Yet another annotated SLEBOK bibliography

Ralf Lämmel

Version 0.003 (2 November 2014)

Abstract

Software Language Engineering (SLE) is a particular view on Software Engineering (SE), which pays specific attention to the many software languages that are used in software development. These are not just programming languages, but also modeling languages, query and transformation languages, schema languages—many of them to be considered domain-specific languages. SLE is concerned with design, implementation, testing, deployment, and evolution of software languages as well as language-based software components.

The purpose of this annotated bibliography is to contribute to the SLE body of knowledge (SLEBOK). The bibliography collects a manageable set of papers that cover many principles and practilities of SLE in an accessible manner. The intension is to favor more fundamental, general papers over specific, highly technical papers. The selection is otherwise not very systematic. The SLE and GTTSE venues were assumed to provide key papers. Yet other venues, such as OOPSLA (SPLASH), ECOOP, and CC as well as special issues on the SLE topic or its vicinity were also considered. Several papers were simply included based on the author’s long-term exposition to SLE school of thought. Moreover, several SLE researchers have provided advice on what additional papers to include.

The bibliography could be useful in teaching. In fact, the selection of papers is largely based on what I have covered or wish or could imagine to cover in a relatively advanced SLE course.
# Contents

1 Disclaimer 3

2 Acknowledgment 3

3 Metamodel of the bibliography 4

4 Papers of the bibliography 5

4.1 Koskimies91 ........................................ 6
4.2 Hughes95 ........................................ 8
4.3 Reynolds98 ....................................... 10
4.4 SirerB99 ........................................ 12
4.5 Sheard01 ........................................ 14
4.6 KurtevBA02 ...................................... 16
4.7 Thomas03 ........................................ 18
4.8 Hainaut06 ....................................... 20
4.9 HappelS06 ....................................... 22
4.10 Bezivin06 ...................................... 24
4.11 BezivinBFGJKKP06 ............................. 26
4.12 BravenboerTV06 ................................. 28
4.13 AlvesV09 ....................................... 30
4.14 Wachsmuth09 .................................... 32
4.15 Moody09 ....................................... 34
4.16 RenggliGN10 .................................... 36
4.17 HeidenreichJSWB09 ............................ 38
4.18 Cordy11 ......................................... 40
4.19 ErwigW12a ...................................... 42
4.20 CookL11 ......................................... 44
4.21 HerrmannsdoerferVWI11 .................... 46
4.22 MullerFBC12 .................................... 48
4.23 JezequelCDGR12 ............................... 50
4.24 VolterSBK14 .................................... 52
1 Disclaimer

This is unfinished work.

2 Acknowledgment

The following people have made suggestions for inclusion into the bibliography or given more general advice on the project: Anya Helene Bagge, David Lorenz, Richard Paige, Andrei Varanovich, Guido Wachsmuth, Andreas Winter, Vadim Zaytsev.
3 Metamodel of the bibliography

The document is generated from a model. The metamodel is given here informally in terms of how the document looks like. There is one page per entry with data as follows:

**Key** Descriptor of the paper.

**Title** Title of the paper.

**Citation** Bibtex citation for the paper.

**Online URL** Public access where possible.

**Required concepts** Assumed background.

**Provided concepts** Knowledge areas served.

**Annotation** Description of the paper.

**Figure** An illustration.

The illustration consists of an annotated figure, which is taken either directly from the paper or assembled. Annotations of papers and their illustrations may also refer to works which are not part of the selection.
4 Papers of the bibliography
4.1 Koskimies91

Koskimies91 – Data

Citation
[17]

Title
Object-Orientation in Attribute Grammars

Online URL
http://link.springer.com/chapter/10.1007%2F3-540-54572-7_11

Required concepts
context-free grammar, attribute grammar, object orientation

Provided concepts
object-oriented context-free grammar, object-oriented context-free grammar

Annotation
The attribute grammar formalism is married with the object-oriented paradigm. Arguably, a side effect of this marriage is that the underlying context-free grammar formalism is also married with object orientation, which is interesting in so far that this (early) explanation of the correspondence is exploited nowadays in diverse mapping tools and code generators.
The figure shows two grammars for the same expression language taken from the paper. The first grammar is a conventional context-free grammar in terms of style, whereas the second grammar is restructured to be in an explicitly OO-enabled form. That is, an object model with single inheritance could be derived from the second grammar directly.
4.2 Hughes95

Hughes95 – Data

Citation
[15]

Title
The Design of a Pretty-printing Library

Online URL

Required concepts
functional programming

Provided concepts
pretty printing, combinator library

Annotation
Pretty printing is clearly an important form of language processing. This is not the first paper on a declarative and compositional approach to pretty printing; it stands out though with a very accessible presentation explaining the design and implementation of a (Haskell-based) combinator library for pretty printing. This library can be viewed as providing a simple embedded language for pretty printing.
Hughes95 – Illustration

\[
\text{Node "foo" (Node "baz" Leaf Leaf) (Node "foobaz" Leaf Leaf)}
\]

‘Ugly’ term

\[
\text{Node "foo" (Node "baz" Leaf Leaf) (Node "foobaz" Leaf Leaf)}
\]

Pretty term

\[
\begin{align*}
pp :: \text{Tree} & \rightarrow \text{Doc} \\
pp \text{ Leaf} & = \text{text "Leaf"} \\
pp \ (\text{Node } s \ l \ r) & = \text{text ("Node " ++ s)} \triangleright \text{sep [pp’} \ l, pp’ r] \\
pp’ \text{ Leaf} & = pp \text{ Leaf} \\
pp’ \ t & = \text{text ("("} \triangleright pp \ t \triangleright \text{")"}
\end{align*}
\]

Pretty printing function

The figure shows snippets (two Haskell terms and one Haskell function) taken from the paper. The figure illustrates pretty printing for binary trees with a string as info at each fork (i.e., non-leaf) node. The pretty-printed term uses line breaks and indentation for prettiness. The pretty printing function maps trees to documents; see the reference to the \textit{Doc} type. Pretty printer combinators are used; see ‘sep’ for example.
4.3  Reynolds98

Reynolds98 – Data

Citation
[28]

Title
Definitional Interpreters for Higher-Order Programming Languages

Online URL

Note
This paper originally appeared as [27].

Required concepts
semantics

Provided concepts
interpreter, continuation

Annotation
The paper discusses the use of interpreters as definitions of languages. There are the notions of defining and defined language (similar to what is also called elsewhere meta and object language). The paper analyzes possible differences between the interpreter-based definition and the formal or informal definition. The paper also discusses different styles of interpreter definition, e.g., a less insightful meta-circular interpreter for a higher-order language versus a first-order interpreter for the same defined language. The issue of application-order dependence is analysed and addressed with continuations.
Reynolds98 – Illustration

\[ \text{eval} = \lambda(r, e). \]
\[ (\text{const?}(r) \rightarrow \text{evcon}(r), \]
\[ \text{var?}(r) \rightarrow e(r), \]
\[ \text{appl?}(r) \rightarrow (\text{eval}(\text{opr}(r), e))(\text{eval}(\text{opnd}(r), e)), \]
\[ \text{lambda?}(r) \rightarrow \text{evlambda}(r, e), \]
\[ \text{cond?}(r) \rightarrow \text{if eval}(\text{prem}(r), e) \]
\[ \text{then eval}(\text{conc}(r), e) \text{ else eval}(\text{altr}(r), e), \]
\[ \text{letrec?}(r) \rightarrow \text{letrec } e' = \]
\[ \lambda x. \text{if } x = \text{dvar}(r) \text{ then evlambda}(\text{dexp}(r), e') \text{ else } e(x) \]
\[ \text{in eval}(\text{body}(r), e') \]
\[ \text{evlambda} = \lambda(\ell, e). \lambda a. \text{eval}(\text{body}(\ell), \text{ext}(\text{fp}(\ell), a, e)) \]
\[ \text{ext} = \lambda(z, a, e). \lambda x. \text{if } x = z \text{ then } a \text{ else } e(x). \]

The figure, taken from the paper, shows a meta-circular interpreter for (in) a simple functional language with lambdas, constants, conditionals, and recursive let.
4.4 SirerB99

SirerB99 – Data

Citation
[30]

Title
Using production grammars in software testing

Online URL

Required concepts
software engineering

Provided concepts
grammar-based testing

Annotation
The paper shows how grammar-based test-data generation and an accompanying methodology of testing may be highly effective and scalable for testing language-based software, in fact, the Java Virtual Machine. Previous publications on grammar-based testing mainly focused on compiler testing. The paper relies on a domain-specific language lava for specifying grammars from which to generate test data – bytecode, in this case. The generated test data is used for stress testing the JVM verifier and also for comparative testing of different verifiers.
Our attempts to create test cases manually were soon overwhelmed, and we sought a testing scheme that possessed the following properties:

- **Automatic**: Testing should proceed without human involvement, and therefore be relatively cheap. The technique should be easy to incorporate into nightly regression testing.
- **Complete**: Testing should generate numerous test cases that cover as much of the functionality of a virtual machine as possible. It should also admit a metric of progress that correlates with the amount of assurance in the virtual machine being tested.
- **Conservative**: Bad Java bytecodes should not be allowed to pass undetected through the bytecode verifier, and incorrectly executed instructions in the compiler or interpreter should be detected.
- **Well-structured**: Examining, directing, checkpointing and resuming verification efforts should be simple. Error messages should be descriptive; that is, it should be easy for a programmer to track down and fix a problem.
- **Efficient**: Testing should result in a high-confidence Java virtual machine within a reasonable amount of time.

The rest of this paper describes our experience with lava aimed at achieving these goals.

### 3. Lava and Grammar-based Test Generation

Our approach to test generation is to use an automated, well-structured process driven by a production grammar. A production grammar is the opposite of a regular parsing grammar in that it produces a program (i.e., all terminals, or tokens) starting from a high-level description (i.e., a set of non-terminals). The composition of the generated program reflects the restrictions placed on it by the production grammar. Figure 1 illustrates the high-level structure of the test generation process. A generic code-generator-generator parses a Java bytecode grammar written in lava and emits a specialized code-generator. The code-generator is a state machine that in turn takes a seed as input and applies the grammar to it. The seed consists of the high-level description that guides the production process. Running the code-generator on a seed produces test cases in Java bytecode that can then be used for testing.

The figure, taken from the paper, carries the following caption (in the paper): The structure of the test generation process. A code-generator-generator parses a production grammar, generates a code-generator, which in turn probabilistically generates test cases based on a seed.
4.5  Sheard01

Sheard01 – Data

Citation

[29]

Title
Accomplishments and Research Challenges in Meta-programming

Online URL

Required concepts
functional programming, metaprogramming

Provided concepts
taxonomy of metaprogramming, program representation, quasi-quotiation,
intensional analysis, staged computation, MetaML

Annotation
The paper provides a (possibly outdated) overview over meta-programming
with focus on the functional approach towards program representation,
code generation, and intensional code analysis. The paper aims to pro-
vide a taxonomy of metaprogramming and it discusses problems in
metaprogramming in a systematic and illustrative manner. MetaML
is shortly introduced as a particular metaprogramming language. The
paper brings up research challenges related to, e.g., dependent typing.
Sheard01 – Illustration

```ocaml
-| fun power_gen m =  
  let fun f n x = if n = 0 then <1> else <"x * ~/ (f (n-1) x)>  
  in <let fun power x = ~/ (f m x)> in power end> end;  
val power_gen = fn : int -> <int -> int>

-| val power_code = power_gen 3;
```

The figure, taken from the paper, shows the MetaML-based definition of a staged exponentiation function. The `power_gen` function describes the code generation for the n-th power. The `power_code` value holds the code for the 3rd power. The `power_fun` function is the function for said code, which we can ultimately apply.
4.6 KurtevBA02

KurtevBA02 – Data

Citation
[18]

Title
Technological spaces: An initial appraisal

Online URL
http://eprints.eemcs.utwente.nl/10206/01/0363TechnologicalSpaces.pdf

Required concepts
model driven engineering

Provided concepts
technological space

Annotation
As suggested by the title, this is the record of the introduction of the technological space notion. Several spaces are identified and discussed: abstract/concrete syntaxes, database management systems, XML, ontology engineering, and MDA. The megamodel underlying the spaces is discussed and instantiated for some spaces. The need for and the role of bridges between the spaces is explained. See [3] for another, more recent description of technological spaces.
The figure, taken from the paper, shows five technological spaces and bridges between them.
4.7 **Thomas03**

**Thomas03 – Data**

**Citation**

[31]

**Title**

The Impedance Imperative, Tuples + Objects + Infosets = Too Much Stuff!

**Online URL**


**Required concepts**

data programming

**Provided concepts**

impedance mismatch

**Annotation**

The paper (a column, in fact) takes a critical look at data programming—specifically in the sense of CRUD (Create, Read, Update, Delete). The discussion covers indexed files, SQL and database access APIs, object-oriented databases, modern wrapping/mapping-based approaches (e.g., object/relational mapping). The column identifies various problems with data programming: diversity of data modeling and CRUD programming options and the practical need to mix them, difficulties of integrating different type systems and data query/transformation languages, proprietary developments, performance issues, and complexity of support technologies. The discussion also briefly touches some contenders that may address some of the problems. The paper may be a good starting point to look for technical publications on the topic.
The figure, taken from Wikipedia, obviously shows the Bermuda triangle. While working with Erik Meijer on [20, 21], I picked up his intuition that data programming (because of the impedance mismatch) is essentially like operating in the Bermuda triangle. That is, data may disappear, if we allow this exaggeration. Just replace Bermuda, Florida, and Puerto Rico by XML, relational databases, and objects. (The idea of a triangle is an understatement because there are, of course, more competitors, e.g., Cobol and ontologies.)
4.8  *Hainaut06*

*Hainaut06 – Data*

**Citation**  
[11]

**Title**  
The Transformational Approach to Database Engineering

**Online URL**  
http://link.springer.com/chapter/10.1007%2F11877028_4

**Required concepts**  
entity-relationship model, relational database

**Provided concepts**  
schema normalization, logical design, schema integration, view derivation, schema equivalence, data conversion, data reverse engineering, schema optimization, data access wrapper generation

**Annotation**  
The paper describes fundamental and practical aspects of database transformation techniques. In particular, the notion of transformation is developed in combination with the correctness and reversibility properties.
The figure, taken from the paper, shows a particular transformation rule. Quoting from the paper “Transforming an is-a hierarchy into one-to-one relationship types and conversely. The exclusion constraint (excl:s.C,r.B) states that an A entity cannot be simultaneously linked to a B entity and a C entity. It derives from the disjoint property (D) of the subtypes”
4.9 \textit{HappelS06}

\textit{HappelS06 – Data}

\textbf{Citation}

\cite{12}

\textbf{Title}

Applications of Ontologies in Software Engineering

\textbf{Online URL}

https://km.aifb.kit.edu/ws/swese2006/final/happel_full.pdf

\textbf{Required concepts}

software engineering, ontology

\textbf{Provided concepts}

analysis, design, requirements engineering, component reuse, implementation, modeling, documentation, semantic middleware, semantic web service, maintenance, testing

\textbf{Annotation}

This paper takes an inventory of applications (usage categories) of ontologies in software engineering. It is rich in pointing out the relevance and potential of ontologies in various contexts (e.g., lifecycle phases) in software engineering.
Applications of Ontologies in Software Engineering

Second, we look at the kind of knowledge the ontology actually compromises. Here, we distinguish between the problem domain that the software system tries to tackle itself, and infrastructure aspects to make the software or its development more convenient. Putting these two dimensions together, we end up with the matrix in figure 1. We see four basic areas there:

- **Ontology-driven development (ODD)** subsumes the usage of ontologies at development time that describe the problem domain itself. Prime example are the approaches in the context of MD, presented in sec. 2.2.1.

- **Ontology-enabled development (OED)** also uses ontologies at development time, but for supporting developers with their tasks. For example, component search (sec. 2.1.2) or problem-solving support (sec. 2.4.1) can be put in here.

- **Ontology-based architectures (OBA)** use an ontology as a primary artifact at run-time. The ontology makes up a central part of the application logic. Business rule approaches are an example for this kind of application.

- **Ontology-enabled architectures (OEA)** finally, leverage ontologies to provide infrastructure support at the run-time of a software system. An example are semantic web services, where ontologies add a semantic layer on top of the existing web service descriptions, adding functionality for the automatic discovery, matching and composition of service-based workflows.

Although the four clusters seem to be quite distinct on first glance, there may be overlaps in some application areas. In particular, the classification scheme does not make any statement about clustering within or between the categorization groups. Indeed, in order to make the case for the large-scale reusability of ontologies, it is crucial to provide evidence for a broad range of applications. So one specific ontology might be useful in several of the described dimensions in parallel.

The figure is taken from the paper. Different roles of ontologies in the context of software engineering are identified along two axes. Legend of acronyms used: Ontology-driven development (ODD), Ontology-enabled development (OED), Ontology-based architectures (OBA), Ontology-enabled architectures (OEA).
4.10  Bezivin06

Bezivin06 – Data

Citation
[3]

Title
Model Driven Engineering: An Emerging Technical Space

Online URL
http://link.springer.com/chapter/10.1007%2F11877028_2

Required concepts
software development

Provided concepts
technological space, model driven engineering, model transformation, metamodeling

Annotation
The paper describes the basic principles and practical characteristics of model driven engineering (MDE). The technological space notion (see also [18]) is used to organize much of the description. In particular, MDE is also compared to other technological spaces. The key notions of metamodeling and model transformation are illustrated. Various technologies and standards are placed in context, e.g., EMF and ATL.
needs injection from its TS to the model engineering TS. The need for extraction is also quite important: many existing tools do not read XMI. A simple example is the Java compiler. What we need here is code generation, which may be seen as a specific case of model extraction. Many other TSs require both injectors and extractors: database systems provide another example in which database schemes have to be generated from model definitions.

5.6 Conclusions

What appear in this presentation are the high complementarities between all four presented functional blocks (ATL, AMW, AM3, and ATP). There are plenty of applications that make use of these four kinds of functionalities at the same time.

6 Conclusions

We have presented in this paper our definition of MDE basic principles and our view of an MDE implementation architectural style. The basic principle on which this work is based (Models as first class entities) is common to many current research communities (Model Management, Model Integrated Computing, etc.) and similar goals and means may be found in other TSs. This is summarized in Fig. 18.

![Diagram](image)

The figure, taken from the conclusion of the paper, on the left, highlights two important relations involved in MDE—the ‘isRepresentedBy’ relation that some thing (perhaps a model) is represented by a model and the ‘conformsTo’ relation related to metamodeling. On the right, the progression from real-world entities, through models and metamodels, up to metametamodels is megamodeled.
4.11  BezivinBFGJKKP06

BezivinBFGJKKP06 – Data

Citation
[4]

Title
A Canonical Scheme for Model Composition

Online URL

Required concepts
metamodeling

Provided concepts
model composition, model weaving, Glue Generator Tool, Epsilon Merging Language, Atlas Model Weaver

Annotation
The paper surveys different methods and tools for model composition. It also analyzes composition scenarios and assesses them, for example, in terms of degree of feasible automation. Further, general requirements for model composition tools are postulated and the degree of tool support is considered for existing technologies (at the time of writing).
The figure, taken from the paper’ section on the Atlas Model Weaver, shows the schema (say, megamodel) of models and metamodels involved in a weaving (i.e., composition) situation together with the required conformance relationships.
4.12  *Bravenboer TV06*

**Bravenboer TV06 – Data**

**Citation**

[5]

**Title**

Declarative, formal, and extensible syntax definition for AspectJ

**Online URL**

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.7867

**Required concepts**

parsing, AspectJ

**Provided concepts**

scannerless parsing, Generalized LR parsing

**Annotation**

The paper is a showcase of using generalized LR parsing in the implementation of frontends. While the underlying theory has been published elsewhere, the paper is nevertheless suitable for studying the basics of the method. AspectJ is picked as the target of a case study because parsing AspectJ challenges more conventional parsing techniques. We mention, in passing, another recent paper on generalized LR parsing [8].
In the figure, taken from the paper, the non-trivial issue of state maintenance in a scanner-based frontend for AspectJ is described at a higher level of abstraction. Depending on the context of parsing (Java, AspectJ, Pointcut), the scanner needs to work differently. In scannerless implementation, such extra effort is not needed.
4.13 AlvesV09

AlvesV09 – Data

Citation
[1]

Title
A Case Study in Grammar Engineering

Online URL

Required concepts
software engineering, parsing, metrics

Provided concepts
grammar engineering, grammar recovery, grammar metrics, grammar testing, grammar versioning

Annotation
The ‘Grammarware Agenda’ [25] properly established the terms grammar engineering (and grammarware engineering). The present paper presents a study that involves several areas of grammarware engineering. The study is concerned with the development of a VDM-SL grammar for actual parsing from its ISO standard language reference. The study involves grammar transformation (recovery), testing, metrics, and version management.
Table 1. Grammar metrics for the three release versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>term</th>
<th>var</th>
<th>mcc</th>
<th>avs-n</th>
<th>avs-p</th>
<th>hal-e</th>
<th>timp</th>
<th>clev</th>
<th>nslev</th>
<th>dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td></td>
<td>138</td>
<td>161</td>
<td>234</td>
<td>4.4</td>
<td>2.3</td>
<td>55.4</td>
<td>1%</td>
<td>34.9</td>
<td>4</td>
</tr>
<tr>
<td>disambiguated</td>
<td></td>
<td>138</td>
<td>118</td>
<td>232</td>
<td>6.4</td>
<td>2.8</td>
<td>61.1</td>
<td>1.5%</td>
<td>43.9</td>
<td>4</td>
</tr>
<tr>
<td>refactored</td>
<td></td>
<td>138</td>
<td>71</td>
<td>232</td>
<td>10.4</td>
<td>3.3</td>
<td>68.2</td>
<td>3%</td>
<td>52.6</td>
<td>3</td>
</tr>
</tbody>
</table>

The initial number of 161 non-terminals (VAR) decreases via 118 after disambiguation to 71 after refactoring. These numbers are the consequence of changes in grammar shape where non-terminals are replaced by their definition. In the disambiguation phase (43 non-terminals removed), such non-terminal inlining (unfolding) was performed to make formulation of the disambiguation information possible, or easier. For instance, after inlining, simple associativity attributes would suffice to specify disambiguation, while without inlining more elaborate reject productions might have been necessary. In the refactoring phase (47 non-terminals removed), the inlinings performed were mainly removals of injections. These were performed to make the grammar easier to read, more concise, and suitable for creation of ASTs closer to the abstract syntax specification in the standard.

The value of the McCabe cyclometric complexity metric decreases by 2 during disambiguation, meaning that we eliminated two paths in the flow graph of the grammar. This was caused by refactoring the syntax of product types and union types in similar ways. The reason for this refactoring during the disambiguation phase was to make disambiguation easier. In case of product types, the following two production rules:

\[\text{ProductType} \rightarrow \text{Type} \]
\[\{ \text{Type} \ast \}^{2+} \rightarrow \text{ProductType} \]

were replaced by a single one:

\[\text{Type} \ast \text{Type} \rightarrow \text{Type} \]

The figure, taken from the paper, shows the development of different grammar metrics over time. The timeline is defined by the commits of the grammar, as it was changed over time to complete the recovery process and to otherwise develop the parser. Test coverage also drives this process.
4.14 Wachsmuth09

Wachsmuth09 – Data

Citation
[35]

Title
A Formal Way from Text to Code Templates

Online URL
http://link.springer.com/chapter/10.1007%2F978-3-642-00593-0_8

Required concepts
metamodeling, formal semantics

Provided concepts
template instantiation

Annotation
The paper addresses the problem of unsafe template instantiation; see [13] for a description of the problem. Both papers share the overall line of attack: adaptation of a language’s metamodel (syntax) so that template-instantiation concepts are made available in a systematic way. The present paper stands out by deploying techniques of programming language theory (operational semantics and type systems) as well as grammar adaptation based on appropriate transformation operators in the tradition of [19].
The figure, taken from the paper, shows a transformation script which loops over symbols of the underlying language grammar and includes additional productions in a systematic manner so that concepts for template instantiation are made available.
4.15 Moody09

Moody09 – Data

Title
The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering

Online URL

Required concepts
visual language

Provided concepts

Annotation
We quote from the paper: “This paper defines a set of principles for designing cognitively effective visual notations: ones that are optimised for human understanding and problem solving. Together these form a design theory, called the Physics of Notations as it focuses on the physical (perceptual) properties of notations rather than their logical (semantic) properties. The principles were synthesised from theory and empirical evidence from a wide range of fields and rest on an explicit theory of how visual notations communicate. They can be used to evaluate, compare, and improve existing visual notations as well as to construct new ones. The paper identifies serious design flaws in some of the leading SE notations together with practical suggestions for improving them. It also showcases some examples of visual notation design excellence from SE and other fields.”
Moody09 – Illustration

<table>
<thead>
<tr>
<th>PLANAR VARIABLES</th>
<th>RETINAL VARIABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Position</td>
<td>Shape</td>
</tr>
<tr>
<td>Vertical Position</td>
<td>Brightness</td>
</tr>
<tr>
<td></td>
<td>Size</td>
</tr>
<tr>
<td></td>
<td>Orientation</td>
</tr>
<tr>
<td></td>
<td>Colour</td>
</tr>
<tr>
<td></td>
<td>Texture</td>
</tr>
</tbody>
</table>

In the figure, taken from the paper and based on seminal work [2], different visual variables are listed. These variables can thought of as defining a set of primitives, we quote from the paper: “a visual alphabet—for constructing visual notations: graphical symbols can be constructed by specifying particular values for visual variables (e.g., shape = rectangle, colour = green). Notation designers can create an unlimited number of graphical symbols by combining the variables together in different ways.”
4.16  RenggliGN10

RenggliGN10 – Data

Citation
[26]

Title
Embedding Languages without Breaking Tools

Online URL
http://scg.unibe.ch/archive/papers/Reng10aEmbeddingLanguages.pdf

Required concepts
Smalltalk

Provided concepts
embedded language

Annotation
The paper describes an embedding approach for the implementation of domain-specific languages (DSLs). Specifically, DSLs are modeled as language extensions of the underlying host language. The approach addresses the challenge of providing the language extensions in a manner that they integrate well with the development tools of the host language. The paper presents the extensible system Helvetia which intercepts the compilation pipeline of the Smalltalk host language to seamlessly integrate language extensions. See [32] for another extensive discussion of language embedding.
5.1 Homogeneous Language Integration

Rules

- <parse> allows one to intercept the parsing of the source code. The result of a parser rule can be either a new source string (in case of a source-to-source transformation) or a Smalltalk AST (in which the original Smalltalk parser is skipped).
- <transform> is performed on the AST after parsing and before semantic analysis. It allows developers to apply arbitrary transformations on the AST. Furthermore, it is possible to change the default semantic analysis and instead perform a custom one.
- <attribute> is performed after symbol resolution and before bytecode generation. This makes it possible to perform transformations on the attributed AST as well.

Compilation errors are handled by the standard toolchain. Since all data passed from one step to the next carries information on its original source location.

The figure, taken from the paper, shows different interception options for realizing embedded languages in the Smalltalk-based Helvetia framework. A pidgin does not require a new parser, but the code needs to be transformed before the semantic analysis. A creole also requires a designated parser. An argot only affects the backend.
4.17  HeidenreichJSWB09

HeidenreichJSWB09 – Data

Citation

[13]

Title

Generating Safe Template Languages

Online URL

https://www.st.cs.uni-saarland.de/~boehme/paper/GPCE09.pdf

Required concepts

metamodeling

Provided concepts

template instantiation

Annotation

The paper addresses the problem of unsafe template instantiation. Such instantiation can be unsafe in the sense that string-level operations are performed at run-time and thus it is not obvious or known at design time whether the described instantiation will actually lead to syntactically correct output eventually. The proposed approach involves the addition of generic template-instantiation concepts to existing language definitions in a generic manner. Thus, template instantiation would only be described in terms of metamodel instantiation, thereby implying syntactically correctness. Similar problems are present in programming languages with a macro system.
The figure, taken from the paper, shows the metamodel for template-instantiation concepts, e.g., a conditional and a loop form. The idea is that these concepts can be specialized for the syntactic categories of the language that is to be extended with template concepts. Whether or not a conditional or a loop is allowed in a certain position also depends on the cardinalities of the model element in the position. There is a generic algorithm to weave the template support in a given metamodel for a language.
4.18  Cordy11

Cordy11 – Data

Citation
[7]

Title
Excerpts from the TXL Cookbook

Online URL
http://cs.queensu.ca/~cordy/Papers/JC_TXLCookbook_LNCS.pdf

Required concepts
software engineering

Provided concepts
source-code analysis, source-code transformation

Annotation
The paper captures some reusable knowledge of implementing software components for source-code analysis and transformation. While the paper is focused on TXL as the underlying transformation system, the overall approach to knowledge representation would also make sense for other systems. The following classes of problems are considered: parsing, restructuring, optimization, static analysis, and interpretation. The solutions to the problems are described in terms of ‘paradigms’ such as ‘Use sequences, not recursions’, ‘Preserve comments in output’, ‘Generate unique identifiers’. 
The figure, taken from the paper, shows a simple TXL rule and its effect on a parse tree. In fact, a binary addition on constants is evaluated, thereby contributing to expression simplification.
4.19  \textit{ErwigW12a}

\textit{ErwigW12a} – Data

Citation

[10]

Title

Semantics First! - Rethinking the Language Design Process

Online URL

http://web.engr.oregonstate.edu/~erwig/papers/SemanticsFirst_SLE11.pdf

Required concepts

functional programming

Provided concepts

language design

Annotation

The paper suggests a semantics-centric approach to language design as opposed to a more syntax-based one. Haskell is used as a metalanguage. General language operators are employed to adapt and grow sophisticated languages out of simple semantics concepts.
Figure 2: Schematic illustration of the steps in the semantics-driven design process and their relationships. The two steps "Domain Decomposition" and "Domain Modeling" taken together comprise the Semantic Modeling part of the design process. The Syntactic Design step can be further distinguished as Inter- and Intra-DSL Syntax Design.

The figure is taken from a book chapter [9] that was derived from the conference paper at hand. The semantics-driven DSL design process is summarized. The idea is that one performs domain decomposition on the semantic side; one associates small languages with domains through domain modeling, and one also performs syntactic design to build a full language from the small languages.
4.20  *CookL11*

*CookL11* – Data

**Citation**

[6]

**Title**

Tutorial on Online Partial Evaluation

**Online URL**

http://arxiv.org/abs/1109.0781v1

**Required concepts**

functional programming

**Provided concepts**

partial evaluation

**Annotation**

We quote from the abstract of the paper: “This paper is a short tutorial introduction to online partial evaluation. We show how to write a simple online partial evaluator for a simple, pure, first-order, functional programming language. In particular, we show that the partial evaluator can be derived as a variation on a compositionally defined interpreter. We demonstrate the use of the resulting partial evaluator for program optimization in the context of model-driven development.”
The figure, taken from the paper, shows a general, sufficiently interpreter function for the simulation of state machines. When provided with the actual description of the state machine, partial evaluation can speziale the function to specific (efficient) dispatch code that essentially represents the state machine as code.
4.21 Herrmannsdoerfer VW11

Herrmannsdoerfer VW11 – Data

Citation
[14]

Title
An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models

Online URL
https://www4.in.tum.de/~herrmama/publications/SLE2010_herrmannsdoerfer_catalog_coupled_operators.pdf

Required concepts
metamodeling

Provided concepts
coevolution

Annotation
The evolution of a language implies that its metamodel has to evolve. Further, in most cases, existing instances may also need to co-evolve. Operation-nased transformation has matured as an automated method of carrying out metamodel/model coevolution. The present paper collects a catalogue of operations on the grounds of a literature survey and case studies; it also organizes the operations along several dimensions.
In the remainder of the paper, we will use instance-preservation properties to identify co-adaptation scenarios. There are two cases where co-adaptation is necessary. First, a variation 'hints a co-adaptation. Second, partial instance-preservation might be extended to complete instance-preservation.

4 Transformational adaptation of MOF compliant metamodels

4.1 Overview

In this section, we present a transformation library for the stepwise adaptation of MOF compliant metamodels. The transformations separate semantics-preservation properties introduced in the last section. Thereby, we can distinguish three kinds of transformations. First, we identify transformations for semantics-preserving (by variation) refactoring. Second, introducing and increasing transformations assist metamodel construction. Finally, eliminating and decreasing transformations allow for metamodel destruction. Table 1 groups the transformations presented in this section by this classification. It also gives semantics-preservation properties and inverse transformations.

We give the transformations as QVT Relations [4]. Thereby, we use its graphical notation. In the remainder of this section, we discuss each transformation in detail. We start with constructors and accordant destructors. Since most transformations for refactoring rely on construction and destruction, they are presented subsequently.

The figure, taken from an earlier paper [34] by one of the authors of the paper at hand, lists some adaptation operators and classifies them in terms of their purpose (refactoring, construction, destruction) and their semantics preservation properties. The paper at hand compiles a much more extensive catalogue and engages in a richer classification.
4.22 MullerFBC12

MullerFBC12 – Data

Citation
[24]

Title
Modeling modeling modeling

Online URL

Required concepts
modeling, model driven engineering

Provided concepts
representation, theory of modeling

Annotation
The paper works towards a theory of modeling. There is a focus on the representation relation that is so central to modeling (in the sense that one thing represents another thing). In fact, different (canonical) kinds of representation relations are identified and organized in a corresponding metamodel. This foundational work is well positioned in the context of previous work on the foundations of modeling (and metamodelling).
Neither things nor representations of things are built in isolation. As said by Steinmüller, both exist for a given purpose, exhibit properties, are built for some given stakeholders.

Table 2: Variations of the $\mu$-relation, and graphical notation

<table>
<thead>
<tr>
<th>Kind</th>
<th>Intention</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>different</td>
<td><img src="https://example.com/different.png" alt="Different Intention" /></td>
<td>X and Y have totally different intentions. This usually denotes a shift in viewpoints.</td>
<td>X $\sim \mu$ Y</td>
</tr>
<tr>
<td>share</td>
<td><img src="https://example.com/share.png" alt="Share Intention" /></td>
<td>X and Y share some intention. X and Y can be partially represented by each other. The representation is both partial and extended.</td>
<td>X $\sim \mu$ Y</td>
</tr>
<tr>
<td>sub</td>
<td><img src="https://example.com/sub.png" alt="Sub Intention" /></td>
<td>The intention of X is a part of Y’s intention. Everything which holds for X makes sense in the context of Y. Y can be partially represented by X.</td>
<td>X $\mu$ Y</td>
</tr>
<tr>
<td>same</td>
<td><img src="https://example.com/same.png" alt="Same Intention" /></td>
<td>X and Y share the same intention. They can represent each other. This usually denotes a shift in linguistic conformance.</td>
<td>X $\mu$ Y</td>
</tr>
<tr>
<td>super</td>
<td><img src="https://example.com/super.png" alt="Super Intention" /></td>
<td>X covers the intention of Y; X can represent Y, but X has additional properties. It is an extended representation.</td>
<td>X $\mu$ Y</td>
</tr>
</tbody>
</table>

The figure, taken from the paper, shows variations on the $\mu$ relation. These variations are essentially based on differences with regard to the intention of things. Quoting from the paper: “The intention of a thing thus represents the reason why someone would be using that thing, in which context, and what are the expectations vs. that thing. It should be seen as a mixture of requirements, behavior, properties, and constraints, either satisfied or maintained by the thing.”
4.23  JezequelCDGR12

JezequelCDGR12 – Data

Citation
[16]

Title
Bridging the chasm between MDE and the world of compilation

Online URL
http://link.springer.com/article/10.1007%2Fs10270-012-0266-8

Required concepts
modelware, grammarware, compilation, MDE

Provided concepts
cross-fertilization

Annotation
The paper attempts a deeper comparison of the technological spaces [18] of modelware (MDE) and grammarware (specifically compiler construction). We quote: “To address the growing complexity of software systems, Model-Driven Engineering (MDE) leverages Domain Specific Languages (DSL) to define abstract models of systems and automated methods to process them. Meanwhile, compiler technology mostly concentrates on advanced techniques and tools for program transformation. For this, it has developed complex analyses and transformations (from lexical and syntaxic to semantic analyses, down to platform-specific optimizations). These two communities appear today quite complementary and are starting to meet again in the Software Language Engineering (SLE) field.”
### JezequelCDGR12 – Illustration

<table>
<thead>
<tr>
<th>MDE shortcomings</th>
<th>Compilation solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increasing need for parsing tools due to increase in number of DSLs</td>
<td>Efficient parsing and parser generators</td>
</tr>
<tr>
<td>Platform Description Model</td>
<td>Capture of platform specific knowledge through dedicated descriptions</td>
</tr>
<tr>
<td>Tool efficiency and scalability</td>
<td>Sophisticated algorithms and heuristics</td>
</tr>
<tr>
<td>Increasingly complex model transformations</td>
<td>Know-how in sophisticated algorithms development and program transformation paradigms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compilation shortcomings</th>
<th>MDE solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRs contain more and more complex information, more and more complex IR processing</td>
<td>Complex data representation and Separation of Concerns</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Homogenization of software through generative approaches</td>
</tr>
<tr>
<td>Documentation</td>
<td>Metamodels as documentation</td>
</tr>
<tr>
<td>Error-prone and time consuming development tasks</td>
<td>Automation through metatools and metatooling</td>
</tr>
<tr>
<td>Ordering of the compilation pass</td>
<td>Design-by-Contract to limit possible choices to meaningful choices</td>
</tr>
</tbody>
</table>

The figure, taken from the paper, shows how the two spaces may mutually benefit from each other: shortcomings of one space may be addressed by adopting solutions known in the other space.
4.24 VolterSBK14

VolterSBK14 – Data

Citation
[33]

Title
Towards User-Friendly Projectional Editors

Online URL

Required concepts
parsing, IDE

Provided concepts
projectional editing

Annotation
The paper analyzes usability issues with projectional editing, but it actually may also serve a good reference for a definition and characterization of projectional editing as such. The discussion demonstrates key characteristics of projectional editing, e.g., the combination of notional styles and the use of composition techniques. We mention, in passing, to another recent paper on projectional editing [22]
Results

We identify 14 usability issues related to efficiently entering code (e.g., non-linear typing), selection and modification of code (e.g., introducing cross-tree parentheses), and integration with existing infrastructure (e.g., version control systems). Half of these issues can be addressed sufficiently, for instance, using code completion or expression-tree-refactoring support. Others require language- or notation-specific implementations, or cannot be mitigated conceptually.

Results of the survey show that developers perceive projectional editing as an efficient technique applicable in every-day work, while the effort of getting used to it is high. However, the survey also reveals weaknesses, such as the support for commenting, which is currently not addressed sufficiently in MPS.

2 Background

2.1 Parsing vs. Projection

In parser-based editors (ParEs), users type characters into a text buffer. The buffer is then parsed to check whether a sequence of characters conforms to a grammar. The parser builds a parse tree, and ultimately, an abstract syntax tree (AST), which contains the relevant structure of the program, but omits syntactic details. Subsequent processing (such as linking, type checks, and transformation) is based on the AST. Modern IDEs (re-)parse the concrete syntax while the user edits the code, maintaining an up-to-date AST in the background that reflects the code in the editor’s text buffer. However, even in this case, this AST is created by a parser-driven transformation from the source text.

A ProjE does not rely on parsers. As a user edits a program, the AST is modified directly. A projection engine uses projection rules to create a representation of the AST with which the user interacts, and which reflects the resulting changes. No parser-based transformation from concrete to abstract syntax involved here.

The figure, taken from the paper, illustrates the difference between parser-based editors (ParEs) and projectional editors (ProjEs). We quote from the paper: “In ParEs (left), users see and modify the concrete syntax. A parser constructs the AST. In ProjEs, users see and interact with the concrete syntax, but changes directly affect the AST. The concrete syntax is projected from the changing AST.”

VolterSBK14 – Illustration

The figure, taken from the paper, illustrates the difference between parser-based editors (ParEs) and projectional editors (ProjEs). We quote from the paper: “In ParEs (left), users see and modify the concrete syntax. A parser constructs the AST. In ProjEs, users see and interact with the concrete syntax, but changes directly affect the AST. The concrete syntax is projected from the changing AST.”
References


