A Chrestomathy of DSL Implementations

Simon Schauss Ralf Lämmel Johannes Härtel
Marcel Heinz Kevin Klein Lukas Härtel
Software Languages Team, University of Koblenz-Landau
Germany

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

Abstract

Selecting and properly using approaches for DSL implementation can be challenging, given their variety and complexity. To support developers, we present the *software chrestomathy* MetaLib, a well-organized and well-documented collection of DSL implementations useful for learning. We focus on basic metaprogramming techniques for implementing DSL syntax and semantics. The DSL implementations are organized and enhanced by feature modeling, semantic annotation, and model-based documentation. The chrestomathy enables side-by-side exploration of different implementation approaches for DSLs. Source code, feature model, feature configurations, semantic annotations, and documentation are publicly available online, explorable through a web application, and maintained by a collaborative process.

CCS Concepts • **Software and its engineering** → *Abstraction, modeling and modularity*; *Syntax*; *Semantics*; Software libraries and repositories;

Keywords DSL implementation. Metaprogramming. Software chrestomathy. Learning. MetaLib. Feature modeling. Model-based documentation.

ACM Reference Format:
Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel, and Thorsten Berger. 2017. A Chrestomathy of DSL Implementations. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136038

1 Introduction

Research context: Learning DSL implementation Over the last decades, many different languages and technologies for the implementation of DSLs have been introduced. We view DSL implementation here as a metaprogramming scenario. Developers—as well as students or course staff—who need to learn metaprogramming or understand the domain overall, are confronted with a myriad of often complex options [11, 12, 28, 31]. Indeed, each relevant language and technology for DSL implementation comes with its specifics and idiosyncrasies.

Our research is based on the hypothesis that learning in this domain can be supported by a so-called *software chrestomathy*, which generally refers to a collection of programs or systems useful for learning (by definition) [14, 24]. We describe the chrestomathy MetaLib as a collection of DSL implementations that exercise metaprogramming in an illustrative manner, while covering a space of options, and using structured and explorable documentation for the benefit of learners. All collected DSL implementations implement the same DSL—a language for finite state machines (FSML).

Research scope: Basics of DSL implementation We focus on metaprogramming techniques for implementing the syntax and semantics of a DSL. At this point, we are not concerned with IDE-oriented aspects [12]. We are also not concerned with 'secondary' language implementation aspects, such as components for software reverse engineering, metrics, re-engineering (e.g., refactoring), and software composition. The scope could be possibly extended in future work. We do not limit ourselves to designated metaprogramming systems or language workbenches, such as Rascal [21] or MPS [38]; we also consider general purpose programming languages, possibly with appropriate extensions or libraries, such as Java and Python with ANTLR [30], and Haskell with template metaprogramming [34] and quasi-quotation [27]. Section 2 describes the sampling of approaches we study.

Contributions of the paper The MetaLib chrestomathy comprises a well-organized and well-documented collection of DSL implementations. We organize the implementations and underlying metaprogramming approaches in terms of features—important design options that DSL developers need to take into account. We organize the identified features in a feature model [6, 18], an intuitive, tree-like notation commonly used to describe the optional and mandatory features of a software product line [2]. The DSL implementations are described with the help of a model-based documentation approach that integrates feature tagging, explanatory portions (headlines or captions), and semantic annotations for the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136038

used languages, technologies, and concepts of metaprogramming. Such a documentation directly enables exploration by learners.

To systematically create the chrestomathy, we started with a basic feature model of DSL implementations (inspired by existing literature) and a sampling of different metaprogramming approaches. We then implemented our language—FSML—using each approach by following best practices. We analyzed each implementation, thereby refining our feature model, tagging the realization of the features in code, and recording important design decisions and experiences.

Our work is publicly available online as the MetaLib chrestomathy.[1] It includes source code, feature model, feature configurations, semantic annotations, and documentation; MetaLib is explorable through a web application and maintained by a collaborative process.

Our work supplements prior DSL surveys in that theoretical aspects are mapped to idiomatic implementations facilitating end-to-end comparison taking advantage of feature modeling and model-based documentation (including semantic annotation).

Roadmap of the paper Section 2 summarizes the methodology underlying this research. Section 3 introduces our simple DSL for finite state machines (FSML). Section 4 analyzes FSML implementations to derive a refined feature model of DSL implementation. Section 5 presents MetaLib's model-based approach to documentation. Section 6 discusses threats to validity. Section 7 discusses related work. Section 8 concludes.

2 Methodology

The methodology for developing the MetaLib chrestomathy of DSL implementations is summarized in Fig. 1. We now describe each intermediate result (see legend) and how it was obtained.

2.1 Domain Analysis

We aimed at a set of most basic and common features of DSL implementation so that we could provide scope to our implementation efforts and have a starting point for feature modeling. To this end, we performed a simple domain analysis based on scholarly work on DSL implementation. In fact, we focused on work that surveys approaches. We consulted two PhD theses in the field of DSL [11, 31], as we were readily familiar with these (one of the authors was on the committees for the theses). We also consulted a paper on the evaluation and comparison of language workbenches [12], as it readily discusses features of DSL implementation approaches; we focus on non-IDE related aspects, that is, basic aspects of language implementation and metaprogramming,

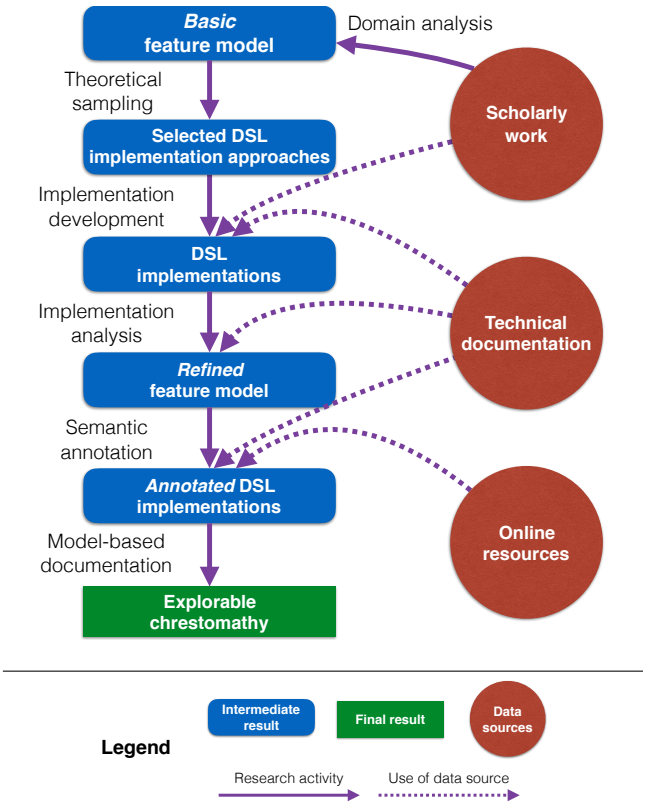


Figure 1. Methodology for the development of the MetaLib chrestomathy.

as noted in Section 1. We also consulted the *Software Languages Book* [25], as it discusses various DSL implementations in a relatively systematic manner and is in itself based on an extensive domain analysis of software language engineering.

In this manner, we observed (obviously) that DSL implementation is concerned with i) *syntax*, ii) typing or well-formedness or semantic analysis, referred to as *static semantics* by us, iii) execution or evaluation or interpretation, referred to as *dynamic semantics* by us, iv) as well as optimization or code generation or translation, referred to as *translation semantics* by us. These key features are also emphasized by the resources mentioned above–in particular: one of the PhD theses [31, Fig. 3.1] (an architectural description) and the language-workbench comparison [12, Fig. 1] (the major non-IDE-related group features).

There is also the established dichotomy of *concrete* versus *abstract syntax*, which we thus incorporated into the basic feature model. When it comes to concrete syntax, an existing feature model [12] readily provides a very general classification: *textual* versus *graphical* syntax, which we thus incorporated into the basic feature model. As far as textual syntax is concerned, we also observed the dichotomy of *projectional editing* versus *parsing*.

[1] http://www.softlang.org/metalib

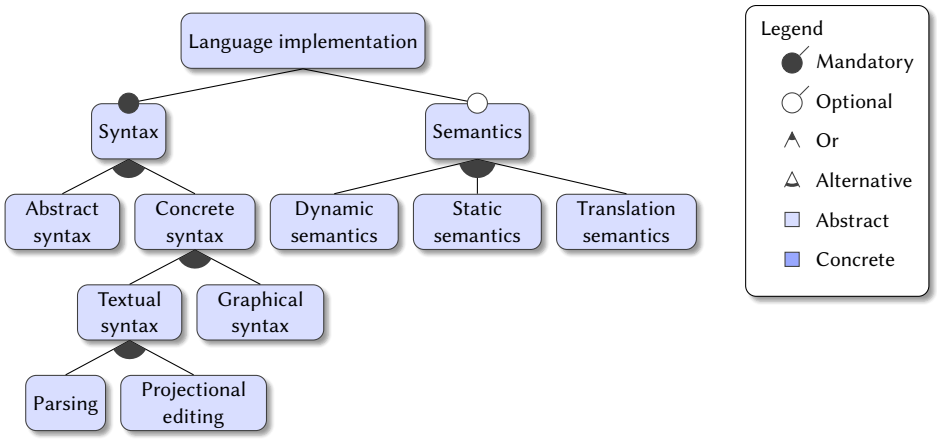


Figure 2. The basic feature model for DSL implementations. (When we use the terms Syntax, Semantics, etc. as feature names, we mean *implementation* (executable specification) of syntax, semantics, etc.)

There are many different ways in which abstract and concrete syntax can be supported, beyond parsing or projectional editing, but we defer the discovery of designated features to the implementation analysis phase (Sec. 4), which is driven by the concrete realizations. Hence, none of the features is assumed to be 'concrete' at this point, as we assume that the analysis of particular implementation approaches and actual implementations may reveal more variability.

The resulting basic feature model is summarized in Fig. 2. Syntax is a mandatory feature, because any sort of DSL implementation must implement syntax. Semantics is optional; for instance, if the goal is to provide a basic, graphical editor for a DSL, then this would be a syntax-only implementation. Yet, most implementations in the MetaLib chrestomathy implement (sub-features of) Semantics.

2.2 Theoretical Sampling

As there are many implementation approaches, related technologies, and applicable languages, we decided to perform theoretical sampling [10] to help with developing a manageable but representative suite of DSL implementations. We apply the following sampling criteria for identifying technologies and approaches:

Coverage of mainstream languages While DSL research often focuses on designated systems or technologies for DSL implementation, 'general purpose programming languages' may also serve as the host language for DSL implementation. We exercise *Java* (as a statically typed object-oriented programming language) and *Python* (as a dynamically typed multi-paradigm programming language) therefore.

Coverage of programming paradigms Thanks to the previous criterion, we readily cover imperative, object-oriented, and bits of functional programming. Additionally, we exercise *Haskell*, *Scala*, and *Racket* as representatives of statically and dynamically typed functional and functional-object oriented programming with well-known capabilities for DSL implementation.

Coverage of DSL implementation styles [11, 17, 25, 31] In external style, the syntax of the DSL is not tied to the host language and the language user may be largely oblivious to the host language; in internal style, the DSL is implemented essentially as a library in the meta- or host language; in a strong form of 'internal', the DSL's syntax and semantics is implemented and integrated into a host language [25, 31]. We exercise *ANTLR* for external style on top of *Java* and *Python*; we exercise *Java* and *Python* for internal style; as to the 'strong form', we exercise *Haskell*, *Scala*, and *Racket* with metaprogramming extensions for quasi-quotation or syntax macros.

Coverage of technological spaces We submit the hypothesis that DSLs may be implemented differently depending on the technological space [23] at hand. We exercise *EMF* and *Sirius* for modelware (or MDEware); we exercise *ANTLR*, *Rascal*, and *Spoofax* for grammarware. These seem to be the two most relevant technological spaces for DSL implementation. In particular, SQLware, XMLware, and RDFware are considered less relevant.

Coverage of the basic feature model Clearly, we should exercise all options of the basic feature model. This implies that we need to exercise parsing, which we cover, for example, by *ANTLR*-based implementations, graphical syntax, which we cover, for example, by a *Sirius*-based implementation, and projectional editing, which we cover specifically by an *MPS*-based implementation. The semantics-related features are covered by several implementations.

2.3 Implementation Development

We developed at least one DSL implementation for each identified technology and approach.

In addition to the technical documentation for exercised languages or technologies, we also consulted key publications about them: ANTLR [30]; EMF [35]; Haskell [17, 27, 34];

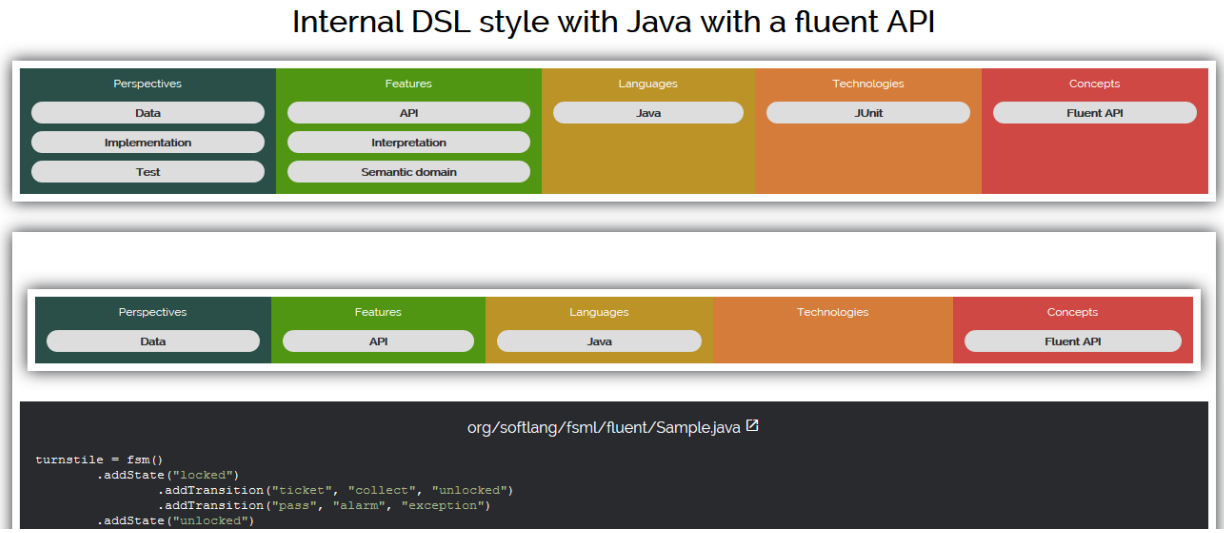


Figure 3. Snapshot of web-based view for MetaLib.

Racket [15]; MPS [38]; Rascal [21]; Scala [4, 33]; Sirius [37], Spoofax [19], Xtext [9].

The DSL implementations were meant to implement best practices for the languages or technologies, but there is clearly variability that we may have missed which we consider as a threat to validity and a future work topic.

2.4 Implementation Analysis

We analyzed the implementations to distinguish them from each other and to capture these differences as a refinement of the basic feature model. We will present the refined feature model in Sec. 4. In addition to identifying features we also identified metaprogramming-related concepts (e.g., quasi-quotation or fluent API). In this manner, we mean to connect the chrestomathy to the vocabularies of DSL implementation and specific approaches, languages, and technologies.

The identification process was supported by data mining techniques. That is, we processed the language- and technology-specific resources (Sec. 2.3) as follows. We performed PDF stripping, camel-case splitting, stop-word removal, stemming, changing every char to lower case and filtering all word smaller than two chars. Stemmed words were mapped back to the most prominent occurrence before stemming. Two ranks were computed: one for basic term frequency (TF) for each resource, another one for inverse document frequency (TF-IDF) with the collection of resources as documents. The authors then examined the top 50 for each resource and both ranks to identify potential features and concepts. (See the MetaLib website for the data set.)

We had to decide for each identified concept whether to promote it to a feature. In the pilot for this research, we assumed that we wanted to have features such as Internal DSL, Fluent API, External DSL, Pidgin ('a grammatically simplified form of a language' [31]), or Creole ('a mother tongue formed from the contact of two languages' [31]). Later, we

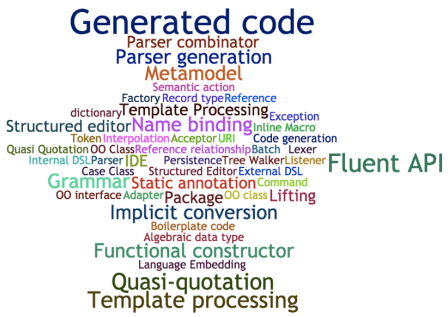


Figure 4. Word cloud of annotated concepts.

decided that examples like those given correspond to implementation 'patterns' or 'qualities' as opposed to features; such concepts still serve semantic annotation (see Sec. 2.5).

We also had to be careful not to designate features to each and every technique of a particular approach because we would otherwise end up with poor abstraction. For instance, the ANTLR technology supports parse-tree listeners, but we identified a more general feature, Abstraction, which covers the essential use case of parse-tree listeners: to map parse trees to abstract syntax trees (ASTs).

2.5 Semantic Annotation

We annotated the implementations with the used languages, technologies, and concepts. To this end, we used a semantic web approach such that we located the relevant entities on appropriate platforms, e.g., Wikipedia, and we documented these entities on a semantic wiki.[2] See Fig. 4 for an indication of tagged concepts.

[2] https://101wiki.softlang.org/

2.6 Model-based Documentation

We will present the documentation model in Sec. 5. The idea is that the DSL implementations with the attached documentation can be directly used for exploring approaches for DSL implementations. In Fig. 3, we show a snapshot of a web-based view for MetaLib for one particular DSL implementation; at the top, there is a summary of features, languages, technologies, and concepts; below, there is the first annotated code section with a DSL sample.

3 The DSL Implemented in MetaLib

We use a very simple finite-state machine language (FSML) as the DSL for implementation in the MetaLib chrestomathy.

3.1 Rationale for Choosing the DSL

FSML was chosen based on the following rationale:

i) The implementation of the language must involve aspects of textual and graphical syntax as well as static, dynamic, and translational semantics. This is, however, not a challenge, as pretty much any domain-specific modeling or programming language could be implemented in such a manner.
ii) We want the language to be very simple so that the resulting implementations are concise, which we assume, helps both implementors and learners.
iii) We want the underlying domain to be familiar to and readily used by the relevant research community. This is certainly the case for the domain of behavioral modeling with finite state machines.

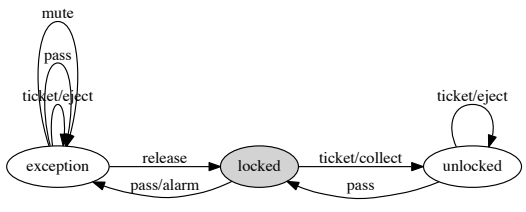
We note that the chosen language and the underlying feature model are very simple, when compared to related comparison efforts, including the transformation-tool contest[3] or the language-workbench challenge[4]. This is a consequence of our focus on basic metaprogramming techniques for DSL implementation, as opposed to a technology- or tool-focused comparison.

3.2 Language Reference

As a semi-normative reference, we specify and illustrate the FSML syntax and semantics in the sequel.

3.2.1 Concrete Syntax

We begin with an illustration of graphical syntax:



[3] http://www.transformation-tool-contest.eu/
[4] http://www.languageworkbenches.net/

The example is concerned with a turnstile (a revolving door) as it may be used in a metro system.[5] An implementation of graphical syntax may deviate from presentational details, but it is expected that any implementation represents states (nodes), transitions (edges), and includes details for state ids, event ids, and action ids. Also, the initial state should be marked (in our illustration, the node 'locked' is filled to designate it as the initial state). Furthermore, it is assumed that the turnstile example is a normative DSL example that should be exercised by any FSML implementation.

Here is the turnstile example in textual syntax:

```
initial state locked {
  ticket/collect –> unlocked;
  pass/alarm –> exception;
}
state unlocked {
  ticket/eject;
  pass –> locked;
}
state exception { ... }
```

An implementation of textual syntax should implement the textual syntax as shown. Thus, we define the syntax by a normative context-free grammar; we use the EBNF-like grammar notation of the Software Languages Book [25]:

```
fsm : {state}* ;
state : {'initial'}? 'state' stateid '{' {transition}* '}' ;
transition : event {'/' action}? {'–>' stateid}? ';' ;
stateid : name ;
event : name ;
action : name ;
// Lexical syntax
name : { alpha }+ ; // alpha proxies for letters
layout : { space }+ ; // To be skipped along parsing
```

3.2.2 Abstract Syntax

The abstract syntax of FSML could be defined in different ways, also depending on the actual definition formalism and the general choice between trees versus graphs. Here is an illustrative definition for the tree-based abstract syntax of FSML; we use the Haskell-/ML-like signature notation of the Software Languages Book [25]:

```
// Sequences of state declarations
type fsm = state* ;
// State declarations with all transitions
type state = initial × stateid × transition* ;
type initial = boolean ;
// Transitions for a given source state
type transition = event × action? × stateid ;
type stateid = string ;
type event = string ;
type action = string ;
```

[5] There are states for the door to be locked or unlocked or to be in an exceptional state, when a person was trying to pass without inserting a ticket. A transition between the states is labeled by an event that triggers the transition, possibly based on a sensor in the real system and an optional action that corresponds to some observable behavior based on an actor in the real system.

Here is also an illustrative definition for the graph-based abstract syntax of FSML; we use the EMF-like metamodeling notation of the Software Languages Book [25]:

```
class fsm { part states : state* ; }
class state {
  value initial : boolean ;
  value stateid : string ;
  part transitions : transition* ;
}
class transition {
  value event : string ;
  value action : string? ;
  reference target : state ;
}
```

An instance of the signature is also referred to as AST (Abstract Syntax Tree). An instance of the metamodel is referred to as ASG (Abstract Syntax Graph). We also speak of 'model' in both cases.

3.2.3 Dynamic Semantics

We sketch an operational semantics (small-step style) on top of the tree-based abstract syntax shown earlier. There is this small-step judgment with appropriate metavariables for the earlier types:

$$f \vdash \langle x, e\rangle \rightarrow \langle x', out\rangle$$

That is, the finite state machine (FSM) f is interpreted ('simulated') to make a transition from a state with id x to a state with id x' while handling an event e, and producing possibly some output *out* (zero or one actions). The reflexive, transitive closure requires a similar judgment:

$$f \vdash \langle in\rangle \rightarrow^* \langle x, out\rangle$$

That is, the FSM f starting from the initial state and an input *in*, the complete input is consumed ending in a state with id x and the complete output *out* with the actions for the transitions.

We specify the one-step relation.

$$\frac{}{\langle \ldots, \langle b, x, \langle \ldots, \langle e, \langle a\rangle, x'\rangle, \ldots\rangle\rangle, \ldots\rangle \vdash \langle x, e\rangle \rightarrow \langle x', \langle a\rangle\rangle} \quad [\text{action}]$$

$$\frac{}{\langle \ldots, \langle b, x, \langle \ldots, \langle e, \langle\rangle, x'\rangle, \ldots\rangle\rangle, \ldots\rangle \vdash \langle x, e\rangle \rightarrow \langle x', \langle\rangle\rangle} \quad [\text{no-action}]$$

That is, there are only two axioms: one for the case of an applicable transition with an action, another one for an applicable transition without action. In both axioms, we simply decompose the FSM from the context to locate a suitable transition (i.e., one with event e within a suitable state declaration (i.e., the one for the current state x). The located transition provides the new state id x' and optionally an action a.

3.2.4 Static Semantics

The static semantics (well-formedness of FSMs) could also be specified by a deductive system like the one for dynamic semantics, but we omit such a specification here for brevity. Overall there are constraints described informally as follows: i) the ids of the declared states need to be distinct; ii) there must be exactly one initial state; iii) the events must be distinct for each state's transitions. iv) the target state of each transition must be declared; v) all states must be reachable from the initial state. For instance, here is an example violating the last constraint:

```
initial state stateA { eventI/actionI -> stateB; }
state stateB { }
state stateC { }
```

3.2.5 Translation Semantics

We decide to approach the feature of translation semantics in a less normative manner because we noted up-front that some metaprogramming approaches target translation or code generation in the context of DSL implementation in a specific manner, as we will substantiate in Section 4.

As one approach to translation, we assume that FSMs are translated to C code with some dispatching logic for state transition. This is illustrated here for the turnstile FSM:

```
enum State {LOCKED,UNLOCKED,EXCEPTION,UNDEFINED};
enum State initial = LOCKED;
enum Event {TICKET,RELEASE,MUTE,PASS};
void alarm() { }
void eject() { }
void collect() { }
enum State next(enum State s, enum Event e) {
    switch(s) {
        case LOCKED:
            switch(e) {
                case TICKET: collect(); return UNLOCKED;
                case PASS: alarm(); return EXCEPTION;
                default: return UNDEFINED;
            }
        case UNLOCKED: ...
        case EXCEPTION: ...
        default: return UNDEFINED;
    }
}
```

That is, there are `enum` types for the states and the events; there are functions for the actions; state transition is modeled by a function `next` which uses `switch/case` -statements to map the current state and a given event to a new state accompanied by a call of the function for an action, if specified.

4 Implementation Analysis

Based on the sampling of Section 2, we implemented FSML with the identified languages, technologies, and approaches; see Fig. 5. We consulted technical documentation and scholarly work, as identified earlier (Section 2.3). Within the author team, we held code review meetings where the primary developer of an implementation would need to explain the implementation overall and defend made choices to reasonably conform to best practices. Eventually, the discussion would aim at the identification of sub-features as described below.

Chrestomathy member	Languages & technologies
javaInfluentInternal	Java
javaFluentInternal	Java
javaExternal	Java, ANTLR
pythonInternal	Python, Graphviz
pythonExternal	Python, ANTLR
haskellQuasiQuotation	Haskell (+TH+QQ)
scalaEmbedded	Scala
mps	MPS
spoofax	Spoofax
racket	Racket
rascal	Rascal
emfXMI	EMF
emfSirius	EMF, Sirius
emfXtext	EMF, Xtext

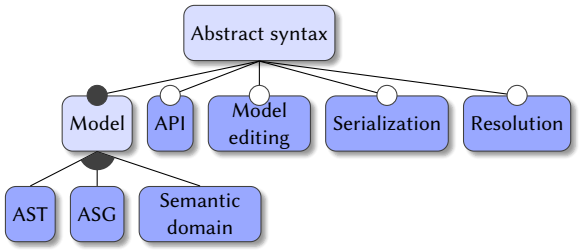
Figure 5. FSML implementations in MetaLib.

4.1 Refinement of the Basic Feature Model

We refine the model as of Fig. 2.

4.1.1 Abstract Syntax

Here is the corresponding refinement:



By exercising pure functional programming (in Haskell) and metamodeling (with EMF specifically), we encountered the obvious AST versus ASG (i.e., tree versus graph) dichotomy, thereby suggesting corresponding subfeatures. We observed that our internal DSL style implementations in Java (javaInfluentInternal) preferred trees over graphs despite the availability of reference semantics because, in this manner, the resulting API was more convenient (think of using a target state in a transition before declaring the state).

The object-oriented implementations with their different object models also made us realize that an important aspect of abstract syntax, especially in internal DSL-style implementation, is the actual API and thus we started separating Model (representation) versus API in the feature model.

In one internal DSL-style implementation (javaFluentInternal), we encountered a model that was closer to a 'semantic domain' (in the sense of semantics) than a tree- or graph-like structure (in the sense of syntax); we show Java code for illustration:

```
private HashMap<
            String,
            HashMap<String, ActionStatePair>
        > fsm = new HashMap<>();
```

That is, the model is a cascaded map for maintaining states and transitions; lookup directly models the semantics of state transition. We determined that it is not uncommon that a DSL implementation may designate a model which captures already semantics, to some extent, and thus, we created the feature Semantic domain as a subfeature of Model–next to AST and ASG. Model is an or-feature because, in principle, a DSL implementation may use different representations.

Most clearly in the context of the EMF-based implementation (emfXMI), we observed that abstract syntax-based (model-based) serialization is an important concern and thus, we created the subfeature Serialization.

Models (instances of abstract syntax) are editable, more or less, as is—that is, subject to a generic projection, which however may be customized to some extent. For instance, EMF's possibly customized model editor (emfXMI) supports such model projection. Therefore, we added the Model editing feature as an optional extension to Abstract syntax.

Finally, we also experimented with implementations that used both ASTs and ASGs (emfXtext)—the former for initial construction, e.g., by means of a fluent API and the latter as the ultimate representation. We created the subfeature Resolution for such a mapping from ASTs and to ASGs.

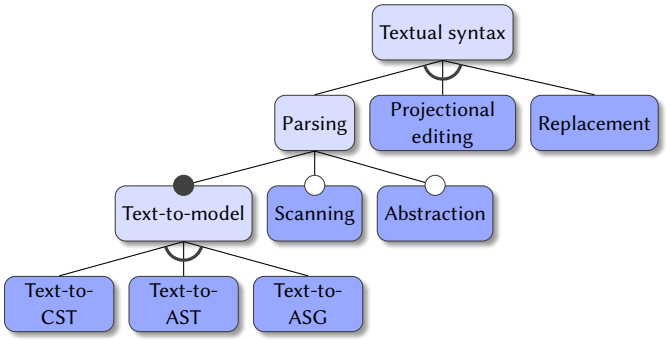
4.1.2 Textual Syntax

We expected to encounter many different kinds of parsers in the context of the implementation of textual syntax or projectional editing; we decided not to consider text formatting (pretty printing) for the DSL. When it comes to parsing, one could end up re-capturing classifications of grammar-class restrictions and parsing algorithms. Instead, we aimed at a high level of abstraction focusing on the I/O behavior of parsing. We observed that some implementations exposed a concrete syntax tree (CSTs), others went right away to ASTs, yet others to ASGs. Thus, we created the features Text-to-CST, Text-to-AST, and Text-to-ASG as subfeatures of Parsing. Some parsers are scannerfull (i.e., they implement a scanner and expose a token stream), others are scannerless, and thus, we created the optional subfeature Scanning.

There exists much variation on projectional editing [3, 12] from which however we aggressively abstract to only one feature already identified in the domain analysis. Thus, Projectional editing is turned into a concrete feature. (Specializations are conceivable, e.g., tabular versus template-based text formats.) MPS (mps) supports such projectional (text) editing.

We also observed that implementations may designate functionality to the actual mapping from CSTs to ASTs or ASGs and thus, we created the subfeature Abstraction. For instance, use of *ANTLR* (e.g., javaExternal) would qualify for Text-to-CST because ANTLR builds parse trees anyway. We may then use ANTLR's parse-tree listeners for Abstraction.

Some implementations (racket, scalaEmbedded) use techniques other than classic parsing to implement the textual syntax, e.g., macros (syntax rules) or parse-tree rewriting, and thus, we created the subfeature Replacement next to Parsing and Projectional editing. Thus:



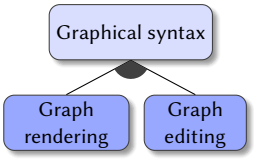
We use alternative features: a parser either maps text to CSTs, ASTs, or ASGs; also, concrete syntax is implemented by either parsing, projectional editing, or replacement.

4.1.3 Graphical Syntax

Clearly, there is large scale on its own, how exactly graphical syntax can be implemented, what flexibility is provided in affecting details of the graphical representation. We limit ourselves here to two major options; exploration of graphical editor frameworks is not our goal.

Graph rendering Graphical syntax is only implemented up to the point that a DSL program can be rendered as a graph according to the visual syntax. For instance, a DSL implementation (e.g., pythonInternal) may target Graphviz' DOT[6] for rendering.

Graph editing There is also editing support for the visual syntax. For instance, Sirius (or GMF or Graphiti) may be used on top of EMF (emfSirius) to achieve graph editing.



4.1.4 Dynamic Semantics

We did not exercise any interesting variation for dynamic semantics. The implementations were straightforward interpreters. No special run-time system or library support was required. We simply use the concrete feature Interpretation in all these cases.

4.1.5 Static Semantics

There are two features:

Analysis A metaprogram implements the static semantics as a type or well-formedness checker on top of the object-program representation for DSL programs (e.g., spoofax).

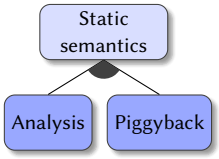
[6] http://www.graphviz.org/

Piggyback By translating object programs according to the translation semantics—the target language of translation, in particular, its type system—may cover some or all aspects of the DSL's static semantics (haskellQuasiQuotation, scalaEmbedded, racket).

For instance, assume that an FSM is not deterministic (Sec. 3.2.4), that is, in a given state StateX, there is more than one transition with a given event, EventY. When generating C code, this would lead to dispatching code like this:

```
switch(e) {
    case EventY: ...;
    case EventY: ...;
    default: return UNDEFINED;
}
```

The compilation of the generated code would catch the violation of the constraint for deterministic FSMs.



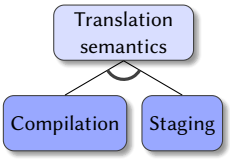
The features Analysis and Piggyback may also be combined because the intrinsic checks of generated code may need to be complemented by a partial analysis prior to translation.

4.1.6 Translation Semantics

There are two features:

Compilation A compiler-like metaprogram maps DSL programs to programs in another language. For instance, a Java-based implementation (javaInfluentInternal) may use template processing to generate C-code.

Staging A meta-program uses language concepts for program generation in the sense of staged computation [36] to translate DSL programs to phrases of the metalanguage—typically at compile time (haskellQuasiQuotation, scalaEmbedded).



4.2 Feature Dependencies

There are a few constraints on the refined feature model:

$$\text{Text-to-AST} \Rightarrow \text{AST} \quad (1)$$

$$\text{Text-to-ASG} \Rightarrow \text{ASG} \quad (2)$$

$$\text{Resolution} \Rightarrow \text{AST} \wedge \text{ASG} \quad (3)$$

$$\text{Piggyback} \Rightarrow \text{Translation semantics} \quad (4)$$

$$\text{Abstraction} \Rightarrow \text{Text-to-CST} \wedge \text{Abstract syntax} \quad (5)$$

	emfSirius	emfXMI	emfXtext	haskellQuasiQuotation	javaExternal	javaFluentInternal	javaInfluentInternal	mps	pythonExternal	pythonInternal	racket	rascal	scalaEmbedded	spoofax
Abstract syntax	×	×	×	×	×	×	×	×	×	×	×	×	×	
AST	×	×	×	×	×		×	×	×	×	×	×	×	
ASG	×	×	×											
Semantic domain						×								
Model editing	×	×												
API	×	×	×			×	×		×	×		×		
Serialization	×	×							×					
Resolution	×	×	×											
Textual syntax			×	×	×			×	×		×	×	×	×
Text-to-CST					×							×		×
Text-to-AST			×	×	×				×				×	
Text-to-ASG			×											
Projectional editing								×						
Scanning			×	×	×				×					
Abstraction					×									
Replacement								×			×		×	
Graphical syntax	×	×	×							×		×		
Graph rendering	×		×							×		×		
Graph editing	×	×												
Dynamic semantics						×	×			×	×			
Interpretation						×	×			×	×			
Static semantics		×	×	×				×		×	×	×	×	×
Analysis		×	×	×				×		×	×	×	×	×
Piggyback				×							×		×	
Translation semantics			×	×			×	×		×		×	×	×
Compilation			×				×	×		×		×		×
Staging				×									×	

Figure 6. Coverage of features by implementations.

That is, Parsing can only target ASTs or ASGs, if the corresponding subfeatures of Model are selected (1 and 2). Further, we assume feature Resolution to correspond to a mapping from ASTs to ASGs (3). Arguably, one could also speak of resolution when ASTs are navigated and subtrees are looked up, e.g., within the implementation of semantics. Further, we clarifiy that piggybacking for the static semantics requires that a translation semantics is implemented (4). Finally, we clarify that abstraction, per our definition, involves CSTs and abstract syntax (5).

4.3 Coverage of Feature Model

Fig. 6 captures the feature configurations for the implementations fitting the theoretical sampling of Section 2.2.

5 Model-based Documentation

As illustrated in Fig. 7, MetaLib assumes two fundamental roles ('hats'): the development role ('code' in the figure) in which to implement FSML with a given approach; the documentation role ('models' in the figure) in which to analyse the implementation, tag features, languages, technologies, and concepts, and possibly make suggestions towards the revision of the feature model. MetaLib's infrastructure automatically composes code and documentation ('models') to check 'well-formedness' of documentation and to publish a web-explorable view (Fig. 3).

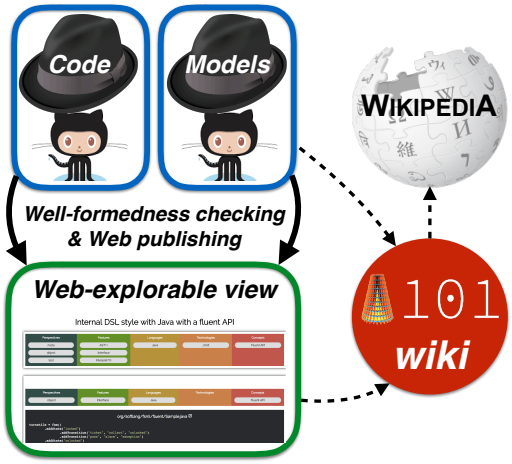


Figure 7. MetaLib's documentation approach.

5.1 Rationale for Documentation Approach

i) In the developer role, there should be no burden regarding MetaLib-specific documentation. The developer should focus on implementing FSML adhering to best practices for the approach at hand.
ii) In the documentation role, there should be guidance on what and how to document. The author should focus on adding documentation elements that directly or indirectly connect the given implementation with other resources (other implementations, semantic wiki 101wiki, Wikipedia).
iii) Collaborative development and documentation leverages distributed version control and source code management (GitHub). New or revised models are pushed to the central MetaLib repository or pull-request are used. The code (but not the model) can be outside the MetaLib repository.
iv) Prior to publishing a model (i.e., a contribution to MetaLib), well-formedness checking is applied.
v) All the semantic entities of MetaLib (features, languages, technologies, and concepts) are hosted on the semantic wiki 101wiki, which in turn references other knowledge resources, e.g., Wikipedia.

5.2 A Sample Model

The following JSON-based model illustrates the part that is shown in Fig. 3:

```
{ "name": "javaFluentInternal",
  "baseuri": "https://github.com/softlang/yas/tree/master/languages/FSML/Java/org/
        softlang",
  "headline": "Internal DSL style with Java with a fluent API",
  "sections": [
    { "features": [ "API" ],
      "perspectives": [ "data"],
      "languages": [ "Java"],
      "concepts": [ "Fluent API"],
      "technologies": [],
      "artifacts": [{ "type": "all", "link": "fsml/fluent/Sample.java"}]
    },
    ...
  ]
}
```

```
// Documentation of contributions
class document {
  value name : string; // The name of the contribution
  value headline : string; // A one–liner explanation
  value baseuri : string; // Base URI for links
  part sections : section+; // Sections of the documentation
}

// Sections in a documentation
class section {
  value headline : string?; // Optional one–liner explanation
  part perspectives : perspective+; // Perspective of section
  value features : string+; // Features addressed by section
  value languages : string*; // Languages used
  value technologies : string*; // Technologies used
  value concepts : string*; // Concepts used
  part artifacts : artifact+; // Artifacts to be shown
}

// Perspectives of documentation
enum perspective {
  implementation, // i.e., feature implementation
  data, // e.g., instance of grammar or metamodel
  test, // i.e., application of implementation
  build, // e.g., code generator application
  capture // e.g., screenshot or session log
}

// Artifacts for projected by section
abstract class artifact {
  value link : string; // A relative URI
  value format : string; // MIME–like format type
}
class none extends artifact { } // Nothing to show
class all extends artifact { } // All to show
class some extends artifact { // A specific line range to show
  value from : integer;
  value to : integer;
}
```

Figure 8. Metamodel of MetaLib Documentation Format.

That is, a name is attached to the contribution, a headline (a short summary) is provided, the GitHub base URI is identified and one of several sections (i.e., projections into code of the contribution) is shown. The section is concerned with a Java file which illustrates the use of a fluent API for FSML.

5.3 The Documentation Metamodel

Fig. 8 shows an EMF-like metamodel for documentation (with a straightforward mapping to JSON). A chrestomathy member is documented by a sequence of 'sections'. Each section projects some'artifacts': source code, data, or even screenshots. Each section relates to a set of 'features'—just one feature in the case of modular implementations. Each section takes a certain 'perspective'.

There are the following perspectives. First, we are concerned with actual feature 'implementation'. Second, we may be concerned with 'data' to exercise the implementation. In the case of metaprogramming, 'implementation' proxies for metaprogram functionality whereas 'data' proxies for object programs or other data consumed or produced by metaprograms. When projecting any sort of code, a 'selection' is attached to specify whether and, if so, how much code is to be shown. Some artifacts (such as blobs of XML) should not be shown; other artifacts should not be shown in full, but only an excerpt thereof because of their size.

The use of perspectives is a key property of MetaLib's documentation approach; perspectives could be useful for any sort of documentation relative to a feature model. That is, perspectives allow to document artifacts other than just implementations of features because we may also tag and document artifacts that are related to features in other ways.

6 Threats to Validity

To enhance *external validity*—that is, the applicability of our feature model to annotate new implementations for the chrestomathy—we systematically selected DSL implementation technologies that are well-known and used in practice. These also cover different technological spaces (e.g., programming languages such as Java and Python, and editor technologies such as parser-based and projectional editing). This selection of approaches can be seen as theoretical sampling [10], commonly used in case-study research, where a representativeness of cases can usually not be established (since the whole population of cases is unknown), but where the coverage of certain criteria (cf. Section 2.2) is desired.

To enhance *internal validity*—that is, that the DSLs were implemented and annotated correctly—the four authors who implemented the DSLs had extensive (yet, academic) experience in model-driven and software language engineering, at least at a Master's level (one conducted his Bachelor's thesis as a preparation for this work). All have either taught or attended a software-language engineering (SLE) course. Extensive experience existed for all technologies; experience was more limited for Rascal, Racket, MPS, and Spoofax. For the latter, the implementers studied documentation and cross-checked even more carefully (in addition to the general cross-check as described in Section 4) their implementations. However, the existing experience with SLE concepts helped significantly. We discuss directions for additional validations of our chrestomathy as future work in Section 8.

7 Related Work

The broader related work scope is some form of comparison of software languages, technologies, or approaches using those. For each entry of related work, it makes sense to examine three points:

- What are the subjects of comparison? ('What')
- What is the method of comparison? ('How')
- What is the purpose of comparison? ('Why')

As a point of reference, the research of the present paper compares DSL implementations ('what'), on the grounds of a feature model derived by domain analysis and implementation analysis ('how') for the purpose of a chrestomathy for DSL implementation to be useful for learning while relying on model-based documentation to this end ('why').

The work the most closely related to ours is on the evaluation and comparison of language workbenches [12]. The subjects of comparison are actual workbenches ('what'). The method comparison involves agreement on challenges (complex tasks), the attempted implementation of the challenges with the various workbenches, and the related definition of a feature model covering both options and capabilities of workbenches ('how'). The purpose is essentially understanding the different workbenches and contrasting them, but also making suggestions for future research on workbenches ('why'). In addition to these different positions on all three points, there is also a major difference regarding the involved feature model. In the case of the workbench research, the model focuses on 'services' and IDE-related aspects, whereas, in our research, the model focuses on metaprogramming.

There is a string of related work on essentially surveying approaches ('what') in domains related to metaprogramming for DSL implementation: model transformation [7], generative programming [8], generic functional programming [32], and DSL implementation [22]. A more or less formal feature model is used in such work to summarize or organize the survey ('how'). In the latter, arguably most closely related case, as it also addresses DSL implementation, comparison is concerned with DSL implementor and end-user effort ('why'); see also [20] for a similar what/how/why.

There exist various program or software chrestomathies; see [24] for a survey and characteristics of chrestomathies. In the broader area of programming, Rosetta Code[7] is a well-known and advanced example of a chrestomathy. Rosetta Code collects programming tasks and task solutions in different programming languages ('what'); we quote from the website (as of 11 May 2017): "*Rosetta Code is a programming chrestomathy site. The idea is to present solutions to the same task in as many different languages as possible, to demonstrate how languages are similar and different, and to aid a person with a grounding in one approach to a problem in learning another. Rosetta Code currently has 847 tasks, 198 draft tasks, and is aware of 650 languages, though we do not (and cannot) have solutions to every task in every language.*" The underlying method is thus to administrate the inclusion of new tasks and groups thereof as well as compliant task solutions to be presented next to each other ('how'). As evident from the quote, comparison is meant here to be useful for learning ('why'). The metaprogramming domain is only touched upon on Rosetta Code.[8]

In fact, there exists research on top of Rosetta Code [29] such that the collection is used to compare programming languages in terms of a number of criteria, such as conciseness. Such secondary uses ('why') of chrestomathies may eventually happen once those collections of artefacts become 'interesting enough'. For instance, the '101' chrestomathy [14] which collects implementations of small information systems ('what') was also eventually used beyond 'learning' ('why'). That is, '101' served as the corpus in comparing languages and technologies in terms of basic code metrics [26] and it provided a code base for studying product line engineering in the context of clone management [1]. It remains to be seen whether MetaLib supports secondary uses.

We are not aware of related work on the specific subject of how to make chrestomathies more useful for the assumed user (the learner), other than perhaps our previous work linking documentation and source code [13]. In actual 'implementations' such as Rosetta Code, as discussed above, this subject is addressed in a pragmatic manner. In the present paper, we aimed at designing a documentation model that supports the learner's experience in an explainable manner; validation is pending. Clearly, there are approaches other than 'collections of systems' to convey domain knowledge in programming. For instance, Günther and Fischer [16] develop a pattern catalog to communicate (Ruby's) metaprogramming features.

8 Conclusion

Mastering the metaprogramming domain in a teaching context entails considerable effort on the side of both teacher and student because of the high level of abstraction in metaprogramming and the myriad of options (languages, technologies, and approaches). On one side, one would like to cover several options (e.g., different systems such as EMF, Rascal, and Haskell with quasi-quotation), as each option makes valuable contributions to the overall domain. On the other side, coverage of multiple options is nearly impractical because of the involved notations and idiosyncratic techniques. Thus, the challenge is to arrive at a knowledge resource that is palatable—this can be compared to teaching programming paradigms, which clearly aims at conveying conceptual knowledge as opposed to making students fluent in a range of programming languages. We have designed MetaLib as a knowledge resource that operates at the conceptual level of metaprogramming and DSL implementation. We have started to use MetaLib in teaching.[9] We plan to evolve MetaLib in future SLE courses and hope to get more collaborators involved eventually.

Future work Validation of a chrestomathy like MetaLib is a challenging topic. Here are some suggestions: 1) Although the implementations including their documentation are relatively simple, we plan to have them reviewed by experts on the technologies. 2) The contributions should be

[7] http://rosettacode.org/wiki/Rosetta_Code
[8] https://rosettacode.org/wiki/Metaprogramming
https://rosettacode.org/wiki/Extend_your_language

[9] 29/30 May, PhD course by Ralf Lämmel, GSSI, l'Aquila, http://www.softlang.org/course:gssi17; 6 June, Lecture in Master-level lecture series by Ralf Lämmel, University of l'Aquila, l'Aquila, http://www.softlang.org/course:univaq17.

re-implemented for means of cross-validation and agreement with best practices. 3) We plan to investigate whether the chrestomathy assists effectively in learning SLE technologies. Thus, we may compare how students perform at similar programming tasks while having access to the chrestomathy versus a control group without such access.

References

[1] Michal Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stanciulescu, Andrzej Wasowski, and Ina Schaefer. 2014. Flexible product line engineering with a virtual platform. In *Proc. ICSE 2014*. ACM, 532–535.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

[3] T Berger, M. Voelter, H. P. Jensen, T Dangprasert, and J. Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *Proc. FSE 2016*. ACM.

[4] Eugene Burmako. 2013. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proc. 4th Workshop on Scala, SCALA@ECOOP 2013*. ACM, 3:1–3:10.

[5] Walter Cazzola and Edoardo Vacchi. 2016. Language Components for Modular DSLs using Traits. *Computer Languages, Systems & Structures* 45 (2016), 16–34.

[6] Krysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional. 1st Edition.

[7] Krzysztof Czarnecki and Simon Helsen. 2006. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45, 3 (2006), 621–646.

[8] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. 2003. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation (LNCS)*, Vol. 3016. Springer, 51–72.

[9] Sven Efftinge and Markus Völter. 2006. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, Vol. 32. 118.

[10] Kathleen M Eisenhardt and Melissa E Graebner. 2007. Theory building from cases: Opportunities and challenges. *Academy of management journal* 50, 1 (2007), 25–32.

[11] Sebastian Erdweg. 2013. *Extensible Languages for Flexible and Principled Domain Abstraction*. Ph.D. Dissertation. Philipps-Universität Marburg.

[12] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47.

[13] Jean-Marie Favre, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. 2012. Linking Documentation and Source Code in a Software Chrestomathy. In *Proc. WCRE 2012*. IEEE, 335–344.

[14] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. 2012. 101companies: A Community Project on Software Technologies and Software Languages. In *Proc. TOOLS 2012 (LNCS)*, Vol. 7304. Springer, 58–74.

[15] Matthew Flatt. 2012. Creating languages in Racket. *Commun. ACM* 55, 1 (2012), 48–56.

[16] Sebastian Günther and Marco Fischer. 2010. Metaprogramming in Ruby: a pattern catalog. In *Proc. PLoP 2010*. ACM, 1:1–1:35.

[17] P. Hudak. 1998. Modular Domain Specific Languages and Tools. In *Proc. ICSR 1998*. IEEE, 134–142.

[18] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. CMU.

[19] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proc. OOPSLA 2010*. ACM, 444–463.

[20] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2010. On the impact of DSL tools on the maintainability of language implementations. In *Proc. LDTA 2010*. ACM, 10.

[21] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2011. EASY Metaprogramming with Rascal. In *GTTSE 2009, Revised Papers (LNCS)*, Vol. 6491. Springer, 222–289.

[22] Toma Kosar, Pablo E. Martínez López, Pablo A. Barrientos, and Marjan Mernik. 2008. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology* 50, 5 (2008), 390–405.

[23] Ivan Kurtev, Jean Bézivin, and Mehmet Akşit. 2002. Technological Spaces: an Initial Appraisal. In *Proc. of CoopIS, DOA 2002, Industrial track*.

[24] Ralf Lämmel. 2015. Software chrestomathies. *Sci. Comput. Program.* 97 (2015), 98–104.

[25] R. Lämmel. 2017. *Software languages: Syntax, semantics, and metaprogramming*. Springer. To appear. See http://www.softlang.org/book.

[26] Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. 2014. Comparison of feature implementations across languages, technologies, and styles. In *Proc. CSMR-WCRE 2014*. IEEE, 333–337.

[27] Geoffrey Mainland. 2007. Why it's nice to be quoted: quasiquoting for Haskell. In *Proc. Haskell 2007*. ACM, 73–82.

[28] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (2005), 316–344.

[29] Sebastian Nanz and Carlo A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *Proc. ICSE 2015*. IEEE Computer Society, 778–788.

[30] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[31] Lukas Renggli. 2010. *Dynamic Language Embedding With Homogeneous Tool Support*. Ph.D. Dissertation. Universität Bern.

[32] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. 2008. Comparing libraries for generic programming in Haskell. In *Proc. of Haskell 2008*. ACM, 111–122.

[33] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* 25, 1 (2012), 165–207.

[34] Tim Sheard and Simon L. Peyton Jones. 2002. Template metaprogramming for Haskell. *SIGPLAN Notices* 37, 12 (2002), 60–75.

[35] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework* (2 ed.). Pearson Education.

[36] Walid Taha. 2008. A Gentle Introduction to Multi-stage Programming, Part II. In *GTTSE 2007, Revised Papers (LNCS)*, Vol. 5235. Springer, 260–290.

[37] V. Viyović, M. Maksimović, and B. Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *Proc. INES 2014*. IEEE, 233–238.

[38] Markus Voelter. 2013. Language and IDE Modularization and Composition with MPS. In *GTTSE 2011, Revised Papers (LNCS)*, Vol. 7680. Springer, 383–430.