have been sufficiently understood. Also, many decisions can be changed after they have been made once (e.g., an external DSL can be added later on), without having to change the DSL toolkit and with rather low change impact. Our insights are not limited to the Frag toolkit, though. By following our approach or combining parts of our approach with other DSL toolkits, with foreseeable effort, other DSL toolkits can be modified or adapted to defer specific architectural decisions. Using a number of DSL project experiences and systematic performance and scalability measures we have shown that the approach is feasible in practice.

Table 4: Performance and Scalability Measurements

Constructing and Interpreting Language Models						
Size	*10*	*20*	*50*	*100*	*500*	*1000*
Average (microseconds)	1502	2491	5219	10054	64871	163598
Standard Deviation	633	865	1541	2735	9211	14337
Creating Model Instances with an Internal Syntax						
Size	*10*	*20*	*50*	*100*	*500*	*1000*
Average (microseconds)	668	1015	2055	3562	25471	78430
Standard Deviation	345	417	715	861	4897	9109
Parsing with an External Syntax						
Size	*100*	*200*	*500*	*1000*	*5000*	*10000*
Average (microseconds)	2870	5685	14468	30207	169053	312452
Standard Deviation	904	1126	3014	6303	6144	12700
Code Generation from Templates						
Size	*1*	*2*	*5*	*10*	*50*	*100*
Average (microseconds)	1574	2175	3936	6534	26868	51137
Standard Deviation	662	864	1210	2059	4456	6850

Acknowledgments

This work was partly supported by the European Union FP7 project COMPAS, grant no. 215175. http://www.compas-ict.eu/.

References

[AP04] M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. TUCS Technical Report No 606, http://tucs.fi/publications/insight.php?id=tAlPo04a, 2004.

[Ben86] J. Bentley. Programming Pearls – Little Languages. *Communications of the ACM*, 29(8):711 – 721, August 1986.

[CB74] D.D. Chamberlin and R.F. Boyce. Sequel: A structured english query language. In *Proc. of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 249–264, May 1974.

[Cor08] L. Corneliussen. Fowler's DSL example with MGrammar. http://startbigthinksmall.wordpress.com/2008/11/26/fowlers-dsl-example-with-mgrammar-draft/, 2008.

[Dmi04] S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/, November 2004.

[Dub06] G. Dubochet. On embedding domain-specific languages with user-friendly syntax. In *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development*, pages 19–22, Nantes, France, July 2006.

[Ecl10] Eclipse. Textual Modeling Framework. http://www.eclipse.org/modeling/tmf/, 2010.

[Eff08] S. Efftinge. XText – Documentation. http://wiki.eclipse.org/Xtext/Documentation, 2008.

[Fow05] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html, June 2005.

[Fow08] M. Fowler. Domain Specific Languages – An Introductory Example. http://martinfowler.com/dslwip/Intro.html, 2008.

[Fre06] J. Freeze. Creating DSLs with Ruby. Ruby Code & Style, Artima, http://www.artima.com/rubycs/articles/ruby_as_dsl.html, March 2006.

[Gho08] D. Ghosh. External DSLs made easy with Scala Parser Combinators. http://debasishg.blogspot.com/2008/04/external-dsls-made-easy-with-scala.html, 2008.

[Gra93] P. Graham. *On Lisp – Advanced Techniques for Common Lisp*. Prentice Hall, 1993.

[GS04] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. J. Wiley and Sons Ltd., 2004.

[HB88] R.M. Herndon and V.A. Berzins. The Realizable Benefits of a Language Prototyping Language. *IEEE Transactions on Software Engineering (TSE)*, 14(6):803–809, June 1988.

[Hud96] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28:196, December 1996.

[ISO96] Information technology – Syntactic metalanguage – Extended BNF – ISO/IEC 14977:1996. http://www.iso.org/iso/iso_catalogue/catalogue_tc*ash*catalogue_detail.htm?csnumber=26153, 1996.

[ISO03] Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework) – ISO/IEC 9075-1:2003. http://www.iso.org/iso/iso_catalogue/catalogue_tc*ash*catalogue_detail.htm?csnumber=34132, 2003.

[JB05] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society.

[JBK06] F. Jouault, J. Bézivin, and I. Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th* *international conference on Generative programming and component engineering*, pages 249–254, New York, NY, USA, 2006. ACM.

[KLBM08] T. Kosar, P.E. Martinez López, P.A. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[KT08] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[Lam94] L. Lamport. *LaTeX: A Document Preparation System (2nd Edition)*. Addison-Wesley, 1994.

[Lat08] LaTeX Project Homepage. http://www.latex-project.org, 2008.

[LKT04] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. In *Proc. of the 4th OOPSLA Workshop on Domain-Specific Modeling*, October 2004.

[LOW97] J.Y Levy, , J.K. Ousterhout, and B.B. Welch. The safe-tcl security model. Technical report, Mountain View, CA, USA, 1997.

[MHS05] M. Mernik, J. Heering, and A.M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[Mic08] Microsoft. Microsoft Modeling Platform (code named Oslo). http://msdn.microsoft.com/en-us/library/cc709420.aspx, 2008.

[MZ05] M. Mernik and V. Zumer. Incremental programming language development. *Computer Languages, Systems & Structures*, 31(1):1–16, 2005.

[Ope08] Open Architecture Ware. openArchitectureWare. http://www.openarchitectureware.org/, 2008.

[Ous94] J. K. Ousterhout. *Tcl and Tk*. Addison-Wesley, 1994.

[PAC+02] S. Pemberton, D. Austin, T. Celik, D. Dominiak, H. Elenbaas, B. Epperson, M. Ishikawa, S. Matsui, S. McCarron, A. Navarro, S. Peruvemba, R. Relyea, S. Schnitzenbaumer, and P. Stark. XHTML 1.0 The Extensible HyperText

Markup Language (Second Edition) – A Reformulation of HTML 4 in XML 1.0. http://www.w3.org/TR/xhtml1, August 2002.

[Par07] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[Ray03] E.S. Raymond. *The Art of Unix Programming*. Addison-Wesley, 2003.

[RJ96] D. Roberts and R. Johnson. Evolve Frameworks into Domain-Specific Languages. In *Proc. of the 3rd Pattern Languages of Programs Conference (PLoP)*, Washington University Technical Report (wucs-97-07), Allerton Park, Illinois, September 1996.

[RJ97] D. Roberts and R. Johnson. Patterns for Evolving Frameworks. In *Pattern Languages of Program Design 3*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[Sch06] D.C. Schmidt. Model-Driven Engineering – Guest Editor's Introduction. *Computer*, 39(2):25, February 2006.

[Sel03] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[Spi01] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development*. John Wiley & Sons, 2006.

[SZ09] M. Strembeck and U. Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience (SP&E)*, 39(15):1253–1292, 2009.

[Tra08a] L. Tratt. Domain specific language implementation via compile-time metaprogramming. *TOPLAS*, 30(6):1–40, 2008.

[Tra08b] L. Tratt. Evolving a DSL implementation. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *LNCS*, pages 425–441, 2008.

[Wel10] D. Welton. Hecl – The Mobile Scripting Language. http://www.hecl.org/, 2010.

[Wil04] D. Wile. Lessons learned from real DSL experiments. *Science of Computer Programming*, 51(3):265–290, June 2004.

[Zdu10] U. Zdun. Frag. http://frag.sourceforge.net/, 2010.