Chapter 1
The Notion of a Software Language

JEAN-MARIE FAVRE.[1]

Abstract In this chapter, we characterize the notion of "software language" in a broad sense. We begin by setting out diverse examples of programming, modeling, and specification languages to cover a wide range of use cases of software languages in software engineering. Then, we classify software languages along multiple dimensions and describe the lifecycle of software languages, with phases such as language definition and implementation. Finally, we identify areas in software engineering that involve software languages in different ways, for example, software reverse engineering and software re-engineering.

[1] When the "Software Languages" community was formed around 2005–2007, Jean-Marie Favre was perhaps the key pillar and visionary and community engineer. His views and interests are captured very well in publications like these: [105, 104, 106, 100, 103].

R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7_1

1.1 Examples of Software Languages

In this book, we discuss diverse software languages; we may use them for illustrative purposes, and we may even define or implement them or some subsets thereof. For clarity, we would like to enumerate all these languages here in one place so that the reader will get an impression of the "language-related profile" of this book.

1.1.1 Real-World Software Languages

By "real-world language", we mean a language that exists independently of this book and is more or less well known. We begin with *programming languages* that will be used for illustrative code in this book. We order these languages loosely in terms of their significance in this book.

- *Haskell*[2]: The functional programming language Haskell
- *Java*[3]: The Java programming language
- *Python*[4]: The dynamic programming language Python

We will use some additional software languages in this book; these languages serve the purpose of specification, modeling, or data exchange rather than programming; we order these languages alphabetically.

- *ANTLR*[5]: The grammar notation of the ANTLR technology
- *JSON*[6]: The JavaScript Object Notation
- *JSON Schema*[7]: The JSON Schema language
- *XML*[8]: Extensible Markup Language
- *XSD*[9]: XML Schema Definition

Furthermore, we will refer to diverse software languages in different contexts, for example, for the purpose of language classification in Section 1.2; we order these languages alphabetically.

- *Alloy*[10]: The Alloy specification language
- *CIL*[11]: Bytecode of .NET's CLR

---

[2] Haskell language: https://www.haskell.org/

[3] Java language: https://en.wikipedia.org/wiki/Java_(programming_language)

[4] Python language: https://www.python.org/

[5] ANTLR language: http://www.antlr.org/

[6] JSON language: https://en.wikipedia.org/wiki/JSON

[7] JSON Schema language: http://json-schema.org/

[8] XML language: https://en.wikipedia.org/wiki/XML

[9] XSD language: https://en.wikipedia.org/wiki/XML_Schema_(W3C)

[10] Alloy language: http://alloy.mit.edu/alloy/

[11] CIL language: https://en.wikipedia.org/wiki/Common_Intermediate_Language

- *Common Log Format*[12]: The NCSA Common log format
  DocBook[13]: The DocBook semantic markup language for documentation
- *FOAF*[14]: The friend of a friend ontology
- *INI file*[15]: The INI file format
- *Java bytecode*[16]: Bytecode of the JVM
- *make*[17]: The make tool and its language
- *OWL*[18]: Web Ontology Language
- *Prolog*[19]: The logic programming language Prolog
- *QTFF*[20]: QuickTime File Format
- *RDF*[21]: Resource Description Framework
- *RDFS*[22]: RDF Schema
- *Scala*[23]: The functional OO programming language Scala
- *Smalltalk*[24]: The OO reflective programming language Smalltalk
- *SPARQL*[25]: SPARQL Protocol and RDF Query Language
- *UML*[26]: Unified Modeling Language
- *XPath*[27]: The XML path language for querying
- *XSLT*[28]: Extensible Stylesheet Language Transformations

1.1.2 Fabricated Software Languages

In this book, we "fabricated" a few software languages: these are small, idealized languages that have been specifically designed and implemented for the purposes of the book, although in fact these languages are actual or de facto subsets of real-world software languages. The language names are typically acronyms with expansions hinting at the nature of the languages. Language definitions of language-based

[12] Common Log Format language: https://en.wikipedia.org/wiki/Common_Log_Format
[13] DocBook language: https://en.wikipedia.org/wiki/DocBook
[14] FOAF language: http://semanticweb.org/wiki/FOAF.html
[15] INI file language: https://en.wikipedia.org/wiki/INI_file
[16] Java bytecode language: https://en.wikipedia.org/wiki/Java_bytecode
[17] make language: https://en.wikipedia.org/wiki/Make_(software)
[18] OWL language: https://en.wikipedia.org/wiki/Web_Ontology_Language
[19] Prolog language: https://en.wikipedia.org/wiki/Prolog
[20] QTFF language: https://en.wikipedia.org/wiki/QuickTime_File_Format
[21] RDF language: https://www.w3.org/RDF/
[22] RDFS language: https://www.w3.org/TR/rdf-schema/
[23] Scala language: https://en.wikipedia.org/wiki/Scala_(programming_language)
[24] Smalltalk language: https://en.wikipedia.org/wiki/Smalltalk
[25] SPARQL language: https://en.wikipedia.org/wiki/SPARQL
[26] UML language: https://en.wikipedia.org/wiki/Unified_Modeling_Language
[27] XPath language: https://en.wikipedia.org/wiki/XPath
[28] XSLT language: https://en.wikipedia.org/wiki/XSLT

software components are available for these languages from the book's repository.[29] The footnotes in the following list link to the repository locations for the languages.

- *BAL*: *B*asic *A*ssembly *L*anguage
- *BFPL*: *B*asic *F*unctional *P*rogramming *L*anguage
- *BGL*: *B*asic *G*rammar *L*anguage
- *BIPL*: *B*asic *I*mperative *P*rogramming *L*anguage
- *BML*: *B*inary *M*achine *L*anguage
- *BNL*: *B*inary *N*umber *L*anguage
- *BSL*: *B*asic *S*ignature *L*anguage
- *BTL*: *B*asic *T*APL *L*anguage
- *BL*: *B*uddy *L*anguage
- *EFPL*: *E*xtended *F*unctional *P*rogramming *L*anguage
- *EGL*: *E*xtended *G*rammar *L*anguage
- *EIPL*: *E*xtended *I*mperative *P*rogramming *L*anguage
- *EL*: *E*xpression *L*anguage
- *ESL*: *E*xtended *S*ignature *L*anguage
- *FSML*: *F*inite *S*tate *M*achine *L*anguage
- *MML*: *M*eta *M*odeling *L*anguage
- *TLL*: *T*yped *L*ambda *L*anguage
- *Text*: The "language" of text (such as Unicode 8.0 strings)
- *ULL*: *U*ntyped *L*ambda *L*anguage

In the rest of this section, we quickly introduce some of these languages, thereby providing a first indication of the diversity of language aspects covered by the book.

***B*inary *N*umber *L*anguage (BNL)** A trivial language of binary numbers with an intended semantics that maps binary to decimal values.

***B*asic *T*APL *L*anguage (BTL)** A trivial expression language in reference to the TAPL textbook (Types and programming languages [210]).

***B*uddy *L*anguage (BL)** A trivial language for modeling persons in terms of their names and buddy relationships.

***B*asic *F*unctional *P*rogramming *L*anguage (BFPL)** A really simple functional programming language which is an actual syntactic subset of the established programming language *Haskell*.

***B*asic *I*mperative *P*rogramming *L*anguage (BIPL)** A really simple imperative programming language which is a de-facto subset of the established programming language C.

***F*inite *S*tate *M*achine *L*anguage (FSML)** A really simple language for behavioral modeling which is variation on statecharts of the established modeling language UML.

***B*asic *G*rammar *L*anguage (BGL)** A specification language for concrete syntax, which can also be executed for the purpose of parsing; it is a variation on the established Backus-Naur form (BNF).

[29] http://github.com/softlang/yas

1.1.2.1 BNL: A Language of Binary Numbers

We introduce *BNL* (*B*inary *N*umber *L*anguage). This is a trivial language whose elements are essentially the binary numbers. Here are some binary numbers and their associated "interpretations" as decimal numbers:

- **0**: 0 as a decimal number;
- **1**: 1 as a decimal number;
- **10**: 2 as a decimal number;
- **11**: 3 as a decimal number;
- **100**: 4 as a decimal number;
- **101**: 5 as a decimal number;
- **101.01**: 5.25 as a decimal number.

Thus, the language contains integer and rational numbers – only positive ones, as it happens. BNL is a trivial language that is nevertheless sufficient to discuss the most basic aspects of software languages such as *syntax* and *semantics*. A syntax definition of BNL should define valid sequences of digits, possibly containing a period. A semantics definition of BNL could map binary to decimal numbers. We will discuss BNL's abstract syntax in Chapter 3 and the concrete syntax in Chapter 6.

1.1.2.2 BTL: An Expression Language

We introduce *BTL* (*B*asic *T*APL *L*anguage). This is a trivial language whose elements are essentially expressions over natural numbers and Boolean values. Here is a simple expression:

```
pred if iszero zero then succ succ zero else zero
```

The meaning of such expressions should be defined by expression evaluation. For instance, the expression form iszero *e* corresponds to a test of whether *e* evaluates to the natural number zero; evaluation of the form is thus assumed to return a Boolean value. The expression shown above evaluates to zero because iszero zero should compute to true, making the if-expression select the then-branch succ succ zero, the predecessor of which is succ zero.

An interpreter of BTL expressions should recursively evaluate BTL expression forms. BTL is a trivial language that is nevertheless sufficient to discuss basic aspects of interpretation (Chapter 5), semantics (Chapter 8), and type systems (Chapter 9).

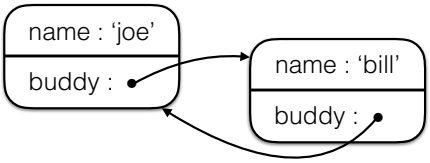

Fig. 1.1 Illustrative graph of buddy relationships.

1.1.2.3 BL: A Language for Buddy Relationships

We introduce *BL* (*B*uddy *L*anguage). This is a trivial language whose elements are essentially graphs of persons with their names and buddy relationships. Figure 1.1 shows an illustrative graph of buddy relationships between two persons; we leverage an ad hoc visual, concrete syntax.

BL is a trivial ontology-like language. It can be considered a trivial variation on FOAF, the "friend of a friend" ontology. Importantly, BL involves references in an essential manner. Thus, BL calls for a graph-based abstract syntax, whereas the other language examples given above (arguably) need only a tree-based abstract syntax. BL also involves an interesting constraint: a person must not be his or her own buddy. We will discuss BL as an example of graph-based abstract syntax in Chapter 3.

1.1.2.4 BFPL: A Functional Programming Language

We introduce *BFPL* (*B*asic *F*unctional *P*rogramming *L*anguage). Here is an illustration of BFPL – a sample program which defines the factorial function recursively and applies to an actual argument:

```
-- The factorial function
factorial :: Int -> Int
factorial x =
  if ((==) x 0)
    then 1
    else ((*) x (factorial ((-) x 1)))

-- Apply the function to 5
main = print $ factorial 5 -- Prints 120
```

(The execution of the program would print "120".) BFPL is a trivial language exercising basic *functional programming* concepts such as function application and recursive function definition. A semantics definition of BFPL should define expression evaluation, including parameter passing for function application. We will develop such a semantics in Chapter 8.

For what it matters, BFPL is a "small" syntactic subset of the established functional programming language *Haskell*. In fact, the sample shown is a valid Haskell program as is, and the Haskell semantics would agree on the output – 120 for the factorial of 5. BFPL was fabricated to be very simple. Thus, BFPL lacks many language constructs of Haskell and other real-world functional programming languages. For instance, BFPL does not feature higher-order functions and algebraic data types.

1.1.2.5 BIPL: An Imperative Programming Language

We introduce *BIPL* (*B*asic *I*mperative *P*rogramming *L*anguage). Here is an illustration of BIPL – a sample program which performs Euclidean division:

```
{
  // Sample operands for Euclidean division
  x = 14;
  y = 4;

  // Compute quotient q=3 and remainder r=2
  q = 0;
  r = x;
  while (r >= y) {
      r = r − y;
      q = q + 1;
  }
}
```

Division is applied to specific arguments x and y. The result is returned as the quotient q and the remainder r. The execution of the sample program would terminate with the variable assignment x=14, y=4, q=3, r=2.

BIPL is a trivial language exercising basic *imperative programming* concepts such as mutable variable, assignment, and control-flow constructs for sequence, selection, and iteration. For what it matters, BIPL is roughly a "small" syntactic subset of the established but much more complicated imperative programming language C. BIPL lacks many language constructs that are provided by C and other real-world imperative programming languages. For instance, BIPL does not feature procedures (functions) and means of type definition such as structs. Further, C requires declaration of variables, whereas BIPL does not. A semantics of BIPL should define statement execution. We will develop such a semantics in Chapter 8.

1.1.2.6 FSML: A Language for Finite State Machines

We introduce *FSML* (*FSM L*anguage, i.e., *F*inite *S*tate *M*achine *L*anguage). Figure 1.2 shows an illustrative FSM (finite state machine) which models the behavior of a turnstile or some sort of revolving door, as possibly used in a metro system. The FSM identifies possible states of the turnstile; see the nodes in the visual notation.

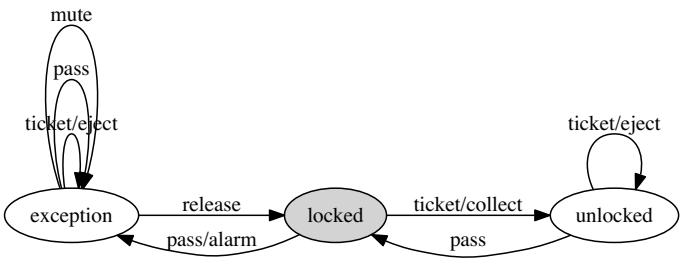

Fig. 1.2 A finite state machine for a turnstile.

The FSM also identifies possible transitions between states triggered by "events," possibly causing "actions"; see the edges in the visual notation.

These are these states in the turnstile FSM:

- *locked*: The turnstile is locked. No passenger is allowed to pass.
- *unlocked*: The turnstile is unlocked. A passenger may pass.
- *exception*: A problem has occurred and metro personnel need to intervene.

There are input symbols which correspond to the events that a user or the environment may trigger. There are output symbols which correspond to the actions that the state machine should perform upon a transition. These are some of the events and actions of the turnstile FSM:

- Event *ticket*: A passenger enters a ticket into the card reader.
- Event *pass*: A passenger passes through the turnstile, as noticed by a sensor.
- Action *collect*: The ticket is collected by the card reader.
- Action *alarm*: An alarm is turned on, thereby requesting metro personnel.

The meanings of the various transitions should be clear. Consider, for example, the transition from the source state "locked" to the target state "unlocked", which is annotated by "ticket/collect" to mean that the transition is triggered by entering a ticket and the transition causes ticket collection to happen.

FSML is a domain-specific modeling language (DSML). FSML supports *state-based modeling* of systems. The specification can be executed to simulate possible behaviors of a turnstile. The specification could also be used to generate a code skeleton for controlling an actual turnstile, as part of an actual metro system. FSML is a trivial language that can be used to discuss basic aspects of domain-specific language definition and implementation. For what it matters, languages for state-based behavior are widely established in software and systems engineering. For instance, the established modeling language UML consists, in fact, of several modeling languages; UML's state machine diagrams are more general than FSML. We will discuss FSML in detail in Chapter 2.

1.1.2.7 BGL: A Language for Context-Free Grammars

We introduce *BGL* (*B*asic *G*rammar *L*anguage). This language can be used to define the *concrete textual syntax* of other software languages. Thus, BGL gets us to the metalevel. Here is an illustration of BGL – a definition of the syntax of BNL – the language of binary numbers, as introduced earlier:

```
[number] number : bits rest ; // A binary number
[single] bits : bit ; // A single bit
[many] bits : bit bits ; // More than one bit
[zero] bit : '0' ; // The zero bit
[one] bit : '1' ; // The nonzero bit
[integer] rest : ; // An integer number
[rational] rest : '.' bits ; // A rational number
```

Each line is a grammar production (a rule) with the syntactic category (or the so-called nonterminal) to the left of ":" and its definition to the right of ":". For instance, the first production defines that a binary number consists of a bit sequence bits for the integer part followed by rest for the optional rational part. The right-hand phrases compose so-called terminals ("0", "1", and ".") and nonterminals (bit, bits, rest, and number) by juxtaposition. The rules are labeled, thereby giving a name to each construct.

BGL is a domain-specific modeling language in that it supports modeling (or specifying or defining) concrete textual syntax. One may "execute" BGL in different ways. Most obviously, one may execute a BGL grammar for the purpose of accepting or *parsing* input according to the syntax defined. BGL, like many other notations for syntax definition, is grounded in the fundamental formalism of *context-free grammars* (CFGs). BGL is a variation on BNF [21]. There exist many real-world notations for syntax definition [277]; they are usually more complex than BGL and may be tied to specific technology, for example, for parsing. We will develop BGL in detail in Chapter 6.

1.2 Classification of Software Languages

There are hundreds or even thousands of established software languages, depending on how we count them. It may be useful to group languages in an ontological manner. In particular, a *classification* of software languages (i.e., a language taxonomy) is a useful (if not necessary) pillar of a definition of "software language".

Wikipedia, which actually uses the term "computer language" at the root of the classification, identifies the following top-level classifiers:[30]

- data-modeling languages;
- markup languages;
- programming languages;
- specification languages;
- stylesheet languages;
- transformation languages.

Any such branch can be classified further in terms of constructs and concepts. For instance, in the case of programming languages, there exist textbooks on programming languages, programming paradigms, and programming language theory such as [199, 232], which identify constructs and concepts. There is also scholarly work on the classification of programming languages [20, 90] and the identification of language concepts and corresponding paradigms [258].

Several classes of software languages (other than programming languages) have been identified, for example, *model transformation languages* [75], *business rule modeling languages* [239], *visual languages* [46, 49, 190], and *architecture description languages* [192]. There is more recent work aimed at the classification of software languages (or computer languages) more broadly [13, 237, 3, 171]. The *101companies* project[31] [102, 101, 173, 166] is also aimed at a taxonomy of software languages, but the results are of limited use, at the time of writing.

In the remainder of this section, we classify software languages along different dimensions. A key insight here is that a single classification tree is insufficient. Multiple inheritance may be needed, or orthogonal dimensions may need to be considered separately.

1.2.1 Classification by Paradigm

When focusing on programming languages as a category of software languages, classification may be based on the *programming paradigm*. A paradigm is characterized by a central notion of programming or computing. Here is an incomplete, classical list of paradigms:

Imperative programming Assignable (updatable) variables and updatable (in-place) data structures and sequential execution of statements of operations on variables. Typically, procedural abstractions capture statements that describe control flow with basic statements for updates. We exercise imperative programming with the fabricated language BIPL (Section 1.1.2.5) in this book.

[30] We show Wikipedia categories based on a particular data-cleaning effort [171]. This is just a snapshot, as Wikipedia is obviously evolving continuously.

[31] http://101companies.org

Functional programming The application of functions models computation with compound expressions to be reduced to values. Functions are first-class citizens in that functions may receive and return functions; these are higher-order functions. We exercise functional programming with the fabricated language BFPL (Section 1.1.2.4) in this book – however, though higher-order functions are not supported.

Object-oriented (OO) programming An object is a capsule of state and behavior. Objects can communicate with each other by sending messages, the same message being implementable differently by different kinds of objects. Objects also engage in structural relationships, i.e., they can participate in whole–part and reference relationships. Objects may be constructed by instantiation of a given template (e.g., a class). *Java* and C# are well-known OO programming languages.

Logic programming A program is represented as a collection of logic formulae. Program execution corresponds to some kind of proof derivation. For instance, *Prolog* is a well-known logic programming language; computation is based on depth-first, left-to-right proof search through the application of definite clauses.

There exist yet other programming or computing notions that may characterize a paradigm, for example, message passing and concurrency. Many programming languages are, in fact, *multi-paradigm* languages in that they support several paradigms. For instance, *JavaScript* is typically said to be both a functional and an imperative OO programming language and a scripting language. Programming languages may be able to support programming according to a paradigm on the basis of some encoding scheme without being considered a member of that paradigm. For instance, in Java prior to version 8, it was possible to encode functional programs in Java, while proper support was added only in version 8.

Van Roy offers a rich discussion of programming paradigms [258]. Programming concepts are the basic primitive elements used to construct programming paradigms. Often, two paradigms that seem quite different (for example, functional programming and object-oriented programming) differ by just one concept. The following are the concepts discussed by Van Roy: record, procedure, closure, continuation, thread, single assignment, (different forms of) cell (state), name (unforgeable constant), unification, search, solver, log, nondeterministic choice, (different forms of) synchronization, port (channel), clocked computation. Van Roy identifies 27 paradigms, which are characterized as sets of programming concepts. These paradigms can be clearly related in terms of the concepts that have to be added to go from one paradigm to another.

1.2.2 Classification by Type System

Furthermore, languages may also be classified in terms of their typing discipline or type system [210] (or the lack thereof). Here are some important options for programming languages in particular:

Static typing The types of variables and other abstractions (e.g., the argument and result types of methods or functions) are statically known, i.e., without executing the program – this is at compile time for compiled languages. For instance, Haskell and Java are statically typed languages.

Dynamic typing The types of variables and other abstractions are determined at runtime. A variable's type is the type of the value that is stored in that variable. A method or function's type is the one that is implied by a particular method invocation or function application. For instance, Python is a dynamically typed language.

Duck typing The suitability of a variable (e.g., an object variable in object-oriented programming) is determined at runtime on the basis of checking for the presence of certain methods or properties. Python uses duck typing.

Structural typing The equivalence or subtyping relationship between types in a static typing setting is determined on the basis of type structure, such as the components of record types. Scala supports some form of structural typing.

Nominal typing The equivalence or subtyping relationship between types in a static typing setting is determined on the basis of explicit type names and declared relationships between them. Java's reference types (classes and interfaces including "extends" and "implements" relationships) commit to nominal typing.

1.2.3 Classification by Purpose

Languages may be classified on the basis of the *purpose* of the language (its usage) or its elements. Admittedly, the term "purpose" may be somewhat vague, but the illustrative classifiers in Table 1.1 may convey our intuition. We offer two views: the purpose of the language versus that of its elements; these two views are very similar.

Table 1.1 Classification by the purpose of language elements

Purpose (language)	**Purpose** (element)	**Classifier**	**Example**
Programming	Program	Programming language	Java
Querying	Query	Query language	XPath
Transformation	Transformation	Transformation language	XSLT
Modeling	Model	Modeling language	UML
Specification	Specification	Specification language	Alloy
Data representation	Data	Data format	QTFF (QuickTime file format)
Documentation	Documentation	Documentation language	DocBook
Configuration	Configuration	Configuration language	INI file
Logging	Log	Log format	Common Log Format
...	...	...	...

In some cases, owing to the ambiguity of natural language, we may even end up with (about) the same purpose for both views, language elements versus language. These classifiers are not necessarily disjoint. For instance, it may be hard to decide in all cases whether some artifact should be considered a model, a specification, or a program. The classifiers can also be further broken down. For instance, "data" may be classified further as "audio", "image", or "video" data; "models" may be classified further as "structural", "behavioral", or "architectural" models.

Let us apply this classification to the illustrative languages of Section 1.1. The basic functional and imperative languages (BFPL and BIPL) are programming languages. The languages for buddy relationships (BL) and finite state machines (FSML) are (domain-specific) modeling languages. The BNF-like grammar language BGL is a specification language. We may also characterize BGL as a syntax-modeling language. We may also characterize the languages for binary numbers (BNL) and buddy relationships (BL) as data formats or data representation languages.

In many cases, when speaking of purposes, we may also speak of domains and, more specifically, of problem or programming domains. For instance, (a) transformation may be considered as a purpose, as an artifact, and a domain. We discuss the domain notion in more detail below.

Exercise 1.1 (Another purpose for classification) [Intermediate level]
Study the classification of languages based on the references given earlier. For instance, you may study the classification of languages according to Wikipedia. Find another classifier to serve for classification based on purpose.

1.2.4 Classification by Generality or Specificity

There is a long-standing distinction between general-purpose (programming) languages (GPLs) versus domain-specific languages (DSLs). The term "domain" may be meant here in the sense of "problem" or "application" domain such as image manipulation, hardware design, or financial products; see [81, 157] for many examples. The term "domain" may also be meant in the sense of "programming" or "solution" domain such as XML, GUI, or database programming [219]. This distinction is not clear-cut; it also depends on the point of view. We do not make much use of this distinction in the sequel.

We quote from [81] to offer a definition of a DSL: "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." We also extract some distinguishing characteristics of GPLs and DSLs from [265]:

Domain Only DSLs have a relatively small and well-defined domain.
Language size GPLs are large. DSLs are typically small.
Turing completeness DSLs may not be Turing complete.
Lifespan GPLs live for years to decades. DSLs may live for months only.
Designed by GPLs are designed by gurus or committees. DSLs are designed by a few software engineers and domain experts.
Evolution GPLs evolve slowly. The evolution of DSLs is fast-paced.
Deprecation/incompatible changes This is almost impossible for GPLs; it is feasible and relatively common for DSLs.

DSLs, in turn, may be subdivided into domain-specific programming versus modeling languages; see [147] for an introduction to domain-specific modeling. However, we argue that the distinction between programming and modeling is somewhat vague, because many modeling languages end up being executable eventually and it is also not uncommon to use programming languages for modeling. DSLs may also be subdivided based on the style of implementation: *internal* versus *external DSLs* [265, 214, 94, 78]. An internal DSL is basically a DSL implementation inside another language (the so-called *host language*) – typically as a library inside a GPL, for example, *Parsec* [182] is implemented as a Haskell-based combinator library for parsing. The DSL user accesses the DSL by means of an API. Internal style may leverage metaprogramming-related facilities of the host language.

An external DSL is a language that is implemented in its own right, for example, *make* as a DSL (and a tool) for build management. There is also the notion of an *embedded DSL* [214], where the DSL's syntax and semantics are integrated into a host language. This may be achieved by the host language's ability to define the DSL syntax and to model the semantics by means of a mapping from DSL syntax to host language syntax.

There is also the related notion of *language composition*. That is, when multiple languages are integrated to be used together, then their semantics may need to be coordinated [141, 48, 95, 78]. Language composition is particularly relevant in the context of (domain-specific) modeling languages.

1.2.5 Classification by Representation

One may distinguish three fundamental representation options: strings versus trees versus graphs. That is, the terms "string", "tree", and "graph" hint at the fundamental structure of language elements. The terms are drawn from formal language theory. Accordingly, there are these language classifiers:

String language (See also "textual language" below.) Language elements are represented, viewed, and edited as strings, i.e., sequences of characters. A string language would be typically defined in terms of a string grammar in the sense of formal language theory, as supported, for example, by the BGL grammar nota-

tion of Section 1.1.2.7. Several of the illustrative languages of Section 1.1 were introduced as string languages.

Tree language (See also "markup language" below.) Language elements are represented, viewed, and edited as trees, for example, as XML trees or JSON dictionaries. A tree language is defined in terms of a suitable grammar or data modeling notation, for example, XSD in the case of XML. As it happens, we did not present any tree languages in Section 1.1.2.7, but we will discuss tree-based abstract syntax definitions later for some of the string languages that we have already seen. Tree languages play an important role in language implementation.

Graph language Language elements are represented, viewed, and edited as graphs, i.e., more or less constrained collections of nodes and edges. Appropriate grammar and data modeling notations exist for this case as well. The language BL for buddy relationships (Section 1.1.2.3) was introduced as a graph language and we hinted at a visual concrete syntax. A graph language may be coupled with a string or tree language in the sense of alternative representations of the same "conceptual" language. For instance, BL may be represented in a string-, tree-, or graph-based manner.

1.2.6 Classification by Notation

One may also distinguish languages in terms of notation; this classification is very similar to the classification by representation:

Textual (text) language This is essentially a synonym for "string language".

Markup language Markup, as in XML, is used as the main principle for expressing language elements. The use of markup is one popular notation for tree languages. With an appropriate semantics of identities, markup can also be used as a notation for graphs. Not every tree language relies on markup for the notation. For instance, JSON provides another, more dictionary-oriented notation for tree languages.

Visual (graphical) language A visual notation is used. The languages BL for buddy relationships (Section 1.1.2.3) and FSML for state-based modeling (Section 1.1.2.6) were introduced in terms of a visual notation.

1.2.7 Classification by Degree of Declarativeness

An (executable) language may be said to be (more or less) *declarative*. It turns out to be hard to identify a consensual definition of declarativeness, but this style of classification is nevertheless common. For instance, one may say that programs (or models) of a declarative language describe more the "what" than the "how". That is, a declarative program's semantics is not strongly tied to execution order.

Let us review the languages of Section 1.1:

Binary Number Language (BNL) A trivial language.

Buddy Language (BL) A trivial language.

Basic Functional Programming Language (BFPL) This language is "pure", i.e., free of side effects. Regardless of the evaluation order of subexpressions, complete evaluation of a main expression should lead to the same result – modulo some constraints to preserve termination. For instance, argument expressions of a function application could be evaluated in different orders without affecting the result. Thus, BFPL is a declarative programming language.

Basic Imperative Programming Language (BIPL) This language features imperative variables such that the execution order of statements affects the result of computation. Thus, BIPL is not a declarative programming language.

Finite State Machine Language (FSML) This language models finite states and event- and action-labeled transitions between states. Its actual semantics or execution order is driven by an event sequence. FSML would usually be regarded as a declarative (modeling) language.

Basic Grammar Language (BGL) This grammar notation defines sets of strings in a rule-based manner. Thus, BGL's most fundamental semantics is "declarative" in the sense that it is purely mathematical, without reference to any operational details. Eventually, we may attach a more or less constrained operational interpretation to BGL so that we can use it for efficient, deterministic parsing. Until that point, BGL would be usually regarded as a declarative (specification) language.

We may also consider subcategories of declarative languages such that it is emphasized how declarativeness is achieved. Two examples may suffice:

Rule-based language Programs are composed from rules, where a rule typically combines a condition and an action part. The condition part states when the rule is applicable; the action part states the implications of rule application. Some logic programming languages, for example, Prolog, can be very well considered to be rule-based languages. Some schemes for using functional programming, for example, in interpretation or program transformation, also adopt the rule-based approach. Event-driven approaches may also use rules with an additional "event" component, for example, the "event condition action" (ECA) paradigm, as used in active databases [88].

Constraint-based language Programs involve constraints as means of selecting or combining computations. These constraints are aggregated during program execution and constraint resolution is leveraged to establish whether and how given constraints can be solved. For instance, there exist various constraint-logic programming languages which enrich basic logic programming with constraints on sets of algebras for numbers [117].

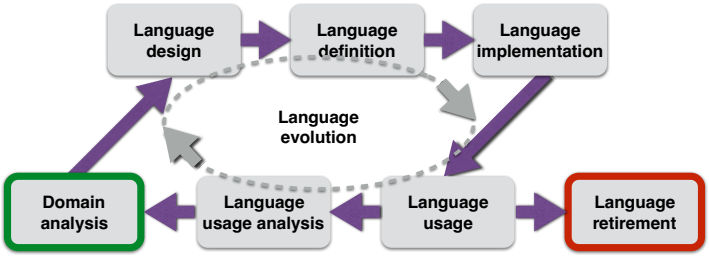

Fig. 1.3 The lifecycle of a software language. The nodes denote phases of the lifecycle. The edges denote transitions between these phases. The lifecycle starts with *domain analysis*. The lifecycle ends (theoretically) with *language retirement*. We may enter cycles owing to language evolution.

Exercise 1.2 (Classification of *make*) [Basic level]
Study the language used by the well-known make *utility and argue that the language is declarative and identify what subcategory of declarativeness applies.*

1.3 The Lifecycle of Software Languages

The notion of a software lifecycle can be usefully adopted for languages. That is, a language goes through a lifecycle, possibly iterating or skipping some phases; see Fig. 1.3. These phases are described in some detail as follows:

Domain analysis A domain analysis is required to discover the domain that is to be addressed by a new language. A domain analysis answers these questions: What are the central concepts in the domain? For instance, the central concepts are states and transitions between states for FSML, differential equations and their solution in a language for weather forecasts, or layouts and their rendering in a language for HTML/XML stylesheets. These concepts form the foundation of language design and for everything onwards. Arguably, no domain analysis is performed for general-purpose programming languages.

Language design The domain concepts are mapped, at the design level of abstraction, to language constructs and language concepts. The emerging language should be classified in terms of paradigm, degree of declarativeness, and other characteristics. A language may be presented as a composition of very specific language constructs as well as reusable language constructs, for example, for basic expressions or modules. The first samples are written so that a syntax emerges, and the transition to the phase of language definition has then begun.

Language definition The language design is refined into a language definition. Most notably, the syntax and semantics of the language are defined. Assuming

an executable language definition, a first language implementation (a proof of concept) is made available for experiments, so that the transition to the phases of language implementation and usage has begun.

Language implementation The language is properly implemented. Initially, a usable and efficient compiler or interpreter needs to provided. Eventually, additional *language processors* and tool support may be provided, for example, documentation tools, formatters, style checkers, and refactorings. Furthermore, support for an integrated development environment (IDE) may be implemented.

Language usage The language is used in actual software development. That is, language artifacts are "routinely" authored and a body of software artifacts acquire dependencies on the language. This is not explicitly modeled in Fig. 1.3, but the assumption is, of course, that the language implementation is continuously improved and new language processors are made available.

Language evolution Language definitions may revised to incorporate new language features or respond to experience with language usage. Obviously, changes to language definitions imply work on language implementations. Language changes may even break backward compatibility, in which cases these changes will necessitate migration of existing code in those languages.

Language usage analysis Language evolution and the systematic improvement of domain analysis as well as language design, definition, and implementation, may benefit from language usage analysis [155, 100, 172], as an empirical element of the lifecycle. By going through the lifecycle in cycles, the language may evolve in different ways. For instance, the language may be extended so that a new version becomes available, which again needs to be implemented and put to use.

Language retirement In practice, languages, once adopted, are rarely retired completely, because the costs and risks of retirement are severe impediments. Retirement may still happen in the narrow scope of projects or organizations. In theory, a language may become obsolete, i.e., there are no software artifacts left that depend on that language. Otherwise, *language migration* may be considered. That is, software artifacts that depend on a language are migrated (i.e., transformed manually or automatically) to another language.

Many aspects of these phases, with some particular emphasis on the lifecycle of DSLs are discussed in [133, 272, 273, 197, 214, 56, 98, 94, 78, 265, 229]. In the present book, the focus is on language definition and implementation; we are concerned only superficially with domain analysis, language design, evolution, and retirement.

1.3.1 Language Definition

Let us have a deeper look at the lifecycle phase of language definition. A language is defined to facilitate implementation and use of the language. There are these aspects of language definition:

Syntax The definition of the syntax consists of rules that describe the valid language elements which may be drawn from different "universes": the set of all strings (say, text), the set of all trees (of some form, e.g., XML-like trees), or the set of all graphs (of some form). Different kinds of formalisms may be used to specify the rules defining the syntax. We may distinguish *concrete* and *abstract syntax* – the former is tailored towards users who need to read and write language elements, and the latter is tailored towards language implementation. Abstract syntax is discussed in Chapters 3and 4. Concrete syntax is discussed in Chapters 6and 7.

Semantics The definition of semantics provides a mapping from the syntactic categories of a language (such as statements and expressions) to suitable domains of meanings. The actual mapping can be defined in different ways. For instance, the mapping can be defined as a set of syntax-driven inference rules which model the stepwise execution or reduction of a program; this is known as small-step operational semantics (Chapter 8). The mapping can also be applied by a translation, for example, by a model-to-model transformation in model-driven engineering (MDE).

Pragmatics The definition of the pragmatics explains the purpose of language concepts and provides recommendations for their usage. Language pragmatics is often defined only informally through text and samples. For instance, the pragmatics definition for a C-like language with arrays may state that arrays should be used for efficient (constant-time) access to indices in ordered collections of values of the same type. Also, arrays should be favored over (random-access) files or databases for as long as in-memory representation of the entire data structure is reasonable. In modeling languages for finite state machine (e.g., FSML), events proxy for sensors and actions proxy for actors in an embedded system.

Types Some languages also feature a *type system* as a part of the language definition. A type system provides a set of rules for assigning or verifying *types*, i.e., properties of language phrases, for example, different expression types such as "int" or "string" in a program with expressions. We speak of *type checking* if the type system is used to check explicitly declared types. We speak of *type inference* if the type system is used additionally to infer missing type declarations. A type system needs to be able to bind names in the sense that any use of an abstraction such as a variable, a method, or a function is linked to the corresponding declaration. Such *name binding* may defined as part of the type system or they may be defined somewhat separately. We discuss types in detail in Chapter 9. Even when a language does not have an interesting type system, i.e., different types and rules about their use in abstractions, the language may still feature other constraints regarding, for example, the correct use of names. Thus, we may also speak of *well-formedness* more generally, as opposed to *well-typedness* more specifically. For instance, in FSML, the events handled by a given source state must be distinct for the sake of determinism.

When definitions of syntax, types, and semantics are considered formal artifacts such that these artifacts are treated in a formal (mathematical) manner, then we operate within the context of *programming language theory*. A formal approach

is helpful, for example, when approaching the question of *soundness*. That is: *Are the type system and semantics in alignment in that properties described by the type system agree with the actual runtime behavior described by the semantics?*

In the present book, we use semiformal language definitions and we assume them to be useful as concise executable specifications that will help software engineers in implementing software languages, as discussed below. For reasons of limiting the scope and size of the book, we are not much concerned with the formal analysis ("metatheory") of language definitions, in the sense of soundness or otherwise.

1.3.2 Language Implementation

Let us also have a deeper look at the lifecycle phase of language implementation. The discussion gravitates slightly towards programming languages, but most elements apply similarly to MDE and DSLs.

1.3.2.1 Compilation versus Interpretation

One may distinguish two approaches to language implementation:

Interpretation An interpreter executes elements of a software language to produce, for example, the I/O behavior of a function, the result of a database query, or the object graph corresponding to method invocations in an OO program. We will develop some interpreters in Chapter 5. We will relate semantics and interpretation in Chapters 8 and 11.

Compilation A compiler translates (transforms) elements of an executable software language into elements of another executable software language. This translation may or may not lower the level of abstraction. For instance, a compiler may translate a high-level programming language into low-level assembly or virtual machine code. Likewise, a compiler for a DSL may target a GPL, thereby also lowering the level of abstraction. Alternatively, a compiler may be more of a translator between similar languages without much lowering the level of abstraction. In particular, a compiler may piggyback on yet another compiler for its target language. We will develop a simple compiler in Chapter 5.

In a formalist's view, an interpreter would be derived more or less directly from a suitable form of semantics definition for a language. However, this does not cover all forms of interpretation, because a given semantics may not directly enable a practically motivated form of interpretation. For instance, consider the semantics of a formal grammar as a language (a set) generated by that grammar. This semantics is not immediately usable for "interpretation" in the sense of parsing.

Executing a program by interpretation is usually assumed to be slower than executing the target of compilation. Interpretation allows a code base to be extended as the program runs, whereas compilation prevents this.

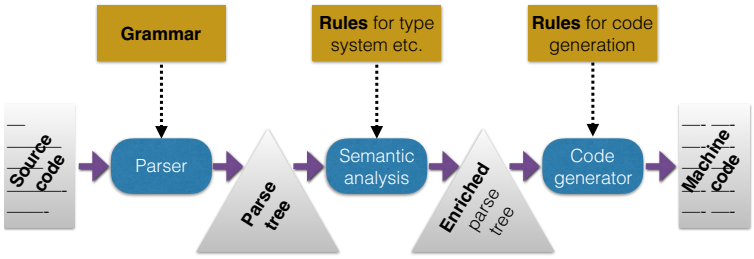

Fig. 1.4 Simplified data flow in a compiler. The rectangles with rounded edges represent logical phases of compilation. The remaining nodes (rectangles and triangles) correspond to input and output, as expressed by the direction of the arrows.

Interpretation versus compilation is not a clear-cut dichotomy; neither do we usually deal with a single layer of program execution. Consider the usual scheme of compiling languages to virtual machines. For instance, Java is compiled to Java bytecode and C# is compiled to.NET's CIL. These target languages are implemented in turn by virtual machines which may be regarded as interpreters of the bytecode languages at hand or, in fact, compilers, as they may translate bytecode into native machine code on a target platform. There is also the notion of just-in-time (JIT) compilation which can be seen as a compilation/interpretation hybrid in that compilation happens as part of interpretation such that compilation can be adjusted to the runtime context. Virtual machines, as mentioned before, usually leverage JIT compilation. In other words, a JIT compiler is therefore an optimization of an interpreter which tries to achieve the performance of a compiler while preserving the dynamicity of an interpreter.

In addition to the basic dichotomy of compilation versus interpretation, there is also the specific classification of implementation strategies for domain-specific languages – internal versus external versus embedded DSL, as discussed in Section 1.2.4.

In MDE, one does not necessarily speak of compilation, but instead of model-to-model transformation, code generation, or model-to-text transformation. However, the underlying principles are very much alike.

1.3.2.2 Architecture of a Compiler

Compilers and interpreters consist of several components. The decomposition of a compiler into standard components with the associated data flow is summarized in Fig. 1.4; see textbooks on compiler construction [2, 186, 14] for an in-depth discussion. Thus, the source code ("text") is mapped to a syntax tree (i.e., a parse tree, a concrete syntax tree (CST), or an abstract syntax tree (AST)), which is then further enriched with attributes and links. Eventually, code for a virtual or actual machine is generated. These conceptual phases may be properly separated ("multi-

pass compilation") or may be integrated into one phase ("single-pass compilation"). The components are explained more in detail as follows:

Parser A parser verifies the conformance of given input (i.e., text) to the syntax rules of a language and represents the input in terms of the structure defined by the rules. A parser performs parsing. Compilers and interpreters begin by parsing. Many other language processors, as discussed below, also involve parsing.

Semantic analysis A syntax tree only represents the structure of the source code. For any sort of nontrivial treatment such as code generation, the syntax tree needs to be enriched with attributes and links related to typing and name binding. Names with their bindings and other attributes may be aggregated in a data structure which is referred to as a symbol table or environment.

Code generator The enriched syntax tree is translated, more or less directly, into machine code, i.e., code of some actual or virtual machine. In particular, code generation involves resource and storage decisions such as register allocation, i.e., assigning program variables to processor registers of the target machine. In this book, few technicalities of code generation are discussed; this topic is covered perfectly by the literature on compiler construction.

Ideally, the components are described by specifications such as grammars, type systems, name-binding rules, and rewrite systems, as indicated in Fig. 1.4. In practice, the components are often implemented in a more ad hoc fashion.

This is a simplified data flow, because actual compilers may involve additional phases. That is, parsing may consist of several phases in itself: preprocessing; lexical analysis (scanning, lexing, or tokenization); syntax analysis including parse-tree construction and syntax desugaring. Also, there may be extra steps preceding code generation: translation to a (simpler) *intermediate representation* (IR) and IR-level optimization. Further, code generation may also involve *optimization* at the level of the target language and a separation between translation to assembly code, mapping to machine code, and some elements of linking. Finally, code generation may actually rely on translation such that the given input language is translated into a well-defined subset of an existing (programming) language so that an available compiler can be used afterwards.

Exercise 1.3 (An exercise on language implementation) [Basic level]
Research the current version of the JDK (Java Development Kit) and identify and characterize at least two language implementations that are part of it.

1.3.2.3 Classification of Language Processors

Languages are implemented in many ways other than just regular compilers and interpreters. We use the term "language processor" to refer to any sort of functionality for automated processing of software artifacts in a language-aware manner,

i.e., with more or less awareness of the syntax, types, and semantics of the artifacts. Examples of language processors include documentation generators, refactoring tools, bug checkers, and metrics calculation tools. Language processors often consist of several components and perform processing in phases, as we discussed above for compilers. Rather than classifying language processors directly, let us classify language-based software components. We do not make any claim of completeness for this classification. Several of the classifiers below will reappear in the discussion of the role of software languages across different software engineering areas (Section 1.4):

Parser or text-to-model transformation The term "parser" has already been introduced in the context of compilation and interpretation. The term "text-to-model transformation" is specifically used in the MDE community when one wants to emphasize that the result of parsing is not a parse *tree*, but rather a model in the sense of metamodeling, thus potentially involving, for example, references after completing name binding.

Unparser, formatter, pretty printer, or model-to-text transformation An artifact is formatted as text, possibly also subject to formatting conventions for the use of spaces and line breaks. Formatting may start from source code (i.e., text), concrete syntax trees (i.e., parse trees), or abstract syntax trees. Formatting is typically provided as a service in an IDE.

Preprocessor As part of parsing, code may be subject to macro expansion and conditional compilation. Such preprocessing may serve the purpose of, for example, configuration management in the sense of software variability and desugaring in the sense of language extension by macros. Interestingly, preprocessing gives rise to a language of its own for the preprocessing syntax such that the preprocessor can be seen as an interpreter of that language; the result type of this sort of interpretation is, of course, text [99]. One may also assume that a base language is extended by preprocessing constructs so that preprocessing can be modeled as a translation from the extended to the base language. In fact, some macro system work in that manner. In practice, preprocessing is often used in an undisciplined (i.e., not completely syntax-aware) manner [29, 18, 184].

Software transformation or model-to-model transformation A software transformation is a mapping between software languages. The term "model-to-model transformation" is used in the model transformation and MDE community. We may classify transformations in terms of whether the source and target languages are the same and whether the source and target reside at the same level of abstraction [195]. Thus:

- **Exogenous transformation** The source and target languages are different, as in the case of code generation (translation) or language migration.
- **Endogenous transformation** The source and target languages are the same, as in the case of program refactoring or compiler optimization [116, 196]. We can further distinguish in-place and out-place transformations [195, 35] in terms of whether the source model is "reused" to produce the target model. (Exogenous transformations are necessarily out-place transformations.)

Horizontal transformation The source and target languages reside at the same level of abstraction, as in the case of refactoring or language migration.

Vertical transformation The source and target languages reside at different levels of abstraction. In fact, both directions, i.e., lowering and raising the level of abstraction, make sense. An example of lowering is code generation or formal refinement (such as refining a specification into an implementation). An example of raising is architectural recovery [158].

Software analysis or software analyzer A software analysis verifies or computes some property of a given language element. Here are a few well-known objectives of software analysis:

Termination analysis The termination of a program is verified or potential or actual termination problems are detected; see, e.g., [235].

Performance analysis The performance of a program (or a model or a system) is predicted or performance problems are detected (see, e.g., [138]).

Alias analysis It is determined whether or not a given storage cell can be addressed in multiple ways (see, e.g., [243]).

Bug analysis Bad smells and potential or actual bugs in a program (or a model or a system) are detected (see, e.g., [19]).

Usage analysis Data about the usage of a language is collected, for example, the frequency or presence of constructs or idioms in a corpus (see, e.g., [172, 167, 120]).

As mentioned before, compilers and interpreters perform a semantic analysis that verifies conformance to rules for typing, naming, scoping, etc. Compilers also perform data-flow analysis and control-flow analysis to facilitate optimizations. Overall, software transformations often rely on software analyses to accomplish their work. One may argue that software analysis is a form of software transformation; we avoid this discussion here.

There are many analyses in software engineering that leverage methods from different areas of computer science, for example, search-based algorithms, text analysis, natural language processing, model checking, and SAT solving. In this book, we focus on the software language- and software engineering-specific aspects of software analysis. We will discuss simple instances of software analysis in Chapter 5.

Software translator The notion of translation generalizes the more basic notion of compilation. A translator implements a mapping between different software languages. A migration tool, for example, to accommodate breaking changes due to language evolution, is an example of a translator that is not also a compiler. Typically, we assume that translation is semantics-preserving. A translation is an exogenous transformation. We will develop a simple compiler (translator) in Chapter 5.

Software generator Generation, as in the case of program generation or generative programming [74], is very similar to translation. The key focus is here on how the generator lowers the level of abstraction and optimizes a program by

eliminating inefficiency due to the use of abstraction mechanisms or domain-specific concepts, subject to specialized analyses and optimizations. Software generation is used, for example, to derive language-processing components (e.g., parsers, rewrite engines, pretty printers, and visitor frameworks) from grammars or rule-based specifications. The implementation of software generators may rely on dedicated language concepts, for example, multi-staged programming [248, 217] (Chapter 12) and templates [236, 76, 256, 211].

Test-data generator Given a grammar or a metamodel, valid language elements are generated in a systematic manner. Such generation may also be controlled by additional parameters and specifications and may be tailored towards a particular use case, for example, for testing a compiler frontend, a virtual machine implementation, or a serialization framework [174].

Program specializer As a special case of program optimization, program specializers or partial evaluators aim at simplifying a given program (or software system) on the basis of statically available partial input [139, 129, 68]. We will discuss partial evaluation in Chapter 12.

Additional details regarding the classification of transformations (software transformations, software analyses, model transformations, source-to-source transformations) can be found elsewhere [59, 195, 75, 249, 121, 8].

Exercise 1.4 (Classification of conference papers) [Intermediate level]
Study the most recent edition of the International Conference on Model Transformation (ICMT) and extract the forms of transformations that are discussed in the papers. Classify these forms according to the classifiers given above. (You may want to follow some of the guidelines for a systematic mapping study [206, 207].)

With reference to "language usage" (as in Fig. 1.3), we should mention another category of language implementation: IDEs integrate a basic language implementation (e.g., a compiler) with other language services for editing, code completion, refactoring, formatting, exploration, etc. Thus, an IDE is an integrated system of components supporting language users.

1.3.2.4 Metaprogramming Systems

Language implementations, including all kinds of language processors, are implemented by means of metaprogramming. A metaprogram is a program that consumes or produces (object) programs. To this end, "normal" programming environments may be used. However, there also exist dedicated *metaprogramming systems* that incorporate expressiveness or tool support for transformation, analysis, and possibly concrete object syntax, for example, Rascal [151, 150], TXL [69, 70], Stratego XT [262, 47], Converge [255], and Helvetia [215, 214].

In the neighborhood of metaprogramming systems, there are also *language definition* or *executable semantic frameworks* (e.g., the K semantic framework [221]

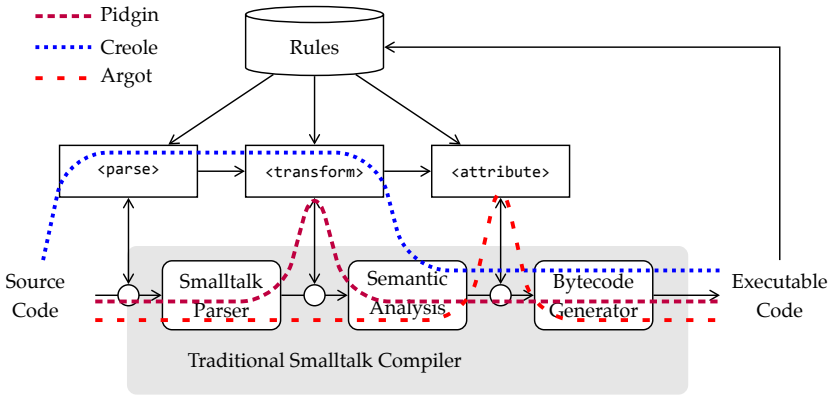

Fig. 1.5 The code compilation pipeline of Helvetia, showing multiple interception paths; there are hooks to intercept parsing <parse>, AST transformation <transform>, and semantic analysis <attribute>. Source: [215]. Additional capabilities of Helvetia support editing (coloring), debugging, etc. © 2010 Springer.

and PLT Redex [107]), *compiler frameworks* (e.g., LLVM [180]), and *modeling frameworks* (e.g., AM3 [24]).

Metaprogramming and software language engineering efforts may be "advertised" through *software language repositories* (SLRs) [165], i.e., repositories with components for language processing (interpreters, translators, analyzers, transformers, pretty printers, etc.). Further examples of SLRs include the repositories for Krishnamurthi's textbook on programming languages [160], Batory's Prolog-based work on teaching MDE [28], Zaytsev et al.'s software language processing suite (SLPS) [278], and Basciani et al.'s extensible web-based modeling platform MDEForge [26].

1.3.2.5 Language Workbenches

Metaprogrammers may also be supported in an interactive and integrated fashion. Accordingly, the notion of *language workbenches* [96, 97, 144, 143, 267, 266, 269, 263] encompasses enhanced metaprogramming systems that are, in fact, IDEs for language implementation. A language workbench assumes specialized language definitions that cater for IDE services such as syntax-directed, structural, or projectional editing, coloring, synthesis of warnings and errors, package exploration, quick fixes, and refactorings.

Figure 1.5 illustrates the compilation pipeline of the metaprogramming system Helvetia [214, 215]. In fact, Helvetia is an extensible development environment for embedding DSLs into a host language (Smalltalk) and its tools such as the editor and debugger. Thus, Helvetia is a language workbench.

1.3.3 Language Evolution

Let us briefly discuss more concrete scenarios of language evolution; see also [48] for more background:

Language extension A language construct or concept is added to the language design. The language definition (syntax, semantics, pragmatics) and implementation, as well as the documentation are to be extended accordingly.

Language restriction Some language construct or concept has been found to be problematic in terms of, for example, its semantics or implementation. The language definition (syntax, semantics, pragmatics) and implementation, as well as the documentation need to be restricted accordingly. A migration path may need to be offered to users.

Language revision A language extension can be formally seen as moving to a superset, when looking at languages in a set-theoretic manner. A language restriction is then seen as moving to a subset. We speak of language revision when neither of these clear-cut cases applies. A migration path may need to be offered to users.

Language integration The scenarios named above are concerned with a single language on the time line. We use the term "language integration" here for situations when more than one language needs to be somehow combined. For instance, one language may need be embedded into another language (e.g., SQL is embedded into Java) or multiple variants of a given language need to be unified or handled in an interoperable manner.

In the context of language revision and integration, we face challenges in terms of coupling between language definitions and existing artifacts. For instance, the evolution of a concrete syntax needs to be complemented by the evolution of the corresponding abstract syntax and vice versa; also, the existing language elements may need to be co-evolved [260, 176].

Exercise 1.5 (Extension of the Java language) [Intermediate level]
Research the available documents on the evolution of the Java language and identify a language extension added in a specific language version. Demonstrate the extension with a sample and argue what syntactic categories of the Java language are affected.

Exercise 1.6 (Restriction of the Haskell language) [Intermediate level]
Research the available documents on the evolution of the Haskell language and identify a language restriction in a specific language version. Demonstrate the restriction with a sample and summarize the reasoning behind the restriction. Why was the restriction considered reasonable? What is the migration path, if any?

1.4 Software Languages in Software Engineering

Various software engineering areas, and, in fact, more broadly, many areas in computer science, involve software languages in an essential manner, i.e., these areas necessitate parsing, analysis, transformation, generation, and other forms of processing software language-based artifacts. Several software engineering areas are discussed in the sequel. For each area, we identify a few typical application domains for software languages. This will give us a good sense of the omnipresence of software languages in software engineering, software development, and IT.

1.4.1 Software Re-Engineering

We quote: "re-engineering ... is the examination and alteration of a subject system to reconstitute in a new form" [59]. These are some application domains for software languages in the re-engineering area:

Refactoring The improvement (in fact, the possibly automated transformation) of the design of code or models without changing its "behavior" [116, 196]. In particular, the "functional" behavior has to to preserved; some nonfunctional aspects such as execution time may be modified. Refactorings are often meant to be performed interactively, for example, by means of an IDE integration. Refactorings may also be recorded and "replayed" to propagate changes, for example, from a refactored library to client code [130, 227]. Ultimately, refactorings may be as complex as system restructuring to serve a different architecture [6].

Migration A migration can be viewed as a more or less automated transformation of a program or a software system to conform to a different API, language, or architectural requirement. For instance, a language migration is concerned with rewriting a system written in one high-level language to use another high-level language instead [250, 44, 241]. Language migrations may be challenging because of an array of aspects, for example, different platforms for source and target, or different type systems or primitive types.

Wrapping A wrapper is a form of adaptor that provides a different interface for existing functionality. The interfaces involved may be essentially object-oriented APIs [227, 25]. We may also wrap a legacy system in terms of its user interface or procedural abstractions as a service [242, 63, 240, 55]. In many cases, wrappers may be semiautomatically generated or vital information may be gathered by an automated analysis.

Figure 1.6 illustrates a very simple refactoring scenario in an OO programming context: the boxed statements at the top are to be extracted into a new method, as shown in completed form at the bottom of the figure. In terms of the structural rules for transforming source code, refactorings may be relatively simple, but they often involve nontrivial preconditions and constraints to be met for correctness' sake [253,

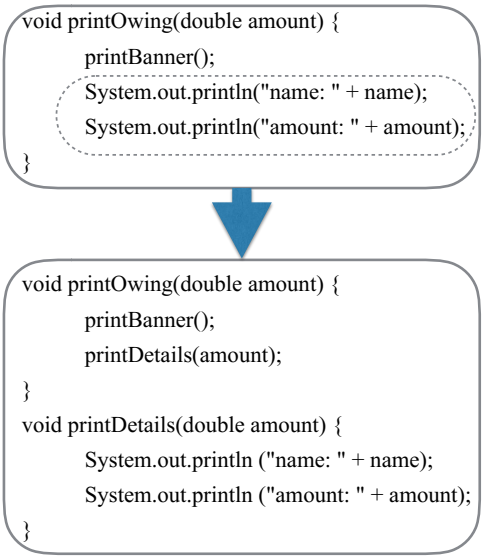

Fig. 1.6 Illustration of the "extract method" refactoring.

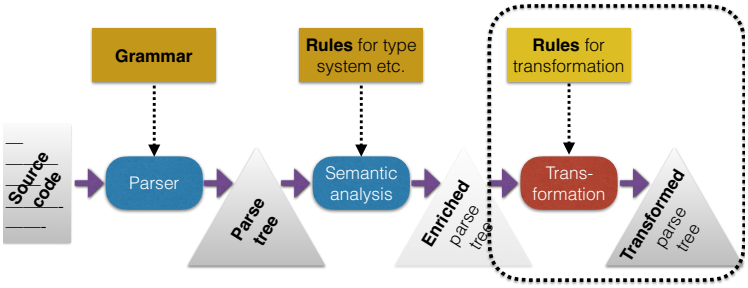

Fig. 1.7 Overall data flow for a re-engineering transformation. We have marked the phase which replaces code generation in the standard data flow for compilation.

228]. Even in the simple example at hand, some constraints have to be met; for example, the extracted statements must not return.

Figure 1.7 shows the overall data flow for a re-engineering transformation as needed, for example, for refactoring or restructuring. This data flow should be compared with the data flow for compilation; see Fig. 1.4. The two data flows share the phases of parsing and semantic analysis. The actual transformation is described (ideally) by declarative rules of a transformation language. Not every re-engineering use case requires a full-blown semantic analysis, which is why we have grayed out slightly the corresponding phase in Fig. 1.7. In fact, not even a proper syntax-aware transformation is needed in all cases, but instead a lexical approach may be applicable [152].

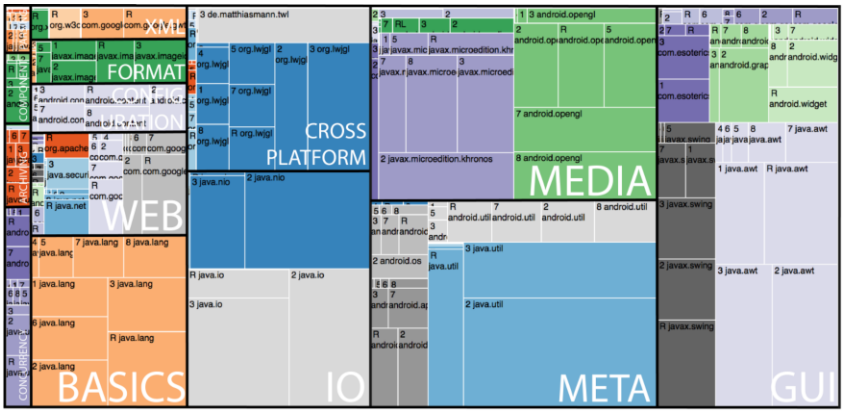

Fig. 1.8 An API-usage map for an open-source Java project. The complete rectangle (in terms of its size) models the references to all APIs made by all developers. The nested rectangles partition references by domain (e.g., GUI rather than Swing or AWT). The rectangles nested further partition references by API; one color is used per API. Within each such rectangle, the contributions of distinct developers (1, . . . , 8 for the top-eight committers and "R" for the rest) are shown. Source: [4].

1.4.2 Software Reverse Engineering

We quote: "reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [59]. For instance, we may extract a call graph from a system, thereby identifying call sites (such as packages, files, classes, methods, or functions) and actual calls (such as method or function calls). Reverse engineering may also be concerned with *architecture recovery* [126, 128, 158, 33], for example, the identification of components in a legacy system. Overall, reverse engineering is usually meant to help with program comprehension and to prepare for software re-engineering or to otherwise facilitate software development.

Figure 1.8 shows the visual result of a concrete reverse engineering effort aimed at understanding API usage in Java projects [4]. The tree map groups API references (i.e., source code-level references to API methods) so that we can assess the contributions of different APIs and of individual developers for each API to the project.

Figure 1.9 shows the overall data flow for a reverse engineering component that is based on the paradigm of fact extraction [109, 201, 185, 27]. Just as in the cases of compilation or transformation for re-engineering, we begin with parsing and (possibly customized) semantic analysis. The data flow differs in terms of last phase for fact extraction. The extracted facts can be thought of as sets of tuples, for example, pairs of caller/callee sites to be visualized eventually as a call graph.

Reverse engineering often starts from some sort of fact extraction. Reverse engineering may also involve data analysis based, for example, on relational alge-

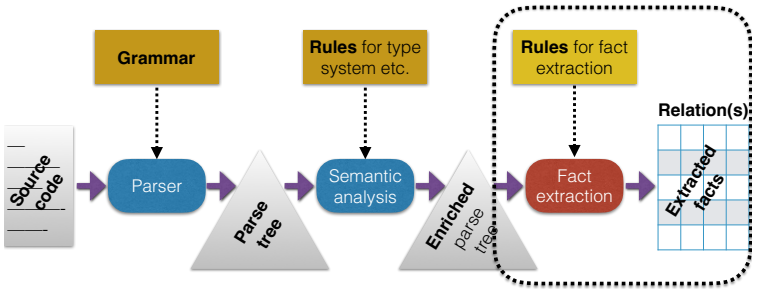

Fig. 1.9 Overall data flow for fact extraction in reverse engineering. We have marked the phase which replaces code generation in the standard data flow for compilation. The resulting tables represent "projections" of the source code, for example, call relationships between functions.

bra [134, 38, 37, 135]. Reverse engineering is by no means limited to source code artifacts, but may also involve, for example, documentation, models, and commits. The results are often communicated to software engineers or other stakeholders by means of software visualization [148, 187, 177, 1, 178]. In the earlier example concerned with API-usage analysis, fact extraction from Java source code was used to extract API references, commits were analyzed to associate API references with developers, and visualization as a tree map was used to communicate the result.

1.4.3 Software Analysis

There exist diverse forms of software analysis that support software reverse engineering, software maintenance, software evolution, and program comprehension. Here are some forms:

Program slicing This is the computation of a simplified program, for example, the statements which affects the state or the result at some point of interest [271, 136, 252, 43, 10].

Feature location This is the semiautomated process of locating features or specific program functionality in software systems or product lines, based on different forms of source-code analysis, helping ultimately with code refactoring, software maintenance, clone detection, and product-line engineering [224, 86, 270, 111, 34, 191, 12, 166].

Clone detection This is the largely automated process of determined duplicated source code, using various means of attesting the presence of equal or similar code at a textual, lexical, syntactic, or semantic level [22, 30, 222, 223].

Traceability recovery This is the largely automated process of recovering trace links between different kinds of artifacts, for example, documentation and source code, to attest that the linked artifacts or fragments thereof are related [64, 145, 118, 188, 233].

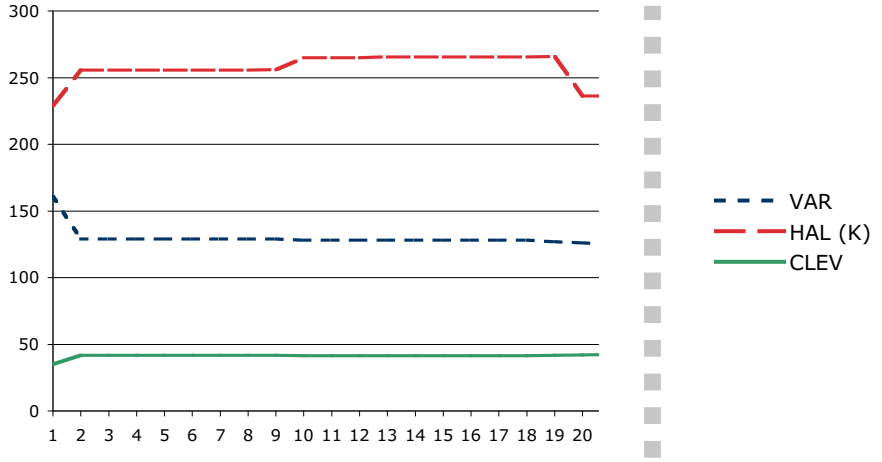

Fig. 1.10 Three different grammar metrics over time for the versions of an industrial-strength parser for a specification language. Source: [7]. © 2008 Springer.

Design-pattern detection This is the automated process of detecting instances of design patterns in actual code, subject to checking structural and behavioral constraints [218, 114, 112, 154, 153, 276]. This area also relates to the detection and analysis of micro-patterns, for example, in bytecode [120], decompilation [110], and disassembly, for example, for the purpose of detecting security issues [9].

Change-impact analysis This is the semiautomated process of "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [15]; see also [246, 183, 119, 87, 212, 213].

Code smell This is an automatically checkable symptom in source code that possibly indicates a deeper problem such as abnormal complexity or insufficient modularity [92, 257, 275, 205, 113, 115]. The corresponding checking or detection activity is accordingly referred to as code smell detection.

Software metric This is a computable standard measure of the degree to which given source code or a complete software system possesses some property, for example, complexity in terms of lines-of-code (LOC) or McCabe (i.e., Cyclomatic complexity) [108, 179, 209, 79, 50, 198, 259, 57].

Coding convention This is a mechanized (executable and checkable) convention for authoring code in a language, possibly also specific to a project; such conventions are checked by corresponding tools that are typically integrated into an IDE so that developers receive continuous feedback on the code's compliance with the conventions [123].

Software analysis is often combined with software visualization; for example, the values of computed software metrics are typically visualized [177, 23] to better access the distribution of metrics over a system or changes in the metrics over time.

Figure 1.10 gives an example of how metrics and simple visualization can be combined to analyze a software process – in this case, a process for the improvement of a grammar [7]. The changes of the values of the metrics can be explained as consequences of the specific grammar revisions applied at the corresponding commit points.

1.4.4 Technological Spaces

We quote: "A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference regular meetings" [161].

For instance, there are the following technological spaces, which we characterize in a keyword style by pointing out associated languages, technologies, and concepts:

Grammarware string, grammar, parsing, CST, AST, term, rewriting, ...
XMLware XML, XML infoset, DOM, DTD, XML Schema, XPath, XQuery, XSLT, ...
JSONware JSON, JSON Schema, ...
Modelware UML, MOF, EMF, class diagram, modeling, metamodeling, model transformation, MDE, ...
SQLware table, SQL, relational model, relational algebra, ...
RDFware resource, triple, Linked Data, WWW, RDF, RDFS, OWL, SPARQL, ...
Objectware objects, object graphs, object models, state, behavior, ...
Javaware Java, Java bytecode, JVM, Eclipse, JUnit, ...

We refer to [40] for a rather detailed discussion of one technological space – modelware (MDE). We refer to [89] for a discussion of multiple technological spaces with focus on Modelware and RDFware centric and cursory coverage of grammarware, Javaware, and XMLware and the interconnections between these spaces.

Technological spaces are deeply concerned with software languages:

Data models The data in a space conforms to some data model, which can be viewed as a "semantic domain" in the sense of semantics in the context of language definition. For instance, the data model of XML is defined by a certain set of trees, according to the XML infoset [274]; the data model JSON is a dictionary format that is a simple subset of Javascript objects; and the data model of SQLware is the relational model [67].

Schema languages Domain- or application-specific data can be defined by appropriate schema-like languages. Schemas are to tree- or graph-based data what (context-free) grammars are to string languages [149]. For instance, the schema language of JSON is JSON Schema [208]; the schema language of grammarware is EBNF [137] in many notational variations [277]; and the schema languages of

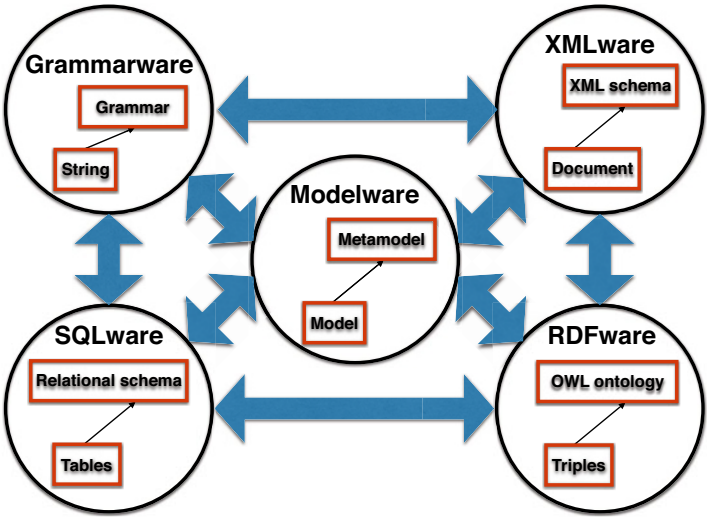

Fig. 1.11 A few technological spaces with their instance (data) level and the schema level. The thin arrows model "conformance" such as an XML document conforming to an XML schema. The thick arrows hint at expected "bridges" between the spaces as needed for technological space travel. Inspired by [161].

RDF are RDFS and OWL. Schemas and their instances give rise to a conformance relationships; see Fig. 1.11 for a few examples.

Query languages A technological space tends to offer one or more languages for querying data in that space. For instance, the query languages of XMLware are XPath and XQuery; the query language of RDFware is SPARQL; and the query language of SQLware is SQL (specifically, the SELECT part of it).

Transformation languages A technological space tends to offer one or more languages for transforming data in that space. For instance, the transformation language of XMLware is XSLT; the transformation language of RDFware is SPARQL with updates; and the transformation language of SQLware is SQL (specifically, the DELETE, UPDATE, and CREATE parts of it).

Programming language integration Technological spaces may be integrated into programming languages by either appropriate APIs for the underlying query and transformation languages, or some form of mapping (see below), or proper language integration. The JDBC approach in the Java platform is a basic example of the API option for integrating SQLware (SQL) into Java.

Mapping There is a recurrent need to map across technological spaces such as objectware, XMLware, and SQLware [169], leading to what we refer to as technological space travel; this is also referred to as operational bridges in [161]; see Fig. 1.11 for a few examples. For instance, object/relational mapping allows one to make objects persistent in a database and to access a database in an object-

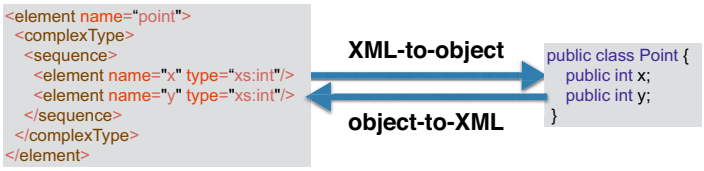

Fig. 1.12 Mapping object models to XML schemas and vice versa

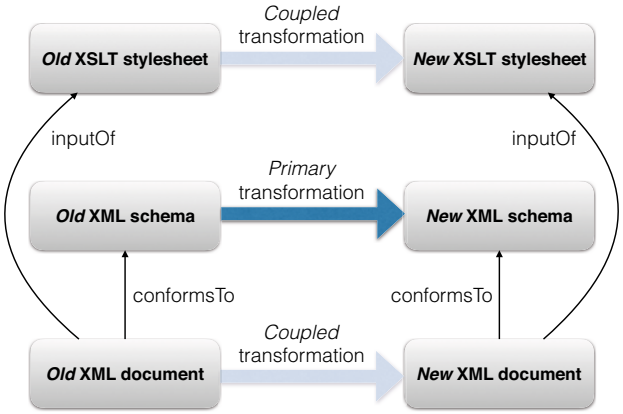

Fig. 1.13 Mapping in an XML context at the instance (XML), the type (XSD), and the stylesheet (XSLT) level.

oriented program. Mapping, as such, is a translation problem – thus the link to software languages.

Coupling Mapping, as just explained, may need to be accomplished at multiple levels: the data level, the schema level, and the processing level (i.e., queries and transformations). Depending on the space, each level may be challenging in itself. For instance, mapping schema languages is complicated in the presence of space-specific constraint forms. However, mapping is specifically challenging because all the levels need to be mapped in a coupled ("synchronized") manner [162, 149, 73, 164].

Figure 1.12 illustrates the XMLware instance of mapping at the schema level; we assume that the XML type on the left may be mapped to the Java/C# class on the right and vice versa. Of course, such a mapping is not straightforward, because of the many differences in the underlying data models and schema languages, giving rise to an impedance mismatch [170].

Figure 1.13 illustrates three levels of mapping for the XMLware space. Consider the following scenario. There is an XML document meant to conform to an XML schema; there is also a document processor (an XSLT stylesheet) that is used to process the XML document. Now assume that the schema is changed to refactor

the format. Existing documents must be converted ("mapped") to conform to the new schema. Also, the existing document processor must be converted to be able to process converted documents.

We refer below to examples of work on mappings:

- XML/object mapping [170, 163, 251, 5];
- XML/grammar mapping [202];
- object/relational mapping [52, 51, 53, 200, 36, 254, 238];
- XML/relational mapping [58];
- relational/RDF+OWL mapping [234];
- XML/relational with support for queries and constraints [32, 264].

We also refer to examples of work on coupling:

- schema/instance mapping for databases [124, 261];
- metamodel/model mapping for modelware [268, 60, 62];
- program/schema mapping for databases and applications [65, 66, 125];
- document/schema mapping for XML [168, 72];
- string/grammar mapping [260, 176];
- co-evolution of GMF editor models [225];
- refactoring of object/relational mapping [230].

1.4.5 Model-Driven Engineering

MDE [39, 40, 231] (see also some related model-based approaches [247, 181]) combines the broad notions of modeling, metamodeling [203], modeling languages, and model transformation. This area constitutes an important space for language definition, implementation, usage, processing, and evolution. We identify the key notions of the MDE (modelware) space as follows:

Modeling languages Models (artifacts, programs, etc.) are elements of suitable modeling languages, of which there are many. For instance, the mainstream UML approach offers a family of modeling notations for behavioral and structural modeling.

Domain-specific modeling languages (DSMLs) On the basis of metamodeling and other modeling concepts, DSMLs are modeled and enable the application of modeling to specific domains. The semantics of these DSMLs are also developed in the modelware space, for example, by means of model transformation [16].

Model transformation Foremost, there are model-to-model transformations. Additionally, there are special kinds of transformations to import and export text-based artifacts, i.e., text-to-model and model-to-text transformations. There are dedicated model-transformation languages [194, 75, 140, 8, 142].

Model evolution Models are subject to evolution. Specifically, metamodels and model transformations are also subject to evolution [204]. There are a number of

concepts (see below) which help in understanding or automating evolution, for example, model comparison and model co-evolution.

Model management With the increase in the numbers of models, metamodels, model transformations, versions of models, and relationships, some sort of management needs to be applied to systematically maintain repositories of models. Such model management gives rise to yet another form of model – megamodels [41, 42, 103, 84, 175, 165, 127].

Model comparison In the context of model evolution, models also need to be compared [244, 122, 146] to extract differences also on the basis of suitable diff models [61, 60].

Model merging We quote: "merging is the action of combining two models, such that their common elements are included only once and the other ones are preserved" [189]; see also [156, 93, 45, 189, 220, 77]. Model merging can be viewed as a form of model synchronization; see below.

Model weaving We quote: "weaving involves two actors: an aspect and a base model. The aspect is made of two parts, a pointcut, which is the pattern to match in the base model, and an advice, which represents the modification made to the base model during the weaving" [189]; see also [226, 193, 159]. Model weaving helps with, for example, crosscutting concerns to be addressed by models.

Model synchronization Software systems may involve multiple models to be related by some consistency relation. Model synchronization is the process of establishing consistency in response to changes in individual models, for example, by means of propagating changes in the form of a difference (a "delta") from one model to another; see [11, 82, 85, 83].

Models@run.time This notion applies when any sort of model is used during the operation of a system. We quote: "the key property of models@run.time systems is their use and provision of manageable reflection, which is characterized to be tractable and predictable and by this overcomes the limitation of reflective systems working on code" ([17] which was part of the seminar [31]). In different terms, the program and model become essentially united or integrated [71, 80]. Arguably, reflective language infrastructures (e.g., Smalltalk-based ones) may also provide manageable reflection and other aspects of models@run.time [91, 54, 245].

Model co-evolution When multiple modeling languages are used simultaneously, the models may involve common entities and, thus, coupled with respect to evolution; see, for example, the coupled evolution of the models of GMF-based editors [225]. Language evolution, as mentioned earlier, is a special case of model co-evolution. That is, language evolution is model/metamodel co-evolution. Language evolution is a particularly important problem in the context of domain-specific (modeling) languages because these languages may evolve rapidly and significantly without being limited by backward compatibility [268, 131, 132, 216].

Summary and Outline

We have argued that software languages permeate software engineering, software development, and IT. We have classified software languages in different ways. We have presented a lifecycle for software languages; the lifecycle covers, for example, language implementation in a compiler or language processor for re- and reverse engineering. Software languages deserve to be treated in an engineering manner, thereby giving rise to the discipline of software language engineering (SLE). All of the example languages and most of the SLE scenarios introduced in this chapter will play a role in the rest of the book.

References

1. Abdeen, H., Ducasse, S., Pollet, D., Alloui, I., Falleri, J.: The Package Blueprint: Visually analyzing and quantifying packages dependencies. Sci. Comput. Program. **89**, 298–319 (2014)
2. Aho, A., Monica S., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison Wesley (2006). 2nd edition
3. Akinin, A., Zubkov, A., Shilov, N.: New developments of the computer language classification knowledge portal. In: Proc. Spring/Summer Young Researchers' Colloquium on Software Engineering (2012)
4. Aksu, H., Lämmel, R., Kwasnik, W.: Visualization of API experience. Softwaretechnik-Trends **36**(2) (2016)
5. Alagic, S., Bernstein, P.A., Jairath, R.: Object-oriented constraints for XML schema. In: Proc. ICOODB, *LNCS*, vol. 6348, pp. 100–117. Springer (2010)
6. Almonaies, A.A., Alalfi, M.H., Cordy, J.R., Dean, T.R.: A framework for migrating web applications to web services. In: Proc. ICWE, *LNCS*, vol. 7977, pp. 384–399. Springer (2013)
7. Alves, T.L., Visser, J.: A case study in grammar engineering. In: Proc. SLE 2008, *LNCS*, vol. 5452, pp. 285–304. Springer (2009)
8. Amrani, M., Combemale, B., Lucio, L., Selim, G.M.K., Dingel, J., Traon, Y.L., Vangheluwe, H., Cordy, J.R.: Formal verification techniques for model transformations: A tridimensional classification. J. Object Technol. **14**(3), 1–43 (2015)
9. Andriesse, D.: Analyzing and Securing Binaries Through Static Disassembly. Ph.D. thesis, Vrije Universiteit Amsterdam (2017)
10. Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., Tratt, L.: State-based model slicing: A survey. ACM Comput. Surv. **45**(4), 53 (2013)
11. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: GTTSE 2007, Revised Papers, *LNCS*, vol. 5235, pp. 3–46. Springer (2008)
12. Antkiewicz, M., Ji, W., Berger, T., Czarnecki, K., Schmorleiz, T., Lämmel, R., Stanciulescu, S., Wasowski, A., Schaefer, I.: Flexible product line engineering with a virtual platform. In: Proc. ICSE, pp. 532–535. ACM (2014)
13. Anureev, I.S., Bodin, E., Gorodnyaya, L., Marchuk, A.G., Murzin, A.G., Shilov, N.V.: On the problem of computer language classification. Joint NCC&IIS Bulletin, Series Computer Science **27**, 1–20 (2008)
14. Appel, A., Palsberg, J.: Modern Compiler Implementation in Java. Cambridge University Press (2002). 2nd edition
15. Arnold, R., Bohner, S.: Software Change Impact Analysis. Wiley-IEEE Computer Society (1996)

16. Aßmann, U., Bartho, A., Bürger, C., Cech, S., Demuth, B., Heidenreich, F., Johannes, J., Karol, S., Polowinski, J., Reimann, J., Schroeter, J., Seifert, M., Thiele, M., Wende, C., Wilke, C.: DropsBox: the Dresden Open Software Toolbox – Domain-specific modelling tools beyond metamodels and transformations. SoSyM **13**(1), 133–169 (2014)
17. Aßmann, U., Götz, S., Jézéquel, J., Morin, B., Trapp, M.: A reference architecture and roadmap for models@run.time systems. In: Bencomo et al. [31], pp. 1–18
18. Aversano, L., Penta, M.D., Baxter, I.D.: Handling preprocessor-conditioned declarations. In: Proc. SCAM, pp. 83–92. IEEE (2002)
19. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Using static analysis to find bugs. IEEE Software **25**(5), 22–29 (2008)
20. Babenko, L.P., Rogach, V.D., Yushchenko, E.L.: Comparison and classification of programming languages. Cybern. Syst. Anal. **11**, 271–278 (1975)
21. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the Algorithm Language ALGOL 60. Commun. ACM **6**(1), 1–17 (1963)
22. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proc. WCRE, pp. 86–95. IEEE (1995)
23. Balogh, G.: Validation of the city metaphor in software visualization. In: Proc. ICCSA, *LNCS*, vol. 9159, pp. 73–85. Springer (2015)
24. Barbero, M., Jouault, F., Bézivin, J.: Model driven management of complex systems: Implementing the macroscope's vision. In: Proc. ECBS 2008, pp. 277–286. IEEE (2008)
25. Bartolomei, T.T., Czarnecki, K., Lämmel, R.: Swing to SWT and back: Patterns for API migration by wrapping. In: Proc. ICSM, pp. 1–10. IEEE (2010)
26. Basciani, F., Rocco, J.D., Ruscio, D.D., Salle, A.D., Iovino, L., Pierantonio, A.: MDEForge: An extensible web-based modeling platform. In: Proc. CloudMDE@MoDELS, *CEUR Workshop Proceedings*, vol. 1242, pp. 66–75. CEUR-WS.org (2014)
27. Basten, H.J.S., Klint, P.: DeFacto: Language-parametric fact extraction from source code. In: Proc. SLE 2008, *LNCS*, vol. 5452, pp. 265–284. Springer (2009)
28. Batory, D.S., Latimer, E., Azanza, M.: Teaching model driven engineering from a relational database perspective. In: Proc. MODELS, *LNCS*, vol. 8107, pp. 121–137. Springer (2013)
29. Baxter, I.D., Mehlich, M.: Preprocessor conditional removal by simple partial evaluation. In: Proc. WCRE, pp. 281–290. IEEE (2001)
30. Baxter, I.D., Yahin, A., de Moura, L.M., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proc. ICSM, pp. 368–377. IEEE (1998)
31. Bencomo, N., France, R.B., Cheng, B.H.C., Aßmann, U. (eds.): Models@run.time – Foundations, Applications, and Roadmaps, Dagstuhl Seminar 11481, November 27 – December 2, 2011, *LNCS*, vol. 8378. Springer (2014)
32. Berdaguer, P., Cunha, A., Pacheco, H., Visser, J.: Coupled schema transformation and data conversion for XML and SQL. In: Proc. PADL, *LNCS*, vol. 4354, pp. 290–304. Springer (2007)
33. Berger, B.J., Sohr, K., Koschke, R.: Extracting and analyzing the implemented security architecture of business applications. In: Proc. CSMR, pp. 285–294. IEEE (2013)
34. Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K.: What is a feature?: A qualitative study of features in industrial software product lines. In: Proc. SPLC, pp. 16–25. ACM (2015)
35. Bergmayr, A., Troya, J., Wimmer, M.: From out-place transformation evolution to in-place model patching. In: Proc. ASE, pp. 647–652. ACM (2014)
36. Bernstein, P.A., Jacob, M., Pérez, J., Rull, G., Terwilliger, J.F.: Incremental mapping compilation in an object-to-relational mapping system. In: Proc. SIGMOD, pp. 1269–1280. ACM (2013)
37. Beyer, D.: Relational programming with CrocoPat. In: Proc. ICSE, pp. 807–810. ACM (2006)
38. Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. IEEE Trans. Softw. Eng. **31**(2), 137–149 (2005)

39. Bézivin, J.: On the unification power of models. SoSyM **4**(2), 171–188 (2005)
40. Bézivin, J.: Model driven engineering: An emerging technical space. In: GTTSE 2005, Revised Papers, *LNCS*, vol. 4143, pp. 36–64. Springer (2006)
41. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: European MDA Workshops MDAFA 2003 and MDAFA 2004, Revised Selected Papers, *LNCS*, vol. 3599, pp. 33–46. Springer (2005)
42. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proc. OOPSLA/GPCE: Best Practices for Model-Driven Software Development Workshop (2004)
43. Binkley, D., Harman, M., Krinke, J. (eds.): Beyond Program Slicing, *Dagstuhl Seminar Proceedings*, vol. 05451. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
44. Blasband, D.: Compilation of legacy languages in the 21st century. In: GTTSE 2011, Revised Papers, *LNCS*, vol. 7680, pp. 1–54. Springer (2013)
45. Boronat, A., Carsí, J.A., Ramos, I., Letelier, P.: Formal model merging applied to class diagram integration. ENTCS **166**, 5–26 (2007)
46. Bottoni, P., Grau, A.: A suite of metamodels as a basis for a classification of visual languages. In: Proc. VL/HCC, pp. 83–90. IEEE (2004)
47. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(1-2), 52–70 (2008)
48. Bryant, B.R., Jézéquel, J., Lämmel, R., Mernik, M., Schindler, M., Steinmann, F., Tolvanen, J., Vallecillo, A., Völter, M.: Globalized domain specific language engineering. In: Globalizing Domain-Specific Languages – International Dagstuhl Seminar, Dagstuhl Castle, Germany, October 5–10, 2014 Revised Papers, *LNCS*, vol. 9400, pp. 43–69. Springer (2015)
49. Burnett, M.M., Baker, M.J.: A classification system for visual programming languages. J. Vis. Lang. Comput. **5**(3), 287–300 (1994)
50. Byelas, H., Telea, A.: The metric lens: Visualizing metrics and structure on software diagrams. In: Proc. WCRE, pp. 339–340. IEEE (2008)
51. Cabibbo, L.: A mapping system for relational schemas with constraints. In: Proc. SEBD, pp. 237–244. Edizioni Seneca (2009)
52. Cabibbo, L.: On keys, foreign keys and nullable attributes in relational mapping systems. In: Proc. EDBT, *ACM International Conference Proceeding Series*, vol. 360, pp. 263–274. ACM (2009)
53. Cabibbo, L., Carosi, A.: Managing inheritance hierarchies in object/relational mapping tools. In: Proc. CAiSE, *LNCS*, vol. 3520, pp. 135–150. Springer (2005)
54. Callaú, O., Robbes, R., Tanter, É., Röthlisberger, D.: How (and why) developers use the dynamic features of programming languages: The case of Smalltalk. Empir. Softw. Eng. **18**(6), 1156–1194 (2013)
55. Canfora, G., Fasolino, A.R., Frattolillo, G., Tramontana, P.: A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. J. Syst. Softw. **81**(4), 463–480 (2008)
56. Ceh, I., Crepinsek, M., Kosar, T., Mernik, M.: Ontology driven development of domain-specific languages. Comput. Sci. Inf. Syst. **8**(2), 317–342 (2011)
57. Chaparro, O., Bavota, G., Marcus, A., Penta, M.D.: On the impact of refactoring operations on code quality metrics. In: Proc. ICSME, pp. 456–460. IEEE (2014)
58. Chen, L.J., Bernstein, P.A., Carlin, P., Filipovic, D., Rys, M., Shamgunov, N., Terwilliger, J.F., Todic, M., Tomasevic, S., Tomic, D.: Mapping XML to a wide sparse table. IEEE Trans. Knowl. Data Eng. **26**(6), 1400–1414 (2014)
59. Chikofsky, E.J., II, J.H.C.: Reverse engineering and design recovery: A taxonomy. IEEE Softw. **7**(1), 13–17 (1990)
60. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: Proc. ECOC, pp. 222–231. IEEE (2008)
61. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A metamodel independent approach to difference representation. J. Object Technol. **6**(9), 165–185 (2007)
62. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: Proc. ICMT, *LNCS*, vol. 5563, pp. 35–51. Springer (2009)

63. Cimitile, A., de Carlini, U., Lucia, A.D.: Incremental migration strategies: Data flow analysis for wrapping. In: Proc. WCRE, pp. 59–68. IEEE (1998)
64. Cleland-Huang, J., Gotel, O., Zisman, A. (eds.): Software and Systems Traceability. Springer (2012)
65. Cleve, A.: Automating program conversion in database reengineering: A wrapper-based approach. In: Proc. CSMR, pp. 323–326. IEEE (2006)
66. Cleve, A., Hainaut, J.: Co-transformations in database applications evolution. In: GTTSE 2005, Revised Papers, *LNCS*, vol. 4143, pp. 409–421. Springer (2006)
67. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
68. Cook, W.R., Lämmel, R.: Tutorial on online partial evaluation. In: Proc. DSL, *EPTCS*, vol. 66, pp. 168–180 (2011)
69. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. **61**(3), 190–210 (2006)
70. Cordy, J.R.: Excerpts from the TXL cookbook. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 27–91. Springer (2011)
71. Cuadrado, J.S., Guerra, E., de Lara, J.: *The Program Is the Model*: Enabling transformations@run.time. In: Proc. SLE 2012, *LNCS*, vol. 7745, pp. 104–123. Springer (2013)
72. Cunha, A., Oliveira, J.N., Visser, J.: Type-safe two-level data transformation. In: Proc. FM, *LNCS*, vol. 4085, pp. 284–299. Springer (2006)
73. Cunha, A., Visser, J.: Strongly typed rewriting for coupled software transformation. ENTCS **174**(1), 17–34 (2007)
74. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional (2000)
75. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–646 (2006)
76. Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In: Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers, *LNCS*, vol. 3016, pp. 51–72. Springer (2004)
77. Dam, H.K., Egyed, A., Winikoff, M., Reder, A., Lopez-Herrejon, R.E.: Consistent merging of model versions. J. Syst. Softw. **112**, 137–155 (2016)
78. Degueule, T.: Composition and interoperability for external domain-specific language engineering. Ph.D. thesis, Université de Rennes 1 (2016)
79. Demeyer, S., Ducasse, S., Lanza, M.: A hybrid reverse engineering approach combining metrics and program visualization. In: Proc. WCRE, pp. 175–186. IEEE (1999)
80. Derakhshanmanesh, M., Ebert, J., Iguchi, T., Engels, G.: Model-integrating software components. In: Proc. MODELS, *LNCS*, vol. 8767, pp. 386–402. Springer (2014)
81. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Not. **35**(6), 26–36 (2000)
82. Diskin, Z.: Model synchronization: Mappings, tiles, and categories. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 92–165. Springer (2011)
83. Diskin, Z., Gholizadeh, H., Wider, A., Czarnecki, K.: A three-dimensional taxonomy for bidirectional model synchronization. J. Syst. Softw. **111**, 298–322 (2016)
84. Diskin, Z., Kokaly, S., Maibaum, T.: Mapping-aware megamodeling: Design patterns and laws. In: Proc. SLE 2013, *LNCS*, vol. 8225, pp. 322–343. Springer (2013)
85. Diskin, Z., Wider, A., Gholizadeh, H., Czarnecki, K.: Towards a rational taxonomy for increasingly symmetric model synchronization. In: Proc. ICMT, *LNCS*, vol. 8568, pp. 57–73. Springer (2014)
86. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: A taxonomy and survey. J. Softw.: Evol. Process **25**(1), 53–95 (2013)
87. Dit, B., Wagner, M., Wen, S., Wang, W., Vásquez, M.L., Poshyvanyk, D., Kagdi, H.H.: ImpactMiner: A tool for change impact analysis. In: Proc. ICSE, pp. 540–543. ACM (2014)
88. Dittrich, K.R., Gatziu, S., Geppert, A.: The active database management system manifesto: A rulebase of ADBMS features. In: Proc. RIDS, *LNCS*, vol. 985, pp. 3–20. Springer (1995)

89. Djuric, D., Gasevic, D., Devedzic, V.: The tao of modeling spaces. J. Object Technol. **5**(8), 125–147 (2006)
90. Doyle, J.R., Stretch, D.D.: The classification of programming languages by usage. Int. J. Man–Machine Stud. **26**(3), 343–360 (1987)
91. Ducasse, S., Gîrba, T., Kuhn, A., Renggli, L.: Meta-environment and executable meta-language using Smalltalk: An experience report. SoSyM **8**(1), 5–19 (2009)
92. Emden, E.V., Moonen, L.: Java quality assurance by detecting code smells. In: Proc. WCRE, p. 97. IEEE (2002)
93. Engel, K., Paige, R.F., Kolovos, D.S.: Using a model merging language for reconciling model versions. In: Proc. ECMDA-FA, *LNCS*, vol. 4066, pp. 143–157. Springer (2006)
94. Erdweg, S.: Extensible languages for flexible and principled domain abstraction. Ph.D. thesis, Philipps-Universität Marburg (2013)
95. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proc. LDTA, p. 7. ACM (2012)
96. Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: The state of the art in language workbenches – conclusions from the language workbench challenge. In: Proc. SLE, *LNCS*, vol. 8225, pp. 197–217. Springer (2013)
97. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015)
98. Erwig, M., Walkingshaw, E.: Semantics first! – rethinking the language design process. In: Proc. SLE 2011, *LNCS*, vol. 6940, pp. 243–262. Springer (2012)
99. Favre, J.: Preprocessors from an abstract point of view. In: Proc. WCRE, pp. 287–296. IEEE (1996)
100. Favre, J., Gasevic, D., Lämmel, R., Pek, E.: Empirical language analysis in software linguistics. In: Proc. SLE 2010, *LNCS*, vol. 6563, pp. 316–326. Springer (2011)
101. Favre, J., Lämmel, R., Leinberger, M., Schmorleiz, T., Varanovich, A.: Linking documentation and source code in a software chrestomathy. In: Proc. WCRE, pp. 335–344. IEEE (2012)
102. Favre, J., Lämmel, R., Schmorleiz, T., Varanovich, A.: 101companies: A community project on software technologies and software languages. In: Proc. TOOLS, *LNCS*, vol. 7304, pp. 58–74. Springer (2012)
103. Favre, J., Lämmel, R., Varanovich, A.: Modeling the linguistic architecture of software products. In: Proc. MODELS, *LNCS*, vol. 7590, pp. 151–167. Springer (2012)
104. Favre, J.M.: Foundations of meta-pyramids: Languages vs. metamodels – Episode II: Story of Thotus the baboon. In: Language Engineering for Model-Driven Software Development, no. 04101 in Dagstuhl Seminar Proceedings (2005)
105. Favre, J.M.: Foundations of model (driven) (reverse) engineering: Models – Episode I: Stories of the Fidus Papyrus and of the Solarus. In: Language Engineering for Model-Driven Software Development, no. 04101 in Dagstuhl Seminar Proceedings (2005)
106. Favre, J.M., Gasevic, D., Lämmel, R., Winter, A.: Guest editors' introduction to the special section on software language engineering. IEEE Trans. Softw. Eng. **35**(6), 737–741 (2009)
107. Felleisen, M., Findler, R., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2009)
108. Fenton, N.E., Pfleeger, S.L.: Software metrics – A practical and rigorous approach. International Thomson (1996). 2nd edition
109. Ferenc, R., Siket, I., Gyimóthy, T.: Extracting facts from open source software. In: Proc. ICSM, pp. 60–69. IEEE (2004)
110. Fokin, A., Derevenetc, E., Chernov, A., Troshina, K.: SmartDec: Approaching C++ decompilation. In: Proc. WCRE, pp. 347–356. IEEE (2011)

111. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Leveraging variability modeling to address metamodel revisions in model-based software product lines. Comput. Lang. Syst. Struct. **48**, 20–38 (2017)
112. Fontana, F.A., Caracciolo, A., Zanoni, M.: DPB: A benchmark for design pattern detection tools. In: CSMR 2012, pp. 235–244. IEEE (2012)
113. Fontana, F.A., Ferme, V., Marino, A., Walter, B., Martenka, P.: Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In: Proc. ICSM, pp. 260–269. IEEE (2013)
114. Fontana, F.A., Zanoni, M.: A tool for design pattern detection and software architecture reconstruction. Inf. Sci. **181**(7), 1306–1324 (2011)
115. Fontana, F.A., Zanoni, M., Marino, A., Mäntylä, M.: Code smell detection: Towards a machine learning-based approach. In: Proc. ICSM, pp. 396–399. IEEE (2013)
116. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
117. Frühwirth, T., Abdennadher, S.: Essentials of constraint programming. Springer (2003)
118. Galvão, I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: Proc. EDOC, pp. 313–326. IEEE (2007)
119. Gethers, M., Dit, B., Kagdi, H.H., Poshyvanyk, D.: Integrated impact analysis for managing software changes. In: Proc. ICSE, pp. 430–440. IEEE (2012)
120. Gil, J., Maman, I.: Micro patterns in Java code. In: Proc. OOPSLA, pp. 97–116. ACM (2005)
121. Gomes, C., Barroca, B., Amaral, V.: Classification of model transformation tools: Pattern matching techniques. In: Proc. MODELS, *LNCS*, vol. 8767, pp. 619–635. Springer (2014)
122. Gonçales, L., Farias, K., Scholl, M., Veronez, M., de Oliveira, T.C.: Comparison of design models: A systematic mapping study. Int. J. Softw. Eng. Knowl. Eng. **25**(9-10), 1765–1770 (2015)
123. Goncharenko, B., Zaytsev, V.: Language design and implementation for the domain of coding conventions. In: Proc. SLE, pp. 90–104. ACM (2016)
124. Hainaut, J.: The transformational approach to database engineering. In: GTTSE 2005, Revised Papers, *LNCS*, vol. 4143, pp. 95–143. Springer (2006)
125. Hainaut, J., Cleve, A., Henrard, J., Hick, J.: Migration of legacy information systems. In: Software Evolution, pp. 105–138. Springer (2008)
126. Han, M., Hofmeister, C., Nord, R.L.: Reconstructing software architecture for J2EE web applications. In: Proc. WCRE, pp. 67–79. IEEE (2003)
127. Härtel, J., Härtel, L., Heinz, M., Lämmel, R., Varanovich, A.: Interconnected linguistic architecture. The Art, Science, and Engineering of Programming Journal **1** (2017). 27 pages. Available at http://programming-journal.org/2017/1/3/
128. Hassan, A.E., Jiang, Z.M., Holt, R.C.: Source versus object code extraction for recovering software architecture. In: Proc. WCRE, pp. 67–76. IEEE (2005)
129. Hatcliff, J.: Foundations of partial evaluation and program specialization (1999). Available at http://people.cis.ksu.edu/~hatcliff/FPEPS/
130. Henkel, J., Diwan, A.: CatchUp!: capturing and replaying refactorings to support API evolution. In: Proc. ICSE, pp. 274–283. ACM (2005)
131. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: Automatability of coupled evolution of metamodels and models in practice. In: Proc. MoDELS, *LNCS*, vol. 5301, pp. 645–659. Springer (2008)
132. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: COPE – Automating coupled evolution of metamodels and models. In: Proc. ECOOP, *LNCS*, vol. 5653, pp. 52–76. Springer (2009)
133. Hoare, C.A.R.: Hints on programming language design. Tech. rep., Stanford University (1973)
134. Holt, R.C.: Structural manipulations of software architecture using Tarski relational algebra. In: Proc. WCRE, pp. 210–219. IEEE (1998)
135. Holt, R.C.: WCRE 1998 most influential paper: Grokking software architecture. In: Proc. WCRE, pp. 5–14. IEEE (2008)
136. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)

137. ISO/IEC: ISO/IEC 14977:1996(E). Information Technology. Syntactic Metalanguage. Extended BNF. (1996). Available at http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf
138. Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S.: Understanding and detecting real-world performance bugs. In: Proc. PLDI, pp. 77–88. ACM (2012)
139. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Inc. (1993)
140. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1-2), 31–39 (2008)
141. Jouault, F., Vanhooff, B., Brunelière, H., Doux, G., Berbers, Y., Bézivin, J.: Inter-DSL coordination support by combining megamodeling and model weaving. In: Proc. SAC, pp. 2011–2018. ACM (2010)
142. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: A survey of the first wave. In: Conceptual Modelling and Its Theoretical Foundations – Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday, *LNCS*, vol. 7260, pp. 197–215. Springer (2012)
143. Kats, L.C.L., Visser, E.: The Spoofax language workbench. In: Companion SPLASH/OOPSLA, pp. 237–238. ACM (2010)
144. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Proc. OOPSLA, pp. 444–463. ACM (2010)
145. Keenan, E., Czauderna, A., Leach, G., Cleland-Huang, J., Shin, Y., Moritz, E., Gethers, M., Poshyvanyk, D., Maletic, J.I., Hayes, J.H., Dekhtyar, A., Manukian, D., Hossein, S., Hearn, D.: TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In: Proc. ICSE, pp. 1375–1378. IEEE (2012)
146. Kehrer, T., Kelter, U., Pietsch, P., Schmidt, M.: Operation-based model differencing meets state-based model comparison. Softwaretechnik-Trends **32**(4) (2012)
147. Kelly, S., Tolvanen, J.: Domain-Specific Modeling. IEEE & Wiley (2008)
148. Kienle, H.M., Müller, H.A.: Rigi – An environment for software reverse engineering, exploration, visualization, and redocumentation. Sci. Comput. Program. **75**(4), 247–263 (2010)
149. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. **14**(3), 331–380 (2005)
150. Klint, P., van der Storm, T., Vinju, J.J.: RASCAL: A domain specific language for source code analysis and manipulation. In: Proc. SCAM, pp. 168–177. IEEE (2009)
151. Klint, P., van der Storm, T., Vinju, J.J.: EASY meta-programming with Rascal. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 222–289. Springer (2011)
152. Klusener, A.S., Lämmel, R., Verhoef, C.: Architectural modifications to deployed software. Sci. Comput. Program. **54**(2-3), 143–211 (2005)
153. Kniesel, G., Binun, A.: Standing on the shoulders of giants – A data fusion approach to design pattern detection. In: Proc. ICPC, pp. 208–217. IEEE (2009)
154. Kniesel, G., Binun, A., Hegedüs, P., Fülöp, L.J., Chatzigeorgiou, A., Guéhéneuc, Y., Tsantalis, N.: DPDX–towards a common result exchange format for design pattern detection tools. In: Proc. CSMR, pp. 232–235. IEEE (2010)
155. Knuth, D.E.: An empirical study of FORTRAN programs. Softw., Pract. Exper. **1**(2), 105–133 (1971)
156. Kolovos, D.S., Paige, R.F., Polack, F.: Merging models with the Epsilon Merging Language (EML). In: Proc. MoDELS, *LNCS*, vol. 4199, pp. 215–229. Springer (2006)
157. Kosar, T., Bohra, S., Mernik, M.: Domain-specific languages: A systematic mapping study. Inf. Softw. Technol. **71**, 77–91 (2016)
158. Koschke, R.: Architecture reconstruction. In: Software Engineering, International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures, *LNCS*, vol. 5413, pp. 140–173. Springer (2009)
159. Kramer, M.E., Klein, J., Steel, J.R.H., Morin, B., Kienzle, J., Barais, O., Jézéquel, J.: Achieving practical genericity in model weaving through extensibility. In: Proc. ICMT, *LNCS*, vol. 7909, pp. 108–124. Springer (2013)

160. Krishnamurthi, S.: Programming Languages: Application and Interpretation. Brown University (2007). https://cs.brown.edu/~sk/Publications/Books/ProgLangs/
161. Kurtev, I., Bézivin, J., Akşit, M.: Technological spaces: An initial appraisal. In: Proc. CoopIS, DOA 2002, Industrial track (2002)
162. Lämmel, R.: Coupled software transformations. In: Proc. SET, pp. 31–35 (2004). Extended Abstract. Available at http://post.queensu.ca/~zouy/files/set-2004.pdf#page=38
163. Lämmel, R.: LINQ to XSD. In: Proc. PLAN-X, pp. 95–96 (2007)
164. Lämmel, R.: Coupled software transformations revisited. In: Proc. SLE, pp. 239–252. ACM (2016)
165. Lämmel, R.: Relationship maintenance in software language repositories. The Art, Science, and Engineering of Programming Journal **1** (2017). 27 pages. Available at http://programming-journal.org/2017/1/4/
166. Lämmel, R., Leinberger, M., Schmorleiz, T., Varanovich, A.: Comparison of feature implementations across languages, technologies, and styles. In: Proc. CSMR-WCRE, pp. 333–337. IEEE (2014)
167. Lämmel, R., Linke, R., Pek, E., Varanovich, A.: A framework profile of .NET. In: Proc. WCRE, pp. 141–150. IEEE (2011)
168. Lämmel, R., Lohmann, W.: Format evolution. In: Proc. RETIS, vol. 155, pp. 113—-134. OCG, books@ocg.at (2001)
169. Lämmel, R., Meijer, E.: Mappings make data processing go 'round. In: GTTSE 2005, Revised Papers, *LNCS*, vol. 4143, pp. 169–218. Springer (2006)
170. Lämmel, R., Meijer, E.: Revealing the X/O impedance mismatch – (changing lead into gold). In: Datatype-Generic Programming – International Spring School, SSDGP 2006, Revised Lectures, *LNCS*, vol. 4719, pp. 285–367. Springer (2007)
171. Lämmel, R., Mosen, D., Varanovich, A.: Method and tool support for classifying software languages with Wikipedia. In: Proc. SLE, *LNCS*, vol. 8225, pp. 249–259. Springer (2013)
172. Lämmel, R., Pek, E.: Understanding privacy policies – A study in empirical analysis of language usage. Empir. Softw. Eng. **18**(2), 310–374 (2013)
173. Lämmel, R., Schmorleiz, T., Varanovich, A.: The 101haskell chrestomathy: A whole bunch of learnable lambdas. In: Proc. IFL, p. 25. ACM (2013)
174. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In: Proc. TestCom, *LNCS*, vol. 3964, pp. 19–38. Springer (2006)
175. Lämmel, R., Varanovich, A.: Interpretation of linguistic architecture. In: Proc. ECMFA, *LNCS*, vol. 8569, pp. 67–82. Springer (2014)
176. Lämmel, R., Zaytsev, V.: Recovering grammar relationships for the Java language specification. Softw. Qual. J. **19**(2), 333–378 (2011)
177. Lanza, M., Ducasse, S.: Polymetric views – A lightweight visual approach to reverse engineering. IEEE Trans. Softw. Eng. **29**(9), 782–795 (2003)
178. Lanza, M., Ducasse, S., Demeyer, S.: Reverse engineering based on metrics and program visualization. In: ECOOP 1999 Workshop Reader, *LNCS*, vol. 1743, pp. 168–169. Springer (1999)
179. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice – Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer (2006)
180. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO, pp. 75–88. IEEE (2004)
181. Lédeczi, Á., Bakay, A., Maroti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. IEEE Computer **34**(11), 44–51 (2001)
182. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Tech. Rep. UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht (2001)
183. Li, B., Sun, X., Leung, H., Zhang, S.: A survey of code-based change impact analysis techniques. Softw. Test., Verif. Reliab. **23**(8), 613–646 (2013)

184. Liebig, J., Kästner, C., Apel, S.: Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In: Proc. AOSD, pp. 191–202. ACM (2011)
185. Lin, Y., Holt, R.C.: Formalizing fact extraction. ENTCS **94**, 93–102 (2004)
186. Louden, K.: Compiler Construction: Principles and Practice. Cengage Learning (1997)
187. Lungu, M., Lanza, M., Gîrba, T., Robbes, R.: The small project observatory: Visualizing software ecosystems. Sci. Comput. Program. **75**(4), 264–275 (2010)
188. Mäder, P., Egyed, A.: Do developers benefit from requirements traceability when evolving and maintaining a software system? Empir. Softw. Eng. **20**(2), 413–441 (2015)
189. Marchand, J.Y., Combemale, B., Baudry, B.: A categorical model of model merging and weaving. In: Proc. MiSE, pp. 70–76. IEEE (2012)
190. Marriott, K., Meyer, B.: On the classification of visual languages by grammar hierarchies. J. Vis. Lang. Comput. **8**(4), 375–402 (1997)
191. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Automating the extraction of model-based software product lines from model variants (T). In: Proc. ASE, pp. 396–406. IEEE (2015)
192. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)
193. Mehner, K., Monga, M., Taentzer, G.: Analysis of aspect-oriented model weaving. In: Trans. Aspect-Oriented Software Development, vol. 5, pp. 235–263 (2009)
194. Mens, T.: Model Transformation: A Survey of the State of the Art, pp. 1–19. John Wiley & Sons, Inc. (2013)
195. Mens, T., Gorp, P.V.: A taxonomy of model transformation. ENTCS **152**, 125–142 (2006)
196. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
197. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
198. Mordal-Manet, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., Ducasse, S.: Software quality metrics aggregation in industry. J. Softw.: Evol. Process **25**(10), 1117–1135 (2013)
199. Mosses, P.D.: Action Semantics. Cambridge University Press (1992)
200. Murakami, T., Amagasa, T., Kitagawa, H.: DBPowder: A flexible object-relational mapping framework based on a conceptual model. In: Proc. COMPSAC, pp. 589–598. IEEE (2013)
201. Murphy, G.C., Notkin, D.: Lightweight lexical source model extraction. ACM Trans. Softw. Eng. Methodol. **5**(3), 262–292 (1996)
202. Neubauer, P., Bergmayr, A., Mayerhofer, T., Troya, J., Wimmer, M.: XMLText: from XML schema to Xtext. In: Proc. SLE, pp. 71–76. ACM (2015)
203. Paige, R.F., Kolovos, D.S., Polack, F.A.C.: A tutorial on metamodelling for grammar researchers. Sci. Comput. Program. **96**, 396–416 (2014)
204. Paige, R.F., Matragkas, N.D., Rose, L.M.: Evolving models in model-driven engineering: State-of-the-art and future challenges. J. Syst. Softw. **111**, 272–280 (2016)
205. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Poshyvanyk, D., Lucia, A.D.: Mining version histories for detecting code smells. IEEE Trans. Softw. Eng. **41**(5), 462–489 (2015)
206. Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M.: Systematic mapping studies in software engineering. In: Proc. EASE, Workshops in Computing. BCS (2008)
207. Petersen, K., Vakkalanka, S., Kuzniarz, L.: Guidelines for conducting systematic mapping studies in software engineering: An update. Inf. Softw. Technol. **64**, 1–18 (2015)
208. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoc, D.: Foundations of JSON schema. In: Proc. WWW, pp. 263–273. ACM (2016)
209. Pfleeger, S.L.: Software metrics: Progress after 25 years? IEEE Softw. **25**(6), 32–34 (2008)
210. Pierce, B.: Types and Programming Languages. MIT Press (2002)
211. Porkoláb, Z., Sinkovics, Á., Siroki, I.: DSL in C++ template metaprogram. In: CEFP 2013, Revised Selected Papers, *LNCS*, vol. 8606, pp. 76–114. Springer (2015)
212. Rajlich, V.: A model and a tool for change propagation in software. ACM SIGSOFT Software Engineering Notes **25**(1), 72 (2000)

213. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.C.: Chianti: A tool for change impact analysis of Java programs. In: Proc. OOPSLA, pp. 432–448. ACM (2004)
214. Renggli, L.: Dynamic language embedding with homogeneous tool support. Ph.D. thesis, Universität Bern (2010)
215. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: Proc. ECOOP, *LNCS*, vol. 6183, pp. 380–404. Springer (2010)
216. Rocco, J.D., Ruscio, D.D., Iovino, L., Pierantonio, A.: Dealing with the coupled evolution of metamodels and model-to-text transformations. In: Proc. Workshop on Models and Evolution, *CEUR Workshop Proceedings*, vol. 1331, pp. 22–31. CEUR-WS.org (2015)
217. Rompf, T.: The essence of multi-stage evaluation in LMS. In: A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, *LNCS*, vol. 9600, pp. 318–335. Springer (2016)
218. Roover, C.D.: A logic meta-programming foundation for example-driven pattern detection in object-oriented programs. In: Proc. ICSM, pp. 556–561. IEEE (2011)
219. Roover, C.D., Lämmel, R., Pek, E.: Multi-dimensional exploration of API usage. In: Proc. ICPC 2013, pp. 152–161. IEEE (2013)
220. Rosa, M.L., Dumas, M., Uba, R., Dijkman, R.M.: Business process model merging: An approach to business process consolidation. ACM Trans. Softw. Eng. Methodol. **22**(2), 11 (2013)
221. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010)
222. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (2009)
223. Roy, C.K., Zibran, M.F., Koschke, R.: The vision of software clone management: Past, present, and future (keynote paper). In: Proc. CSMR-WCRE, pp. 18–33. IEEE (2014)
224. Rubin, J., Chechik, M.: A survey of feature location techniques. In: Domain Engineering, Product Lines, Languages, and Conceptual Models, pp. 29–58. Springer (2013)
225. Ruscio, D.D., Lämmel, R., Pierantonio, A.: Automated co-evolution of GMF editor models. In: Proc. SLE 2010, *LNCS*, vol. 6563, pp. 143–162. Springer (2011)
226. Sánchez, P., Fuentes, L., Stein, D., Hanenberg, S., Unland, R.: Aspect-oriented model weaving beyond model composition and model transformation. In: Proc. MoDELS, *LNCS*, vol. 5301, pp. 766–781. Springer (2008)
227. Savga, I., Rudolf, M., Goetz, S., Aßmann, U.: Practical refactoring-based framework upgrade. In: Proc. GPCE, pp. 171–180. ACM (2008)
228. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A comprehensive approach to naming and accessibility in refactoring Java programs. IEEE Trans. Softw. Eng. **38**(6), 1233–1257 (2012)
229. Schauss, S., Lämmel, R., Härtel, J., Heinz, M., Klein, K., Härtel, L., Berger, T.: A chrestomathy of DSL implementations. In: Proc. SLE. ACM (2017). 12 pages
230. Schink, H., Kuhlemann, M., Saake, G., Lämmel, R.: Hurdles in multi-language refactoring of Hibernate applications. In: Proc. ICSOFT, pp. 129–134. SciTePress (2011)
231. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006)
232. Sebesta, R.W.: Concepts of Programming Languages. Addison-Wesley (2012). 10th edition
233. Seibel, A., Hebig, R., Giese, H.: Traceability in model-driven engineering: Efficient and scalable traceability maintenance. In: Software and Systems Traceability., pp. 215–240. Springer (2012)
234. Sequeda, J., Arenas, M., Miranker, D.P.: On directly mapping relational databases to RDF and OWL. In: Proc WWW 2012, pp. 649–658. ACM (2012)
235. Sereni, D., Jones, N.D.: Termination analysis of higher-order functional programs. In: Proc. APLAS, *LNCS*, vol. 3780, pp. 281–297. Springer (2005)
236. Sheard, T., Peyton Jones, S.L.: Template meta-programming for Haskell. SIGPLAN Not. **37**(12), 60–75 (2002)
237. Shilov, N.V., Akinin, A.A., Zubkov, A.V., Idrisov, R.I.: Development of the computer language classification knowledge portal. In: Perspectives of Systems Informatics – 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27–July 1, 2011, Revised Selected Papers, *LNCS*, vol. 7162, pp. 340–348. Springer (2012)

238. Singh, R., Bezemer, C., Shang, W., Hassan, A.E.: Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm. In: Proc. ICPE, pp. 309–320. ACM (2016)
239. Skalna, I., Gawel, B.: Model driven architecture and classification of business rules modelling languages. In: Proc. FedCSIS, pp. 949–952 (2012)
240. Sneed, H.M.: Wrapping legacy COBOL programs behind an XML-interface. In: Proc. WCRE, p. 189. IEEE (2001)
241. Sneed, H.M.: Migrating PL/I code to Java. In: Proc. CSMR, pp. 287–296. IEEE (2011)
242. Sneed, H.M., Majnar, R.: A case study in software wrapping. In: Proc. ICSM, pp. 86–93. IEEE (1998)
243. Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Alias analysis for object-oriented programs. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification, *LNCS*, vol. 7850, pp. 196–232. Springer (2013)
244. Stephan, M., Cordy, J.R.: A survey of model comparison approaches and applications. In: Proc. MODELSWARD, pp. 265–277. SciTePress (2013)
245. Stinckwich, S., Ducasse, S.: Introduction to the Smalltalk special issue. Comput. Lang. Syst. Struct. **32**(2-3), 85–86 (2006)
246. Sun, X., Li, B., Leung, H., Li, B., Zhu, J.: Static change impact analysis techniques: A comparative study. J. Syst. Softw. **109**, 137–149 (2015)
247. Sztipanovits, J., Karsai, G.: Model-integrated computing. IEEE Computer **30**(4), 110–111 (1997)
248. Taha, W.: A gentle introduction to multi-stage programming, part II. In: GTTSE 2007, Revised Papers, *LNCS*, vol. 5235, pp. 260–290. Springer (2008)
249. Tamura, G., Cleve, A.: A comparison of taxonomies for model transformation languages. Paradigma **4**(1), 1–14 (2010)
250. Terekhov, A.A., Verhoef, C.: The realities of language conversions. IEEE Softw. **17**(6), 111–124 (2000)
251. Terwilliger, J.F., Bernstein, P.A., Melnik, S.: Full-fidelity flexible object-oriented XML access. PVLDB **2**(1), 1030–1041 (2009)
252. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**(3) (1995)
253. Tip, F., Fuhrer, R.M., Kiezun, A., Ernst, M.D., Balaban, I., Sutter, B.D.: Refactoring using type constraints. ACM Trans. Program. Lang. Syst. **33**(3), 9 (2011)
254. Torres, A., de Matos Galante, R., Pimenta, M.S.: ENORM: An essential notation for object-relational mapping. SIGMOD Record **43**(2), 23–28 (2014)
255. Tratt, L.: Domain specific language implementation via compile-time meta-programming. ACM Trans. Program. Lang. Syst. **30**(6) (2008)
256. Trujillo, S., Azanza, M., Díaz, O.: Generative metaprogramming. In: Proc. GPCE, pp. 105–114. ACM (2007)
257. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M.D., Lucia, A.D., Poshyvanyk, D.: When and why your code starts to smell bad. In: Proc. ICSE, pp. 403–414. IEEE (2015)
258. van Roy, P.: Programming paradigms for dummies: What every programmer should know. In: New Computational Paradigms for Computer Music, IRCAM/Delatour, France, pp. 9–38 (2009)
259. Vasilescu, B., Serebrenik, A., van den Brand, M.: You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In: ICSM 2011, pp. 313–322. IEEE (2011)
260. Vermolen, S., Visser, E.: Heterogeneous coupled evolution of software languages. In: Proc. MoDELS, *LNCS*, vol. 5301, pp. 630–644. Springer (2008)
261. Vermolen, S.D., Wachsmuth, G., Visser, E.: Generating database migrations for evolving web applications. In: Proc. GPCE, pp. 83–92. ACM (2011)
262. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: Proc. RTA, *LNCS*, vol. 2051, pp. 357–362. Springer (2001)
263. Visser, E., Wachsmuth, G., Tolmach, A.P., Neron, P., Vergu, V.A., Passalaqua, A., Konat, G.: A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In: Proc. SPLASH, Onward!, pp. 95–111. ACM (2014)

264. Visser, J.: Coupled transformation of schemas, documents, queries, and constraints. ENTCS **200**(3), 3–23 (2008)
265. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering – Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
266. Voelter, M., Ratiu, D., Kolb, B., Schätz, B.: mbeddr: instantiating a language workbench in the embedded software domain. Autom. Softw. Eng. **20**(3), 339–390 (2013)
267. Völter, M., Visser, E.: Language extension and composition with language workbenches. In: Companion SPLASH/OOPSLA, pp. 301–304. ACM (2010)
268. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Proc. ECOOP, *LNCS*, vol. 4609, pp. 600–624. Springer (2007)
269. Wachsmuth, G., Konat, G.D.P., Visser, E.: Language design with the Spoofax language workbench. IEEE Softw. **31**(5), 35–43 (2014)
270. Wang, J., Peng, X., Xing, Z., Zhao, W.: How developers perform feature location tasks: A human-centric and process-oriented exploratory study. J. Softw.: Evol. Process **25**(11), 1193–1224 (2013)
271. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)
272. Wile, D.S.: Lessons learned from real DSL experiments. In: Proc. HICSS-36, p. 325. IEEE (2003)
273. Wile, D.S.: Lessons learned from real DSL experiments. Sci. Comput. Program. **51**(3), 265–290 (2004)
274. WWW: XML Information Set (2004). https://www.w3.org/TR/xml-infoset/
275. Yamashita, A.F., Moonen, L.: Do code smells reflect important maintainability aspects? In: Proc. ICSM, pp. 306–315. IEEE (2012)
276. Zanoni, M., Fontana, F.A., Stella, F.: On applying machine learning techniques for design pattern detection. J. Syst. Softw. **103**, 102–117 (2015)
277. Zaytsev, V.: BNF WAS HERE: What have we done about the unnecessary diversity of notation for syntactic definitions. In: Proc. SAC, pp. 1910–1915. ACM (2012)
278. Zaytsev, V., et al.: Software language processing suite (2008). http://slps.github.io/