Review of the Support for Modular Language Implementation with Embedding Approaches

Tom Dinkelaker
email: dinkelaker@cs.tu-darmstadt.de
date: November 12, 2010
technical report number: TUD-CS-2010-2396

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachgebiet Softwaretechnik
Fachbereich Informatik
Technische Universität Darmstadt
Germany

Abstract

Embedded domain-specific languages (DSLs) are a new light-weight approach to implement DSLs with lower initial costs than traditional non-embedded DSL approaches. However, existing embedded DSL approaches only support a subset of DSLs that can be implemented with traditional non-embedded approaches. This is because existing embedding approaches lack support for important requirements that currently are only supported by traditional non-embedded approaches. This technical report identifies important requirements for language implementation. It gives an extensive review of the support for a selection of desirable properties by embedding approaches that address these requirements. The review explains details of the available mechanisms in existing embedding approaches; it identifies open issues and limitations of the current techniques. To overcome current shortcomings of embedded DSL approaches, the review proposes a roadmap for the research in techniques for embedding DSLs. For the roadmap, the review draws conclusions from studying the available support for the desirable properties in related work on traditional non-embedded approaches.

Contents

1 Introduction

In recent years, there has been an increasing interest in new languages that provide special syntax and semantics for certain problem and technical domains, so called *domain-specific languages* (DSLs) [vDKV00, MHS05]. Because most DSLs provide a *concrete syntax* that is closer to its problem domain than a general-purpose language (GPL), they allow for higher end-user productivity [MHS05, KLP+08]. Furthermore, DSLs provide domain-specific abstractions and constraints, which provide opportunities for analysis and optimizations [vDKV00, MHS05]. Famous examples of DSLs are BNF,[1] SQL, and HTML.

Many DSLs are virtually indispensable tools for *language end users* to efficiently implement software artifacts for special problem domains. However, in general, developing a new DSL creates large costs for the language developer. A language developers needs to implement the infrastructure for this DSL. DSL artifacts need to be integrated with other artifacts written in GPLs. Hence, there are increasing requests to compose DSLs with existing GPLs [ME00, BV04, TFH09], e.g. SQLj [ME00] composes SQL [DD89] and *Java*™ [LY99]. These increasing amount of requests for new DSL integrations create a challenge for traditional language implementation approaches that have little support for extensibility [BV04, Cor06, HM03].

A light-weight approach to DSLs that addresses parts of these problems is to *embed* a language into an existing language [Lan66, Hud96, Hud98, Kam98]. The existing language serves as a *host language* for implementing the *embedded language*. Following this approach, the embedded language can reuse the general-purpose features of the host language. Consequently, when the embedded language shares common language constructs with the host language, this reduces the costs to implement the DSL. Further, the approaches eliminates the costs for developing a special infrastructure for the embedded language by reusing the existing host language infrastructure [Hud98, Kam98, KLP+08], i.e., existing development tools, parsers, compilers, and virtual machines.

From the language developer's perspective, there are interesting benefits when following the embedded approach. New language features can be added incrementally by "simply" extending the corresponding libraries. Embeddings are easier to compose in contrast to languages that are implemented with traditional approaches, such as pre-processors, interpreters, and compilers [MHS05]. Those benefits are a competitive advantage over traditional language implementation approaches.

However, embedding approaches also have important drawback compared to traditional language implementation approaches. Existing embedding approaches have no full support for many properties that would be available when using a tradition approach. Most importantly, in embedding approaches, there is a lack of support for extensibility with semantic adaptations, composition of interacting languages, new composition mechanisms, partial concrete syntax, pluggable scoping, analyses and transformations.

This technical report identifies important requirements for language implementation. It gives an extensive review of the support for a selection of desirable properties by embedding approaches that address these requirements. The review explains details of the available mechanisms in existing embedding approaches, it identifies open issues and limitations of the current techniques. To overcome current shortcomings of embedded approaches, the review proposes a roadmap for the research in techniques for embedding DSLs. For the roadmap, the review draws conclusions from studying the available support for the desirable properties in related work on traditional non-embedded approaches.

The remainder of this report is structured as follows. Section 2 surveys existing embedding approaches. Section 3 identifies a set of desirable properties for language implementation that developers expect support for—not only by traditional non-embedded approaches but also by embedded approaches. Then, Section 4 reviews the current support for these properties by embedding approaches and draws the future research roadmap. Finally, Section 5 summarizes the review results.

[1] Backus-Naur Form

2 Embedding Approaches

Various embedding approaches propose using different host languages, namely languages that allow *pure functional programming* [Hud98, CKS09, ALY09], *dynamic programming* [Pes01, TFH09, KG07, KM09, RGN10, AO10], *staging* and *meta-programming* [COST04, SCK04, Tra08], *strong-typing* for classed-based object oriented programming [Eva03, Fow05, Gar08, Dub06, HO10], *generative programming* [Kam98, EFDM03, CM07], and *source code transformations* [BV04, Cor06].

Using these approaches, DSLs have been implemented for various domains, such as for *mathematical calculations* [Hud98, COST04, Dub06, HORM08, CKS09, ALY09], *query languages* [LM00, Cor06, Dub06], *image processing* [Kam98, EFDM03, SCK04], *user/web interfaces* [BV04, TFH09, KG07, Gar08], *code generation* [BV04, CM07, Tra08], *simulations* [Pes01, OSV07] and *testing* [FP06, AO10].

There are two distinct styles of embeddings, which have different qualities. According to Tratt [Tra08] language embeddings can be distinguished with respect to the relation to their host language: *homogeneous* and *heterogeneous* embeddings. A *homogeneous embedding* is an embedding in Hudak's sense [Hud96, Hud98], where a language developer implements a language basically as a library whereby the host language infrastructure compiles or executes homogeneously *embedded programs* and their embedded language libraries together with other programs in the host language in a uniform way. In contrast, a *heterogeneous embedding* is an embedding in Kamin's sense [Kam98]: a language developer uses the host language (also called *meta-language*) to implement a language as an embedded compiler. This embedded compiler pre-processes or generates code in a target language (also called *object language*). In heterogeneous embeddings, one can understand an embedded program as a specification for the embedded compiler that produces executable code for it in the target language. Although the idea to reuse features of the host languages in the two embedding styles is similar, the qualities of embeddings are fundamentally different and inherit different characteristics for the embedded language. Therefore, the homogeneous and heterogeneous embedding styles are used to categorize the embedding approaches in the remainder of this report.

2.1 Homogeneous Embedding Approaches

Homogeneous embedding approaches inherit most of their host language's features, since the embedding and its programs are seamlessly integrated with the host language. Thus, parts of the host language features to be used in programs of the embedded language. The literature proposes to use various programming languages with different features for homogeneously embedding languages, namely (a) *pure functional languages*, (b) *dynamic languages*, (c) *multi-stage languages*, and (d) *strongly-typed object-oriented languages*.

Functional Languages: In functional host languages, data types and (higher-order) functions are used to encode domain semantics. Several approaches have been proposed.

Hudak [Hud96, Hud98] proposes to use a *pure* functional host language for implementing embeddings, called *pure embedding*. A language developer defines domain types with algebraic data types and their domain operations with higher-order functions on these types. The major advantage when embedding a language in a functional language is that language developers can rely on the host language's support for functional and type-safe composition. To evolve languages, developers can compose languages from modularly implemented constructs using *monads* [Mog89, Wad90], if the used algebraic types are compatible. Hudak demonstrates that with his approach, one can implement simple domain abstractions for small languages Such as a mathematical language for calculations on regions. Further, one can implement common features found in mainstream programming languages, such as *state* and *error handling*. With Hudak's embedding, there are several disadvantages that DSL researchers have identified and that are common to almost all other homogeneous embedding approaches. Most important, the syntax of programs is often inappropriate [MHS05, KLP+08], since programs are encoded the host language in an abstract syntax. Another problem is that there is an interpretative overhead, when

executing embedded programs, since there is function application and pattern matching on the embedded library, which poses an *interpretative overhead* at runtime [She04a]. In [Hud98], Hudak proposes to exploit partial evaluation to remove part of the interpretative overhead, but he found that the quality of partial evaluation of embedded expressions depends on the functional language the embedding uses. Hudak proposal to embed language have launched an extensive body of ongoing research that targets at improving the composition of type between languages as well as at removing the interpretative overhead of embedding DSLs in functional languages.

Leijen et al. [LM99, LM00] apply Hudak's technique and use monads to homogeneously embed support for *SQL* [DD89] statements in Haskell. They call their embedded library *HaskellDB*. They map SQL queries to *list comprehensions* that provides syntax in functional language to build a new list from existing lists. Further, they map SQL expressions to unsafe algebraic data types, and they make them type-safe using *phantom types*. Finally, a generator rewrites the embedded SQL queries to SQL code in standard syntax that is then executed on the data base server. The advantages of embedding SQL into Haskell are that they can guarantee type-safe SQL queries, i.e. once the Haskell compiler has type checked those SQL queries, they cannot fail. In particular, this property prevents SQL statements to select unknown columns. Unfortunately, HaskellDB has an abstract syntax that due to list comprehensions is very different from the standard SQL syntax. After all, their technique can be seen as an inspiration for embedding other languages, but they do not focus on providing a general approach to embed new languages, as the scope of their discussion remains restricted to SQL, therefore, this report excludes their approach from subsequent comparisons.

Carette et al. [CKS07, CKS09] address several problems of pure embeddings. They use functional composition to build *typed, embedded DSLs*. In contrast to computations that are tagged with type constructors such as by Hudak, they homogeneously embed tagless code generator functions in the *OCaml* language and discuss the transferability of their results in other typed functional languages, such as Haskell and MetaOCaml. They encode embedded programs in *higher-order abstract syntax* (HOAS) [PE88], also known as *Church encoding* [Chu40]. HOAS encodes expressions as lambda expressions, which enables reusing the host languages binding mechanism for the embedding. Using the host binding frees the language developer from keeping track on environments when developing evaluators. Moreover, HOAS enables using functional composition for the compositionality of embedded expressions, whereby preserving types. They define syntax in an OCaml *module*, and semantics in its *module implementation* that implements the corresponding module signature. Further, they use *functors* to bind expressions in a program to their semantics. A key property in their approach is that embeddings are *well-typed* and implemented in a typed host language. The resulting type-preserving interpretations have the guarantee that programs execute without type failures. Another key property of their approach is that, due to the encoding, programs can *abstract over semantics*, i.e. they can use different evaluators to interpret one and the same program representation under various semantics. The main advantage of this approach is that they represent typed expressions in the embedded language as type expressions in the host languages, i.e. using the same types makes the type system uniform such that the host compiler can check type-safety. While Carette et al. need only simple features, other program encodings in other approaches need advanced type-system features. The problem is that the use of those advanced type-system features, such as *generalized abstract data types* (GADTs), *dependent types*, or *universal types* either have disadvantages w.r.t. complexity and possibilities for optimizations. With their HOAS encoding of programs, the compiler of the host language can perform much more optimizations, since HOAS does not hamper with partial evaluation, in contrast to other functional embedding approaches in which tagging often prevents partial evaluation.

Atkey et al. [ALY09] address the problem that it is awkward to analyze and manipulate embedded DSL expressions encoded in HOAS. To solve the problem, they perform *intensional analysis* enabled by *unembedding* of embedded expressions as *de Bruijn terms*—a special encoding that they implement using GADTs in Haskell. In previous work, Atkey proofed that HOAS encodings can be mapped with an isomorphism to de Bruijn encodings and back [Atk09]. The advantage of de Bruijn encoding over HOAS is

that language developers can implement analyses more conveniently. They demonstrate the applicability of their approach by presenting several small embeddings, such as untyped and typed lambda calculus, functions with limited pattern matching, Boolean values, numbers. They demonstrate a simple analysis that counts expression in a program, and transformations between HOAS and de Bruijn terms, but domain specific analysis and transformation is out of scope. Further, they show that their approach can enable *mobile code* and *nested relational calculus*, which permits nested queries in query languages. The downside of using GADTs is that embeddings may suffer from exhaustive pattern matching [CKS09]. Further, they identify several problems and limitations with the current Haskell type system, which cannot proof type soundness in certain situations that lead to so-called *exotic types*. They address some of these problems, e.g. with *type casts*, while other problems remain unsolved and the resulting limitations pose an additional overhead on language developers.

To recapitulate, functional languages allow embedding and independent composing DSLs using functional and monadic composition. However, so far, only the implementation of small languages have been demonstrated. Because of the complexity pure embeddings, there are little applications outside academic community for several reasons. First, the work on embedding DSL in functional languages yet has been too little compared to existing work on DSLs, which makes it hard to understand their benefits and drawbacks. Second, there is no support for concrete DSL syntax, which reduces the end user productivity to write DSL programs [KLP$^+$08]. In other words, these approaches trade the ease for the language developers to embedding a language over ease the language end users that want to write DSL programs in concrete syntax. Third, the understanding of DSLs is quite different from the common understanding of DSLs by the DSL community. Pure embedding mostly demonstrate re-implementation of general-purpose language constructs (e.g. lambda abstractions) that are already available in their host, from which it is hard draw conclusions about applicability for implementing industrial DSLs. Although this mismatch does not violate the liberal definition of DSLs in general, there are little convincing example DSLs that demonstrate Forth, most concepts cannot be reused in main-stream programming languages that are used in industry, which do not have the required features and which have side-effects. Fifth, for exploiting the advantages of the functional embedding approaches, a rather practicable limitation is that these approaches assume the developer to be a domain expert *as well as* an expert in functional languages having advanced type systems, monads, and higher-order functions. Their assumptions heavily restrict the pool of available people, since only few developers in industry have both skills. In sum, because of these reasons and combinations of thereof, so far, their applicability is rather limited.

Dynamic Languages: There is a long tradition to embed domain-specific languages in dynamic languages. In general, embedding a DSL in a dynamic language is easier for the language developer, because there are no or little restrictions by a type system. Because embedding use rather simple techniques, they are frequently implemented by end users and average-skilled programmers. The downside of this is that DSL are implemented rather in an ad-hoc manner and that the host language provide less guarantees for embedding and DSL programs.

Embedding domain-specific languages has been a well-known technique in languages of untyped functional languages, such as Scheme from the LISP family. In [Pes01], Peschanski refers to such an embedded language as a *jargon*. A jargon is implemented in Scheme using Scheme's macro system. Embedded programs are represented as S-expression in abstract syntax. To define the abstract syntax, the language developer uses a meta-language, which is itself implemented as a jargon—a meta-jargon, that uses Scheme macros to define syntax and semantics. To define an expression type, the developer defines a new macro of which the name defines a keyword and of which the parameters its subexpressions. To define semantics, there are macro implementations that produce Scheme code at runtime. To evolve jargons, hierarchical composition of jargons is supported by one jargon explicitly importing other jargons. Other forms of compositions are not discussed. The benefits of jargons is that they are simple because there is no implementation overhead due to type annotations or restrictions by a type system. A drawback is that program execution can lead to runtime errors.

In object-oriented scripting languages, embedded DSLs are frequently used to implement rather small, ad-hoc DSLs that are also very popular outside the academic world.

Ruby [Rub, TFH09] is a fully object-oriented scripting language that frequently uses embedding DSLs in frameworks. Ruby allows modularly defining embedded DSLs in classes. Embedded programs are Ruby scripts in abstract syntax. Language developers define the abstract syntax for expression types in a class's method signatures and the corresponding method implementations define the semantics. For example, Ruby uses a family of embedded DSLs in its popular *Ruby on Rails* Web framework. There are numerous demonstrations of practicable embedded DSLs. To evolve embedded languages in Ruby, Ruby can re-open the class definition of an embedded language implementation to add new abstract syntax and semantic at runtime. There is little research of composing independently developed languages in Ruby, although it support features for composition, such as mixins, and feature for invasive adaptations via reflection.

Achenbach et al. [AO10] present an embedding approach for embedding languages in Ruby that targets at implementing dynamic analyses using dynamic aspects. They use a *meta-aspect protocol* [DMB09] and special *scoping strategies* for aspects [Tan08, Tan09] to control the binding and activation of aspects for dynamic analyses. Further, they apply a special technique to intercept execution at the basic block level, which is similar to the concept of *sub-method reflection* [DDLM07], but has been developed independently. On top of these techniques they implement special abstractions for dynamic analysis. The advantage is that end users can easily embed dynamic analysis for debugging aspects, similar to [DMB09], and it enables explorative testing with non-deterministic input data. Unfortunately, it is not clear how good the approach scales w.r.t. to language evolution, since composable analyses and transformation are not addressed.

Groovy [Gro, KG07] is another fully object-oriented scripting language that uses similar features like Ruby for embedding DSLs. Embedded programs are Groovy scripts that have an abstract syntax. Groovy supports extensible EDSLs using so-called *builders*, but they support only hierarchical extensions. A Groovy *builder* must extend a certain standard library class and add methods to encode syntax and semantics. Composition of independently embedded DSLs is possible, when they are implemented as *categories* using Groovy's support for dynamic mixins. Still, resulting language compositions have little guarantees for correctness, when languages have interactions and conflicts.

The π language [KM09] is a special host language with special features to change the syntax and semantics at runtime. What is special is that π programs can have any syntax of a *context-free grammar* (CFG). The language developers defines DSL expression types as so-called *patterns*. Each pattern recognizes a piece of the concrete syntax and gives it a *meaning*—an interpretation in the π language. The π interpreter processes DSL program line by line. When encountering expressions in a line, there must be always exactly one matching pattern for an expression type. Pattern can be redefined and they are lexically scoped, thus π always uses the inner-most enclosing pattern definition to interpret an encountered expression. The benefit of using *pi* is that syntactic and semantic extensibility built into the host language, which makes it particularly natural to evolve embeddings with the provided host language features. Unfortunately, π exceptional language features do not allow adopting the approach to other host languages, and they also require the language to be executed with an interpreter.

Renggli et al. embedded DSLs into the Smalltalk [RGN10]. Their approach, called Helvetia, addresses the problem of providing support for concrete syntax and improving tool support. End users can encode DSL programs either in Smalltalk syntax or, if a concrete syntax was defined by the language developer, they can use DSL syntax. To embed a language without special syntax, the language developer defines a set of Smalltalk classes of which the methods defines expression types. To embed a language with a special syntax, the language developer implements a parser in Smalltalk using a parser combinator library. To define execution semantics for special syntax, the developer uses an embedded DSL to implement transformation rules on AST nodes. Later, after parsing a DSL program, its expressions in AST nodes are transformed to ordinary Smalltalk code and then compiled by the host compiler. The advantage of Helvetia is that it supports certain kinds of evolution. Developers can extend a language by attaching ad-

ditional parser components to an existing parser using the combinators. They can define several parsers that can be used in parallel. They can even define parsers that use special reflective features of Smalltalk that transform existing programs. Another benefit of choosing the Smalltalk platform is that Helvetia integrates with the Smalltalk tools that developers can extend for syntax highlighting of DSLs. Helvetia's homogeneous integration with Smalltalk allows the debugger to trace transformed code back to its textual representation in concrete DSL syntax. Unfortunately, because Helvetia relies on the exceptional features of Smalltalk, e.g. that a compiler component is accessible at runtime, the approach cannot be adopted for other host languages that do not provide these features.

In sum, the advantage of embedding in these host languages is that their dynamic features provide great flexibility. Unfortunately, embedding in these language is ad-hoc and rather a craft than a discipline. Another downside of embedding in dynamic languages is the interpretative overhead of indirections needed for their dynamicity and for realizing the flexibility of their features. Ruby is an interpreted language. Groovy compiles to *Java bytecode*, but the generated bytecode contains many indirections. Further, since there are little guarantee for DSL programs and composition, since the interpreter and compiler do not check types before runtime. Last but not least, the power and flexibility of dynamic scripting languages has not been systematically studies and compared with other embedding approaches.

(Multi-)Stage Languages: A (multi-)stage host language [SBP99, COST04] has a small set of language constructs for the constructing AST nodes, combining them, and generating executable code from ASTs, whereby often a static type system guarantees that all programs they generate are correct. In (multi)-stage host languages, developers can implement languages embeddings using meta-programming in a homogeneous way, i.e. programs that generate other program in the same language. Staging-based embedding approaches address the problem of the interpretative overhead for embedded languages that is removed by generating code. There are several embedding approaches that use different host languages. Czarnecki et al. [COST04] compare *MetaOCaml*, *TemplateHaskell*, and *template meta-programming* in *C++*. The difference between these host languages and the approaches are rather minor and not relevant for a first comparison.

(Multi-)stage languages provide special features for construction, combination, and execution of program expressions. For constructing ASTs, the (multi-)stage languages provide a *quotation operator* with which developers can embed expressions of the object language into the meta-language. For example, in *TemplateHaskell*, one can quote a Haskell expression in Oxford brackets `[|..|]` that *reifies* a corresponding AST representation of it. For combining expressions of different stages, often there is a special *anti-quotation operator* to escape inside a quoted expression. Finally, for execution, there is a *splicing operator* that *reflects* an AST back to code, i.e. it generates executable code. With splicing, staging allows compiling programs from the object language to the meta-language, hence there is no interpretative overhead.

The biggest advantage with staging is that there are no library calls to an embedded library, but the embedding generates code at compile-time [COST04, Tra08]. Another advantage of typed multi-staged embeddings is that the host's type system can guarantee that (more or less) all generated code is well-typed [COST04]. Further, the quoting mechanism eases to mix expressions in the meta-language and the object language, which makes it relatively simple for language developer to switch stages (or levels) in the interpreter.

However, there are several disadvantages w.r.t. the support for concrete syntax, both for language developers and end users. For developers, although staging facilitates access to the AST, adding specific AST nodes e.g. for DSL syntax is out of scope, and not addressed in most embedding approaches for (multi-)stage host languages, in particular for the language end user.

Seefried et al. [SCK04] address problems of both homogeneous and heterogeneous embedding approaches. For homogeneous staging-based embeddings, they address the problem that the language developer has to implement a *compiler front-end* for the embedded language, i.e. the AST nodes for the embedded languages (cf. [COST04, SBP99]). For heterogeneous embeddings, which are discussed below, they address the problem that the developer has to implement a new compiler back-end

(cf. [Kam98, EFDM03]). To address these problems, they propose that an embedded compiler should uses compile-time meta-programming, which *TemplateHaskell* facilitates, which they call *extensional meta-programming*. With meta-programming, they can implement optimizations in a more homogeneous way, such as *unboxing* arithmetic expressions, *aggressive inlining*, and *algebraic transformations*. To validate their approach, they have reimplemented Elliott et al.'s *Pan* language [EFDM03], which is homogeneously embedded, to their homogeneous embedding with meta-programming, and compare the performance of their implementation with and without optimizations. Their use a different platform to implement Elliott's *Pan*, and thus, it is not fully comparable, but their measurements show that still the original heterogeneous implementation of *Pan* outperforms their reimplementation.

Tratt [Tra08] is the only who proposed a single-stage embedding approach with support for concrete syntax in the *Converge* programming language. It is different from the other staging-based approaches in that the language developer describes the syntax of the embedded language in a BNF-like DSL, generates a parser from this, and specifies transformation rules to rewrite AST nodes to Converge code. A Converge program can use a quotation operator to embed DSL code in concrete syntax into a so-called *DSL block*, which will *reify* a corresponding AST representation that is then rewritten by the rewrite rules, which *reflects* the AST nodes to executable code. The advantage of Converge is that language end users can write the program in any concrete notation. Further, transformation happens at compile-time and therefore the execution of DSL code can be expected to be rather fast. Unfortunately, in Converge, there are no guarantees that the generated code is type-safe.

Typed Object-Oriented Languages: Embeddings are implemented in object-oriented host languages that allow modular and type-safe language embeddings.

Evans [Eva03] and Fowler [Fow05] propose to embed DSLs into main-stream programming languages used in industry, such as Java. In Java, language end users can encode embedded programs in abstract syntax as ordinary Java programs that call the API of an embedded library. This API is structured in a special way, which Evans and Fowler call a *fluent interface*. The classes of the library define expression types in the embedded language using Java constructs. Literals are encoded as *constants*, domain-specific operations are encoded with method calls. For creating complex expressions, method calls can be *chained* together, where the return parameter of a method in the fluent interface represents the syntactic category of the next possible expression. He proposed to refer to such an embedded DSL as an *internal DSL* since the embedded DSL is implemented as a library, which contrast it from *external DSLs* that are implemented with pre-processors or other external tools. The advantage of their approach is that no special language features are required from the host language for embeddings. The disadvantage is that for the language developer it is hard to design the abstract syntax close to the domain, mostly because the Java syntax and semantics are not flexible enough to omit type annotations and delimiters.

Freeman et al. [FP06] apply Fowler's technique to embed one particular DSL—*jMock*—a library to support *test-driven development* by facilitating the creation of *mock objects*. In particular, they discuss lessons learned from previous versions of embedded libraries for testing. Further, they describe challenges when embedding DSLs in languages with a rather large syntax, such as Java and C++, which are not so much prevalent in languages with a small syntax such as LISP, Haskell. They also discuss the need for *user extensions*, For example, to use jMock for testing in a particular application framework, the end user must tailor error reporting that is built into *jMock* DSL for this particular framework. Specifically, they found that they as language developers *"cannot hard-code [...] error reporting since [they] do not know how the framework will be extended by its users"*. Further, they demands that language end users need *"programming hooks to make any extensions they write indistinguishable from core features in error reporting"*. After all, their contribution is a valuable experience report, but they do not propose a general technique to embed arbitrary embedded languages, and therefore this approach will be excluded from subsequent comparisons.

Garcia [Gar08] addresses the problem to reduce the effort to implement an embedded DSL as a fluent interface using generative techniques. A language developer models the syntax of an embedded DSL as a tree-based model in Eclipse EMF [SBP+09], and a generator generates the Java code for a fluent

interface API, called *EMF2JDT*. The major advantage is that the generator takes over the tedious task to encode syntax in a fluent interface from the language developer. Developers can combine their generator with another generator for model constraints, this enables generating constraint checks for embedded expressions to be well-formed. Further, standard tools services, such as code completion and debugging can be reused. The disadvantage is that language evolution is more difficult. Once the embedded DSL is generated, in case there is language syntax evolves, a language developer must update the model and generate the whole language again.

Dubochet [Dub06] and Odersky et al. [OSV07] experimented with embedded DSLs in Scala [Sca] – a statically typed language that combines features of object-oriented and functional languages. Currently, extending embedded DSLs and composing independently developed embedded DSLs is not addressed by Dubochet or Odersky et al. Dubochet and Odersky et al. have rather focused on demonstrating small examples of embedded DSLs, but they do not provide a disciplined approach for embedding.

Hofer et al. [HORM08] also use Scala to embed DSL. To enable multiple interpretations of programs, they apply Carette's technique [CKS09] in the context of *Scala*, which they call *pluggable semantics*. DSL programs are encoded in abstract syntax. Syntax of a language is defined by method signatures. Semantic are defined in the method bodies of classes or traits. To evolve languages, they use traits to hierarchically extended existing languages, with new expression types and semantic types. They also address the composition of independently developed languages. While composition independent languages are discussed, they do not address composition of languages that have interactions in the syntax and semantics. Composition of semantics are based on monadic composition of computations. In [HO10], Hofer et al. have adopted the idea of [ALY09] to use different forms of encodings to allow developer to simpler express new analyses and transformations, but none of the encoding is both extensible and composable. Unfortunately, since they do not address implicit isomorphic conversion from one encoding to another like [ALY09], developers can no more freely choose the best encoding after they have committed to one particular encoding.

2.2 Heterogeneous Embedding Approaches

Heterogeneous embedding approaches are interesting since they try to address the weaknesses of homogeneous embedding approaches by being inspired from traditional non-embedding-based language implementation approaches. Heterogeneous embedding approaches can also be distinguished w.r.t. what kind of host language is used to implement the embedding. First, there is the *embedded compiler* approach that embeds a DSL compiler/generator into a general-purpose language that generates code in the same or another GPL. Second, there is are approaches that embed DSLs in *source translation languages* What is common for both classes is that often the host and the target language are different, therefore they do not allow reusing the host language features within the embedding—embedding can only use the target language features. Moreover, even if they generate code in the same language that implements the embedding, they do not have a uniform compile- and runtime between the host and the embedded language, therefore they cannot uniformly exchange objects between those host and embedded programs.

Embedded Compilers: In [Kam98], Kamin proposes to embed languages as *program-generating languages*, where a (meta)-program in one language generates a program in another language, which basically are embedded generators or compilers, for which DSL programs are actually specifications for generating programs in another language. In such an embedding, the embedded language and the host language are heterogeneous, they may have different syntax, the may even have different semantics, and both languages may be processed by different infrastructure (i.e. compiler or interpreter). In such an embedding, a program of the embedded language uses the host language to rewrite its expressions into a target language. Kamin uses *ML* as a host language and generates code in *C++*, which is the target language. To define new syntax, the language developer defines a new expression type as a new function in ML. To define semantics for an expression, the corresponding function generates and

returns a code fragment in from of a string. To transform a program into its executable form, the fragments of all programs expressions are concatenated and then compiled by the target language compiler. Kamin demonstrates that his technique can be used to implement various embedded program generators (i.e., embedded compilers), such as *FPIC* a small language for drawing pictures, a *parser generator* that is combined from smaller parsing components, i.e. *parser combinators*, and a parser generator for the LL(1) sub-class of context-free grammars. The advantage of Kamin's technique is that the execution is less bound to a specific target language, as the code of the generator can be changed to produce code in a different target language. The generated code in the target language, does not suffer from interpretative overhead like homogeneous approaches. Compared to homogeneous approaches that reuse the host compiler, a disadvantage is the large effort a language developer has to spent for implementing a complete compiler back-end—i.e. the embedded compiler/generator. Moreover, heterogeneous embedded languages cannot reuse the host language features in the generated code (e.g. the host compiler's optimizations like partial evaluation), but they have to re-implement them using the features of the target language. Further, the program syntax in the embedded language still has abstract syntax and error messages that the target language produces are even more incomprehensible than with homogeneous embeddings.

Elliot et al. [EFdM00, EFDM03] extend Kamin's technique by embedding an *optimizing compiler* in that compiles Haskell to Haskell and that uses *algebraic manipulation*, which substitutes expressions by more optimal but semantically equivalent expressions. They address the problem that homogeneous embedded DSLs suffer from interpretative overhead. They first tried to speed up homogeneous embedded DSLs by adding custom optimizations using user-defined rewrite rules, which special host compilers enable, such as the *Glasgow Haskell Compiler*[1]. But, they made the experience that they could not remove this interpretative overhead. When they combined multiple of such rewrite rules, but there were too complex interactions between the rewrite rules that could not be controlled. To solve the problems, they represent program expressions in abstract syntax as *algebraic types* and statically optimize expressions when these are constructed. For optimizing an expression, they use a *smart constructor* for this expression that pattern match on its sub-expressions to detect opportunities for optimizations, so that an optimized expression is created. They apply optimization techniques inspired from traditional non-embedding-based approaches, such as *constant folding*, *if-floating*, and *static expression type specific rewrites for domain-specific optimizations*. Finally, an embedded compiler rewrites the optimized expressions to the target language. They detected an efficiency problem with a first embedded compiler version that repetitively rewrites *common sub-expressions* in a program. To avoid repetitive rewrites, they perform *common sub-expressions elimination* (CSE) that identifies common sub-expressions in a program, shares them between the expressions, and rewrites them only once. They demonstrate their application of their technique by implementing the *Pan* language, a small language for image synthesis and manipulation. The major advantage is that the language developer can evolve an embedded compiler into an optimizing compiler with only a few changes made to its code. Further, they claim far better performance of programs and more efficient program generation due to CSE. Unfortunately, they do not proof this claim by evaluating the actual performance speed up with measurements.

Cuadrado et al. [CM07] use Ruby to embed DSLs that are generators for model-driven development. They call their approach *RubyTL*. DSL programs are models from which the embedded DSL, which is a Ruby class, generates e.g. Java code. The approach provides an easy way to implement an ad-hoc generator.

Source Transformation Languages: MetaBorg [BV04] is an embedding approach that uses a source transformation language that can rewrite a heterogeneous embedded DSL to any GPL. It is the most mature approach for heterogeneously embedding DSLs with a concrete syntax [Tra08]. DSL programs are encoded in arbitrary syntax. MetaBorg uses Stratego/XT as a host language, which is a DSL for defining syntax and AST transformations. To define syntax, a language developer uses Stratego to define

[1] The Glasgow Haskell Compiler Homepage: `http://www.haskell.org/ghc/`.

new expression types as lexical patterns, which can recognize expressions in a program, and which create an AST representation. To define semantics, the developer associates each expression type either with a constructor for an AST node, or with a piece of embedded target language code that defines corresponding semantics. Given such a syntax definition of the EDSL and the host language, MetaBorg generates for this definition a corresponding pre-processor that internally parses programs and rewrites their AST. In MetaBorg, every language definition is module that other models can extend by importing all syntax rewrite rules of a super-module. To transform a DSL program, MetaBorg parses it to an AST representation, applies AST transformations successively annotate AST nodes, and finally applies an AST transformation that generates code in the target language. They demonstrate their approach by implementing a DSL for creating Swing applications with a syntax that is more concrete than the plain Java syntax to create Swing interfaces. The generated pre-processor rewrite the DSL to equivalent Java code. Further they provide an embedding of Java in Java that allows generating Java programs, and XML in Java for generating XML documents. The advantage of using MetaBorg to implement DSLs are manifold, since MetaBorg is a very mature tool. Years of investments have equipped Stratego with many useful features, for defining lexical patterns, importing grammars, priorities, quoting and unquoting, rewriting strategies, generic traversals, and advanced disambiguation with disambiguation filters. These features allow defining modular and composable syntax and semantics. Composable syntax requires supporting a subclass of grammars that is closed under composition, such as the full class of *context-free grammars* (CFG) that is supported by MetaBorg that generates *scannerless GLR* parsers [Vis97a] and resolve syntactical ambiguation using *disambiguation filters* [vdBSVV02]. In MetaBorg, embeddings are independent of the target language. Unfortunately, the fact that MetaBorg is not integrated with the target language disallows safe embeddings, since the generated parsers and pre-processors output code that may contain errors, which are later on detected by the target language compiler and which are hard to trace back. Further, MetaBorg is inconvenient for incremental language evolution, since whenever a language definition changes, its complete infrastructure must be regenerated, which is disruptive.

TXL [CHHP91, Cor06] is an embedding approach that uses a source transformation language that is similar to Stratego in MetaBorg, however the TXL language provide a different set of features. To defines new concrete DSL syntax, a language developer specifies the grammar with BNF-like syntax rules (productions), which recognize expressions and create an AST representation. To define new semantics, the developer can encode a transformation rule that has parameters accepting data from the AST. TXL uses these rules to rewrite AST nodes to the target language. Furthermore, TXL allows defining *functions* that traverse the AST to extract information from it and which developer can call in their rewrite rule implementations. To transform a DSL program, TXL parses it into an AST representation and uses the transformation rule the rewrite it into an executable form in the target language. Cordy demonstrates the applicability of TXL by implementing several language, such as a heterogeneous embedding of SQL in the *Cobol* [oD65] programming language, and a little generator that transform XML to C++ code. The advantage of TXL is that its allows modular language definition with transformation rules. In particular, rules and functional abstractions are interesting, since in TXL the developer can precisely scope the rules, by building hierarchies of rules that have sub-rules, whereby rules can pass parameters to their sub-rules. In TXL grammars are implicitly free of ambiguities, since every production is prioritized by the order the productions in a grammar are defined. Unfortunately, TXL does not support a composable subset of CFGs.

3 Desirable Properties for Language Embeddings

This chapter discusses desirable properties for languages that are currently only adequately supported by traditional non-embedded language approaches. The proposed set of desirable properties either has been identified by related work or identified as open problems of language embedding approaches. The central question is *what desirable properties should be supported by a language embedding approach to become competitive to traditional language approaches*.

3.1 Extensibility

When there are new requirements for a language, i.e. the language evolves in time, a language developer needs to extend the language's implementation. To cope with changing requirements, language implementation approaches should support extensibility [Ste99]. In general, a language implementation approach is said to support extensibility if it allows the developers to extend their language implementations [Hud96, EH07a].

Various language implementation approaches facilitate language extensibility [Hud96, Par93, NCM03, EH07a]. In these approaches, a *base language*[1] is extended with new language features that form a so-called *extension* to the base [NCM03]. The most important benefit is that the language features of the base language must not be reimplemented in the extended language [vDKV00, MHS05].

There are dedicated features for language evolution, such as grammar inheritance [AMH90, KRV08, Par08], overriding of grammars rules and transformation rules [Cor06]. Although these mechanisms are inspired by existing extensibility mechanisms in GPLs, they have been specialized for language engineering.

When extending a language, we can distinguish two kinds of extensions w.r.t. what facet of the language is extended: extensions that add new keywords to the language's syntax, and extensions that do not add new keywords but that extend the language's semantics. Each kind of these extensions is discussed in the following.

3.1.1 Adding New Keywords

An important form of *language evolution* is adding new language keywords, or respectively language constructs, to an existing language [MHS05]. When looking at the evolution history of many languages, a language often starts with a very limited set of keywords. Later, more keywords are added resulting in new versions of the language. The new keywords help the language to cope with additional requirements of its programs. For such language evolutions, a language implementation approach is said to support *extensibility* if it supports incremental extensions of a language with new language constructs [KAR+93, Ste99, Vis08].

Supporting this *incremental* extensibility is particularly important for domain-specific languages, because they evolve more frequently than general-purpose languages [MHS05]. When new keywords are frequently added to a language, it is beneficial if the language implementation approach supports incremental extensibility [MHS05, Vis08].

To cope with continuous language evolutions, a language implementation approach should support incremental extensions of syntax and semantics. It is important that incremental extensions do not only support adding new keywords but also overriding existing ones. Only when there is an extension mechanism that enables language developers to precisely select what parts of existing language implementations they want to reuse, a language implementation approach can minimize the implementation efforts of the developers for extending languages.

[1] The term *base language* should not be confused with the term *host language*. A language extension relates to its base language, which is extended by the extension. In contrast, an embedded language relates to its host language, which hosts the library of the embedding.

3.1.2 Semantic Extensions

In contrast to adding new keywords, there is a need for semantic extensions that do not alter the syntax of a language, but only extend the language's semantics [KRB91, HORM08].

When extending a language's semantics, we can distinguish two classes of semantic extensions w.r.t. whether the existing semantics are preserved: *conservative* and *non-conservative* extensions. A *conservative semantic extension* does not change the meaning of existing language constructs in their base language. When the meaning of existing constructs is preserved, programs written in the base language still produce the same results when they are evaluated using the extended language. In other words, conservative extensions maintain backward compatibility. In contrast, *non-conservative extensions* can alter the meaning of existing constructs in their base language. Therefore, when programs written for the base language are evaluated using the non-conservative extension, the evaluation can produce different results. In that case, there is no *operational equivalence* [Fel90] between the extension and its base. This section discusses such conservatives and non-conservative extensions.

3.1.2.1 Conservative Semantic Extensions

To support backward compatibility for end user programs, it is desirable for a language implementation approach to support semantic extensions that do not change existing semantic invariants of their base language. For example, consider an executing a program of an interpreted language with an optimization version (e.g. compiled version) of the language.

When providing conservative semantic extensions, language developers must make sure that all semantic invariants are preserved. For example, when executing an optimized version of a a program, it must be ensured that the outcome of the programs is not changed according to the language's semantic invariants.

3.1.2.2 Semantic Adaptations

When language developers cannot anticipate all possible future requirements for a certain language implementation, the language implementation needs to be open for extensions in the user domain [KRB91]. This is in particular interesting in two special cases. First, the user's requirements for a language are not exactly known before delivering the language implementation to the user's domain. Second, if the requirements for a language are expected to change *late*, that means after the language implementation has been delivered, e.g. at runtime of a program. Since a language designer cannot foresee all possible end user requirements for such a language, its language implementation should be designed according to the *open implementation* principle [Kic96]. The open implementation principle allows semantically adapting a language's implementation in the user domain by exchanging parts of its implementation strategies.

To provide support for adapting languages, variability needs to be built into language implementations. We call this variability of languages in the user domain *late variability*. Having support for such late variability in a language has a similar motivation like having support for late variability in other software systems with changing requirements [vG00, VGBS01]. The customizability, enabled through late variability in languages, would enable a better reuse and extensibility of language implementations in different end user domains. Late variability enables user-specific extensions to be provided even at runtime.

3.2 Composability of Languages

When there are diverse requirements by groups of end users in different application domains, this often motivates having a specialized language for each domain. For a better maintainability as motivated in the previous section, it should be possible for specialized languages to evolve independently from each other in a hierarchy of extensions. However, since application domains of independent languages often

overlap, Language developers are requested to reconcile two or more specialized languages into one language [BV04, OSV07, Par08]. Unfortunately, incrementally extending one of the specialized languages with others languages is not adequate for composing them, since the result is that multiple languages share similar constructs, which leads to code duplication in the extension implementation [Par08].

To address the problems of hierarchical extensibility, several language implementation approaches have been proposed that support composability of languages [EVI05b, KL05, Cle07]. When languages have overlapping domains, the language developer can decompose languages into smaller reusable sublanguages, from which one can compose these sublanguages into new a language. We refer to such a composed language as a *composite language*, and we refer to the sublanguages as the *constituent languages*. When composing languages, the advantage is that language developers must develop each constituent language only once and that the implementations of constituent languages can be shared among several composite languages.

To elaborate on these issues, we discuss composing languages with and without interactions in the following.

3.2.1 Composing Languages without Interactions

At times problem domains overlap, i.e. the same language constructs are used in several languages for different domains. In such scenarios, it is desirable to reuse those constructs of their language implementations. To compose stand-alone languages, a language developer needs to compose their syntax and semantics.

3.2.2 Composing Languages with Interactions

When composing languages, the language developer has to compose the syntax and semantics of these languages in a correct way so that expressions have a well-defined meaning in the composed language. In case the languages' syntax or semantics are not orthogonal to each other, the languages cannot be composed straightforward. To compose interacting languages, their *syntactic* and *semantic interactions* need to be handled correctly, as elaborated in next two sections.

3.2.2.1 Syntactic Interactions

For composing the syntax of several languages, the syntax of each constituent language must be integrated into the composite language in a consistent way [SCD03, BV04, Cor06]. But when the syntax of one constituent language is incompatible with the syntax of another constituent language, there is a *syntactic conflict*, such as an ambiguity. Such conflicting languages cannot be composed straight ahead.

To support composing languages, it is desirable that an implementation approach can detect, resolve, and prevent syntactic conflicts in language compositions. In case of a conflict, e.g. when two languages define the same keyword, it is not clear which language implementation is responsible for evaluating the keyword. There need to be a mechanism that helps language developers to declare a resolution of such syntactic interactions.

3.2.2.2 Semantic Interactions

When composing sublanguages, their semantics need to be composed correctly, so that expressions in the composed language always have a well-defined meaning. When non-orthogonal semantics of sublanguages are composed, the evaluation of a language construct in one language may affect the evaluations of a language construct in another language. In is the case, we speak of a *semantic interaction* between the constituent languages. When composing non-orthogonal sublanguages, it is not trivial to create a composed language from existing implementations [KL05, KL07], since interactions between the constituent languages can be unintended. If an interaction is unintended this can lead to unintended composition semantics, in that case, we speak of a *semantic conflict*.

For composing languages with semantic interactions, language embedding approaches should allow intended interactions and prevent unintended ones.

3.3 Enabling Open Composition Mechanisms

For one particular composition scenario, often its constituent languages can be composed in a predefined way. To support the language developer to implement a language composition for such a scenario, a language implementation approach should support declarative composition of the languages [BV04, Cor06].

The following sections motivate language-composition mechanisms that help controlling syntactic interactions, in Section 3.3.1, and second, declaratively controlling semantics interactions, in Section 3.3.2.

3.3.1 Open Mechanisms for Handling Syntactic Interactions

Composition mechanisms that allow controlling syntactic interactions can be classified w.r.t. how conflicts are handled. There are composition mechanisms that enforce that composition must be conflict free, that are discussed in Section 3.3.1.1, and composition mechanisms that resolve conflicts in a certain way. In Section 3.3.1.2, we will discuss resolving syntactic conflicts by renaming conflicting keywords, and in Section 3.3.1.3, we will discuss resolving conflicts by prioritizing expression types of the languages in a composition.

3.3.1.1 Generic Mechanism for Conflict-Free Compositions

When independent languages are composed, a language implementation approach should prevent syntactic conflicts [BV04].

To support safe compositions of independent languages, there should be a composition mechanism that automatically detects syntactic conflicts and provides feedback to the language developer. The mechanism should report the conflicting keywords to the language developer. To allow language developers composing various languages in conflict-free compositions, the composition mechanism should be *generic*, i.e. the composition logic to detect conflicts should not be specific for particular languages.

For cases in which there are special requirements on a language composition, the generic composition mechanism should be open for extensions. When a language developer composes several languages, the developer may decide to restrict the lexical regions in which certain keywords can be used. E.g., the developer declares that it is forbidden to use a subset of the keywords in the body of an abstraction operator. If a program uses a lexically restricted keyword in a wrong lexical region, we refer to this as a *context-sensitive syntax conflicts*. To prevent context-sensitive syntax conflicts, the language developer must have the possibility to specialize generic composition logic for handle those conflicts by taking into account the keywords' contexts and the constituent languages.

3.3.1.2 Supporting Keyword Renaming

When languages with syntactic conflicts are composed, a language implementation approach can resolve such conflicts by adjusting parts of the syntax for a composition, e.g. by overriding one of the conflicting expression types [Cor06].

To support composing language with syntactic conflicts, there should be a composition mechanism that enables the language developer to declaratively resolve the interactions by changing the conflicting parts in the syntax. When adjusting parts of the syntax for a composition, one has to keep in mind that the composite language is not backward compatible to the old syntax. Therefore, likely existing programs are not compatible to the new composite language, until the keywords of these programs have been renamed to the new syntax.

3.3.1.3 Supporting Priority-Based Conflict Resolution

Often language extensions are implemented independently from each other, still the language developers can plan for composing extensions with the base. Composing several extensions to the same base language is easier than composing stand-alone languages, because the extensions can rely on the same base language theorems. When extensions have a common base, even when there are interactions, conflicts are less frequent. Therefore in such compositions, often is is sufficient to resolve conflicts using a priority.

When there are multiple extensions to a shared base language, a language implementation approach should support composing the implementations of those extensions [KL07, EH07a]. In contrast to composing stand-alone languages, interactions between extensions and their base language are easy to resolve. In conflict-free stand-alone languages, such common keywords are disallowed for a conflict-free composition, since keyword semantics would not be well-defined. In contrast, because the extensions have a common base, sharing the common keywords of the base language is not a conflict, since the interaction can be resolved.

A possible resolution to compose languages that have syntactic interactions is to prioritize the constituent languages and always use the keyword semantics of the language with the highest priority. Another approach is to declare priorities on the level of expression types.

3.3.2 Open Mechanisms for Handling Semantic Interactions

When semantics are non-orthogonal, it is not straight-forward for a language developer to compose constituent languages. Therefore, it is desirable to have composition mechanisms that support the developer in scenarios by providing common logic for handling particular kinds of semantic interactions.

3.3.2.1 Generic Mechanism for Crosscutting Composition of DSLs

Existing language embedding approaches focus on composition scenarios where the use of abstractions from one domain does not affect the evaluation of abstractions from another domain. We refer to such non-interacting compositions as *black-box compositions*, since they compose languages using black-box abstractions. The problem with black-box compositions is that, when multiple DSLs with crosscutting concerns are composed, programs exhibit scattering and tangling symptoms, as elaborated below.

The scattering and tangling symptoms are not restricted to one particular DSL, but many DSLs suffer from these problems. Example DSLs are *workflow languages* [CM04] (e.g., BPEL [AAB+07]), *query languages* [Alm] (e.g., SQL [DD89]), *grammar specification languages* [RMHP06, RMH+06] (e.g., BNF or SDF2 [Vis97b]), and languages for modeling *finite state machines* [Zha06]. Although scattering and tangling is a general problem in DSLs, surprisingly, there is little research on aspect-oriented programming for DSLs.

3.3.2.2 Supporting Composition Conflict Resolution

When composing multiple semantically interacting languages, there can be composition conflicts that are complicated to resolve [KL07, HBA08]. In general, since such composition conflicts must take into account the semantics of the application context, the system cannot resolve such conflicts automatically [Kni07].

For composing semantically interacting languages, a language implementation approach needs to detect such composition conflicts. Since the system cannot resolve composition conflicts automatically, the provided composition mechanisms should be open and configurable by end users i.e. the application developers.

3.4 Support for Concrete Syntax

Although abstraction mechanisms, such as function and objects, provide means for semantic abstractions, these mechanisms often fail to provide the right means for syntactic abstraction. Since most

language embedding approaches are restricted to comply with the concrete syntax of the host language, they often do not allow defining adequate syntactic abstractions [BV04, Tra08]. This missing support for arbitrary concrete syntax for language embeddings is one of the biggest obstacles for their adoption [KLP+08, MHS05].

There are numerous problems that hinder concrete syntax in language embeddings. First, many embedding approaches do not support automatically converting end user programs in concrete syntax into abstract syntax, but the end users are required to encode program expression in abstract syntax themselves. Second, an embedded language generally cannot define or refine a keyword that already is defined in the host language. Hence, embeddings cannot define new domain-specific semantics for existing syntax. Third, keywords that are special characters, such as brackets, operator symbols, and Unicode symbols, cannot be freely used in embedded expressions in most host languages. Often the special characters cannot be used in names. Finally, while certain host languages have support for overriding infix operators or defining new operators, *mixfix operators* [Mos80, DN09]—a mixed form of prefix, infix, and suffix—are mostly not supported by host languages.

3.4.1 Converting Concrete to Abstract Syntax

When the concrete syntax of a language is not directly supported in the host language, to still allow embedding expressions of an arbitrary language, an embedding approach should allow encoding the concrete syntax of this language in abstract syntax [Fow05].

According to [Kam98, ALSU07], the *abstract syntax* of a language consists of two sets: *abstract syntax types* and *abstract syntax operators*. The type set is defined as $T = \{\tau_1, ..., \tau_m\}$. The operator set Θ consists of operators $\Theta = \{\sigma_1, ..., \sigma_n\}$, whereby each operator σ_i is a mapping $\sigma_i : \tau_{i_1} \times ... \times \tau_{i_j} \rightarrow \tau_k$ with a unique signature.

To represent basic expressions, the language developer needs to support an abstract syntax encoding for each kind of expression. Each kind of expression $e : \tau_k$ begins with its operator followed by its subexpressions $e_1 : \tau_{i_1}$, ..., $e_j : \tau_{i_j}$ whose types match the operator's signature: $e = \sigma_i(e_1, ..., e_j)$.

To support abstract syntax for arbitrary embedded languages those concrete syntax is specified as a CFG, there must be two generic mappings. The first mapping needs to map concrete syntax of the embedding to abstract syntax. The second mapping needs to map abstract syntax to the host language syntax to make the abstract syntax executable.

3.4.2 Supporting Prefix, Infix, Suffix, and Mixfix Operators

Most host languages used for language embeddings have only restricted support for defining domain-specific operators [BV04, MHS05, Tra08]. Often it is only possible to override a subset of possible operators that are pre-defined in the host language.

It is desirable to support operators at every position, whether it is prefix, infix, suffix, or mixfix. An example for a prefix operator is "**not(**$pred$**)**" that negates a predicate function expression $pred$. An example infix operator is the circle operator that composes two functions "f **compose** g" to build a composite function h, where $f : X \rightarrow Y$, $g : Y \rightarrow Z$, $h : X \rightarrow Z$, such that $h(x) = f(g(x))$, $\forall x \in X$. An example for a suffix operator is "**(**k**)prime**", which can be used to refer to the derivative of a function k. *Mixfix operators* [Mos80, DN09] are combinations of the previous operator types as they have multiple operands whose positions are mixed with the keywords of the operator. Note that mixfix operators are of particular interest, because they are often used to define domain-specific abstraction operators. An example for such an operator is the SELECT-statement in SQL [DD89]:
"**SELECT** c **FROM** t **WHERE** p", where c is a selection of columns, t is a selection of tables, and p is a predicate clause on the rows of the tables.

3.4.3 Supporting Overriding Host Language Keywords

It is desirable to freely use arbitrary keywords, operators, and delimiters in DSLs [BdGV06, FC09]. In general, the host language's parser disallows using keywords in embedded DSL expressions that are also reserved keywords in the host language.

A problem is that most host language parsers disallows the reserved keywords to be used as identifiers. To still allow using such keywords in DSL expression types, it is required to transform DSL expressions that use those keywords to legal host language expressions. Consider an embedding with DSL expressions that use the reserved keywords of the host language. For example, overriding keywords "**if (...) then {...} else {...}**", when "**if**" is a reserved keyword in the host language

Another challenge in many host languages is that delimiter keywords (such as, brackets) cannot be overridden. Often defining new delimiters or Unicode delimiters is also disallowed by the host language parser. Consider embedding user-defined delimiters. For example, overriding curly brackets that delimit code blocks by the user-defined delimiters `begin` and `end`.

3.4.4 Supporting Partial Definition of Concrete Syntax

There are special embedding approaches [BV04, Tra08, KM09, RGN10] that support concrete syntax for DSLs by using *meta-programming*. They use meta-programming to create *meta-programs* that rewrite DSL programs in concrete syntax to host syntax. For this, the meta-program parses the program's expressions in concrete syntax, creates an AST from this parse, and rewrites the AST to equivalent expressions in the host language's syntax.

The problem with using meta-programming for embedding is that parsing requires the language developer to specify the complete syntax of the embeddings. Specifically, the developer has to specify the complete DSL syntax, the complete host language, and how expression types of the two languages are integrated. However, defining a full-fledged concrete syntax for a language using a formalisms can be very expensive [vdBvDK+96, Moo01, MHS05]. Providing a complete syntax definition for a large language is a tedious and error-prone task that can take several person months [vdBvDK+96, vdBSV97, Moo01]. The large costs for the concrete syntax are a competitive disadvantage of these embedding approaches compared to traditional embedding approaches [Hud96, HORM08] that do not need to specify concrete syntax.

In particular, for language embedding approaches, it is desirable to minimize the costs for concrete syntax. To better control the costs for defining the syntax, it would be interesting if language developers must not define the complete concrete syntax, but only for expression types where the concrete syntax is incompatible with the host syntax and that need to be rewritten. Further, it would be interesting if the developers do not have to define a complete concrete syntax at the beginning, but that they can add concrete syntax incrementally at the granularity of single expression types. When there is support partial syntax, then developers can start with implementing an embedding with abstract syntax. The developers can deliver a full-functional language prototype with abstract syntax in a shorter time, and they can add concrete syntax later on. The benefit is that end users can already use the prototype embedding with abstract syntax.

3.5 Enabling Pluggable Scoping

In programming languages, a *scope* is a region in a program in which variables (or another language constructs) are defined and bound to a value. A *scoping scheme* defines how such a *binding* of variables (or other constructs) propagate through the program.

When embedding a language, in traditional embedding approaches the host language controls the scoping of language constructs. However, there are several cases in which the language developer needs to control the scoping of the language constructs [LLMS00, Tan09].

3.5.1 Supporting Dynamic Scoping

Language embedding approaches inherit the semantics of their host languages. Thus, they inherit the scoping rules of their host to resolve identifiers in expressions, which is a fixed scoping strategy for most host languages.

The two major scoping schemes in programming languages are *lexical scoping* (also known as static scoping) and *dynamic scoping*. While host languages with lexical scoping allow lexical-scoped language embeddings, dynamic scoping is not available for them and vice versa. Often supporting lexical scoping is sufficient for an embedding approach, but this is only because most of today's programming languages use lexically scoping.

However, there is a need to freely select the scoping strategy for an embedded language. For example, there are languages that use dynamically-scoped variables in lexically-scoped functions [LLMS00]. When we want to embed such a language with special scoping requirements, host languages that only support one closed scoping scheme cannot be used for the embedding. For such situations, language developers need to embed scoping schemes into a host language that are different to its native scoping scheme.

When a language embedding requires dynamic scoping, normally language developers would select a dynamically scoped host language to embed it. Restricting the selection of possible host languages based on their scoping rules is problematic, since this restriction would disallow most host languages that are lexically-scoped to be used for the embedding. Further, when the embedded language needs both dynamically and lexically scoping, such as required in [LLMS00], host languages and embedding approaches that only support one closed scoping scheme can no longer be used. Still, it would be interesting to allow embedding dynamic scoping into host languages with lexical scoping.

3.5.2 Supporting Implicit References

There are languages that have *implicit references* [KM06] which are special references that are not user-defined but that are implicitly available in a certain context.

Frequently in general-purpose programming languages, there are special keywords such implicit references that point to values that depend on the context the keyword is used in. Most prominently in OO languages, there are implicit references that are used to refer to objects in the current lexical context. For example, Java defines the keywords `this` to refer to the enclosing object and `super` to refer to its super class. Similarly, aspect-oriented languages define implicit references, too, such as AspectJ [Asp] defines the keyword `thisJoinPoint` that can be used in advice to refer to the current join point. Those primitive identifiers are special in that the host language defines a closed scheme how to resolve them.

To enable an implicit reference, first, the language needs to establish a special binding for the implicit reference. Second, the language needs to resolve implicit references in a special way—different from explicitly defined identifiers, such as variables or functions. Therefore, it is desirable that language embedding approaches support embedding a special resolution scheme for implicit references. With this support, the language embedding could resolve those references via the lexical or the dynamic context the reference is used in.

3.5.3 Supporting Activation of Language Constructs

There are examples in which language explicitly want to control the activation [Tan09] of a certain language construct, such as dynamic extensions to classes and objects with *aspects* [PGA01] and *layers* [CH05]. With activation, one can explicitly control whether a certain language construct propagates or not, such as, the developer can declare that the extensions in an aspect or a layer are only effective in a certain region in the program.

In particular, activation is used in *dynamic aspect-oriented programming* [PGA01] to control the activation scope in which an aspect is effective. In dynamic AOP, a program can dynamically *deploy/undeploy* aspects to/from the running system, e.g., this allows activating or deactivating features those code crosscut several modules. There is good support for dynamic AOP for general-purpose programming

languages, such as Smalltalk [Hir01, Hir09] or Java [PGA01, Bon03, BHMO04], But, unfortunately, dynamic AOP is not available for most DSLs. Today, dynamic AOP support must be enabled by invasively changing the code of the DSL execution infrastructure [CM04]. What is needed is a systematic language implementation approach that helps developing a dynamic AOP solution for an arbitrary DSL.

3.6 Enabling Pluggable Analyses

Traditional language implementation approaches support analyses, such as syntax checks, detecting redundant expressions, domain analysis and so forth. When it is possible to inspect the code a program, this often called *intensional analysis*. Intensional analysis allows automatic reasoning on the program syntax and semantics. Normally, a parser makes a syntactic representation of a program available in form of an abstract syntax tree (AST). By traversing the AST nodes, these approaches allow performing syntactic analyses that analyze the syntactic structure of a program in the AST nodes. By storing information in AST nodes or by rewriting the AST nodes into another intermediary representation, it is possible to perform semantic analyses. For language with sophisticated syntax and semantics, traditional compiler and interpreter approaches allow combining multiple analyses.

To allow language embeddings in which analysis is needed, it would be desirable to support *syntactic* and *semantic analyses*.

3.6.1 Syntactic Analyses

DSLs need sophisticated syntactic analyses [MHS05]. *Syntactic analyses* enable reasoning over programs by analyzing their syntactic representation. Syntactic analyses are an added value for their language, since they support the end users to automate analyses that would be tedious for humans to check.

When programs are analyzed that mix the syntax of several languages, it is desirable to combine available analyses of their constituent languages instead of implementing a new analysis from scratch.

3.6.2 Semantic Analyses

Before executing a program, often it is interesting to determine important properties of the program that abstract over concrete executions of the program. For determining a property of a program, traditional language approaches use *semantic analyses* that evaluate a program under abstract semantics [MHS05, HORM08, Par10], instead of using the default execution semantics. Such an alternative program evaluation is interesting because it enables *abstract interpretation* [CC77, Cou96]. Abstract interpretation is a well-established technique that is available for general-purpose languages to check programs for certain semantic properties and to use the retrieved information to find errors in a program, to improve the program compilation or its evaluation. Yet, for most DSLs, no tools for abstract interpretation are available.

There are various opportunities for abstract interpretations of domain-specific programs. To identify possible bottlenecks in DSL programs, it is interesting to calculate runtime costs of DSL programs [WGM09], in particular, to support later optimization decisions. To help end users identifying obvious errors in their DSL programs, semantic analyses allow type checking. To detect more complicated semantic errors in DSL programs, semantic analyses allow checking of domain-specific constraints. A language embedding approaches should enable such analyses.

3.7 Enabling Pluggable Transformations

A *program transformation* is a process that allows constructing a program by successive applications of *transformation rules* that are mappings from one program representation to another [PS83]. A transformation allows rewriting a program into another language or into another version of the program written in the same language.

We can distinguish two types of transformations: *static* and *dynamic transformations*. *Static transformations* transform programs by taking only into account information from the static structure of a program. In contrast, *dynamic transformations* take also into account the runtime information, and thus they can be context-specific. In the following, we elaborate the need for such transformations.

3.7.1 Static Transformations

We can classify static transformations into two classes: *syntactic transformations* that transform only the syntactic representation of a program and *semantic transformations* that change the evaluation of a program by transforming it.

3.7.1.1 Syntactic Transformations

Often when language end users write down expressions, they find themselves in typing redundant information that is clear from the context. One solution is to define syntactic sugar that allow language end users to express their intents more concisely. When language developers do not want to include a syntactic abstraction into the language itself, there is the possibility to add support for syntactic sugar to the language by defining a transformation that *desugars* a syntactic abstraction.

In general, there can be two kinds of transformations, namely *exo-transformations* that have a different input and output language, and *endo-transformations* that have the same input and output language. For example, the above transformation is an exo-transformation, since a program with an extended syntax is transformed to a program with a limited syntax. It depends on the purpose of the transformation, what the right domain (source) and co-domain (target) of a transformation is.

3.7.1.2 Semantic Transformations

There are static transformations that do not only change the syntax, but they can also change the execution semantic of the expressions of a program. While this potentially may give a program a different meaning, in most cases, it is a desirable characteristic of a transformation that the transformation preserves *semantic invariants* for the program after the transformation. Such a semantic invariant is a part of the semantic contract of a language, and programs of the language assume all invariants to be always be true. In case all transformation rules guarantee that the transformed version of the program will still satisfy its initial specification, we speak of a *correctness-preserving transformation* [PS83].

3.7.2 Dynamic Transformations

A *dynamic transformation* allows transforming expressions in a program depending on their evaluation context. Specifically, a subset of their transformation rules dependent on the program context. When a transformation is context-dependent, its transformation rules are either only applicable in a certain context or the transformation itself is parameterized by the context.

In general, also for dynamic transformations, there can be *exo-transformations* and *endo-transformations*. What is in particular interesting with the above transformation is that it is an endo-transformation, because the domain and co-domain (source and target) are equal. Moreover the transformation must be *causally connected*. That means it must transform the running program, which is only possible if there is the runtime that is uniform and causally connected with the code that is processed by the transformation.

4 Review of the Support for the Desirable Properties in Related Work

This chapter gives an overview of support for the desirable properties for language evolution in related work. To review the desirable properties, this section gives evidence on how good a particular embedding approach copes with each identified desirable property. The review distinguishes homogeneous and heterogeneous embedding approaches because of their fundamental differences.

To classify the available support, the review gives four marks for the quality of the support: (N/A) when there is no support for the property, (◔) when there is a very limited support for the property, but the limitations prevent using the embedding approach in most cases, (◗) when there is a good support for the property, but there are more or less little restrictions in special situations that are mentioned, and (●) when there is excellent support for the property without important restrictions. In the following, for each property, this chapter discusses first how homogeneous embedding approaches support that property, second how heterogeneous embedding approaches do, and we summarize the complementary work from non-embedded traditional language implementation approaches.

After reviewing the available support by embedding approaches for each desirable property, the review draws a research roadmap for improving the support for that property, which is inspired by concepts from non-embedded approaches.

4.1 Extensibility

Recall that Mernik [MHS05] classifies a language approach to extensible, when the approach provides the developer with a special extension mechanism. In general, embedding approaches reuse the extensibility mechanisms of their host languages in order to extend embedded languages. Whereby this review gives evidence that those mechanisms have different qualities and how good they support language evolution. As it is argued for homogeneous embedding approaches in Section 4.1.1 as well as for heterogeneous embedding approaches in Section 4.1.2. Unfortunately, the available extensibility mechanisms are not adequate to deal with all forms of extensibility. Still, there is little research on how to extend existing mechanisms in the host language for better language evolution. Therefore, Section 4.1.3 outlines a road map that researchers can follow to improve the extensibility of embedding approaches. The road map discusses what mechanisms embedding approaches could borrow from non-embedded approaches.

The review evaluates the qualities of the extensibility mechanisms by related approaches. It reviews the available support for each identified kind of extensibility: (1) *adding keywords*, cf. Section 3.1.1, (2) *conservative extensions*, cf. Section 3.1.2.1, and (3) *late semantic extensions*, cf. Section 3.1.2.2.

4.1.1 Homogeneous Embedding Approaches

In general, most related homogeneous embedding approaches have a good support for extensibility.

Extensibility with Functional Host Languages: In general, pure embedding approaches with functional host language have excellent support for black-box extensions, namely higher-order functions and monadic composition (adding keywords: ●), whereby functional composition guarantees absence of side-effects between base and possible extensions (conservative extensions: ●). Functional abstractions are inherently inadequate for gray-box extensions that are required for late semantic adaptations, since the concept of a function prevents adaptation from the outside (late semantic extensions: N/A).

Hudak [Hud96, Hud98] discusses providing incremental extensions to pure embeddings as monads, such as adding state to pure embeddings, error handling, and optimizations. Hudak's embedding are type-safe and extensions respect the denotational semantics of their base, but because the computations of monads are tagged, an extended embedding can get stuck at compile-time, i.e. when extending the embedding, again and again, the developer has to define additional functions to handle new types until

all cases are treated, which makes extending embeddings safe but inconvenient compared to most other embedding approaches.

Carette et al. [CKS07, CKS09] extend embedding by adding new functions. Whereby, their advantage is that their embeddings are tagless, which allows them to derive functionality for host types. Further, they state that their technique has no negative implications for standard partial evaluation optimizations that the host compiler performs. They discuss how to extend an OCaml embedding with a modular compiler as a partial evaluator, whereby the approach still guarantees type-safety.

Atkey et al. [ALY09] discuss defining extensions using Haskell's type classes, parametric polymorphism, and GADTs. Their approach is type-safe, because they support conservative extensions. Although achieving type-safety is somewhat more complex than in the other pure embedding approaches, since the developers have to take care themselves of casts and exotic types.

Extensibility with Dynamic Languages: In general, dynamic host languages have excellent support for extensibility due to their dynamicity.

In Peschanski's *jargons* [Pes01], the language developer can extend jargons by importing into them another jargons. Still, jargon programs cannot abstract over their jargon's semantics (adding keywords: ◖). Unfortunately, conservative extensions are no addressed, since there is no indirection between macro calls and macro definitions. Furthermore, because he uses macros, there is the danger of unintended variable capture, since the macro he uses are not hygienic (conservative extensions: ◑). Although jargons have a meta-level—meta-jargons—the meta-level of a language is not accessible to language developers or users. Because the meta-level is not causally connected, semantic adaptations are not supported (late semantic extensions: N/A).

The embedding approaches based on dynamic scripting languages, such as ad-hoc embedding in Ruby [TFH09], TwisteR [AO10], ad-hoc embedding in Groovy [KG07] are similar. To extend embeddings, one can use single inheritance mechanisms, but they also support adding new members (i.e. methods and fields) to classes at runtime using their reflective features. In Ruby, since embedded DSLs are defined with classes, which are dynamically extensible, embedded DSL are dynamically extensible in the same way. Ruby allows to *re-open* class definitions and existing method can be renamed. One can dynamically add methods to a class that are defined in a *module*, with the restriction that Ruby does not allow modules defining additional state as fields that are added to classes, which is similar to disallowing state in traits and mixins of other languages. In Groovy, similarly methods can dynamically be added to EDSL classes, using *meta-objects* and *categories*. These features are in particular well-suited for unanticipated extensions, because there are several features that increase the flexibility for embeddings (adding keywords: ●). By omitting types annotations in the signatures, *duck typing* allows defining embedding with method signatures that can abstract over expression types. Still, executions are safe because Ruby and Groovy only defers type checking of dynamically-typed variables until runtime, however extension can potentially lead to runtime errors (conservative extensions: ◖). Moreover, the Ruby's reflective features and Groovy's MOP have good means for late semantic adaptations. Unfortunately, there is no disciplined approach to control semantic adaptations. The additional flexibility weakens possible guarantees that are provided for base languages and their extensions (late semantic extensions: ◖).

In π [KM09], it is a very natural approach to extend embeddings with new expressions simply by defining new patterns (adding keywords: ●). However, π does not support defining language as components, since there is no module mechanism to group expression types. Consequently, patterns and therefore extensions must be type-safe, but there can be runtime errors (conservative extensions: ◖). Late semantic adaptations are possible by redefining patterns, but π's scoping mechanism is only a little means to control adaptations (late semantic extensions: ◖).

Renggli's Helvetia [RGN10] has good support for extending languages. When no special syntax is used, Helvetia has the same extensibility properties as Ruby and Groovy. When extending special syntax, although it is convenient to construct new language from scratch by combining little parsers, it can be somewhat inconvenient to adds new keywords to an existing combination (adding keywords: ◖). Run-

time extensibility and abstracting over semantics is currently not targeted, it would require to revoke code that is already compiled. For extending semantic rewriting, multiple rule-sets can be active at the same time, but in case there are conflicting rules Helvetia will report an error. Unfortunately, there is no means to override rules or to resolve rule conflicts (conservative extensions: ◑). Helvetia does not consider late semantic adaptations were the user adapts the language implementations. But, Helvetia supports several individual semantic adaptations of user programs that rewriting programs as transformations. In Helvetia such transformations are provided by the language developer, no by the end user. Renggli demonstrates such semantic adaptation by instrumenting user programs to use *transactional memory* (late semantic extensions: ◑).

Extensibility with Staged Languages: Multi-stages approaches focus on extending transformation for optimizing the execution of embedded programs. Since extensible front-ends with syntactic (and semantics) extensibility are not such much in the focus, these approaches have a somewhat different focus of extensibility, namely extensibility of the back-end.

In multi-stage host languages embedding, developers must often explicitly define expression types, such as Sheard et al. [SBP99] do in MetaML and Czarnecki et al. [COST04] do in MetaOCaml, in TemplateHaskell, and in C++ templates. These languages do not adequately support extensibility, since the developer cannot easily extend the AST. A problem is that an existing data type, which encodes an AST node, cannot be incrementally extended (adding keywords: ◑). Their optimizing transformation are conservative extensions, and moreover they guarantee that generated code is type-safe (conservative extensions: ●). Semantic extension can be defined in new stages. Late semantic adaptation of one stage by the other is not in the focus, since reflective features are only used to reify the AST, manipulate it, and reflect it to the next stage. User extensions can only be provided as new stages, but manipulating existing stages is not in the focus. The use of *intensional analysis* to inspect code can be seen as an introspection mechanism for semantic extensions, but it is only supported by TemplateHaskell (late semantic extensions: ◑), in contrast, MetaOCaml (late semantic extensions: N/A) does not provide such as mechanism.

Seefried et al. [SCK04] have the advantage that they define DSL expressions as function calls and boxing and unboxing functions for conversions, which allows inheriting host language operations in their embedded functions. Similar to REA, they can add new expression types (keywords) simply by defining new functions, but abstracting over semantics is not addressed (adding keywords: ◖). Their optimizing transformations are conservative extensions, similarly to REA, and moreover their generated code is guaranteed type-safe (conservative extensions: ●). Late semantic adaptations are also not addressed (late semantic extensions: N/A).

Tratt [Tra08] is the only homogeneous approach base on meta-programming languages that supports extensible concrete syntax. Incrementally extending an existing language embedding is inconvenient, since the developer has to update the syntax definition, add new BNF productions, and regenerate the parser (adding keywords: ◑). Additionally, rewrite rules must be added. It is not clear how conservative extensions could be supported, because overriding rewrite rules is not possible (conservative extensions: N/A). Late semantic adaptations are also not addressed (late semantic extensions: N/A).

Extensibility with OO Languages: In general, embedded approaches using typed object-oriented languages have more or less excellent support for black-box extensions and gray-box extensions.

In theory, *fluent interface* by Evans [Eva03] and Fowler [Fow05] allow extensions, but extending an existing language with new keywords is somewhat complicated, because due to the special structure of a fluent interface API. Recall that fluent interfaces allow method chaining. Method chaining defines a language's syntax in several classes and that the return types of those methods. A disadvantage of method chaining for extensibility is that these methods are scattered over several classes, which makes it harder to find the right place where to add a new expression type. There is only a limited support for adding abstractions operators with inner classes (adding keywords: ◑). Because Garcia [Gar08] generates fluent interfaces, extensibility seems less problematic w.r.t. extensions, since one can simply re-generate the

code. But, as a developer must regenerate the whole API, this invalidates existing semantics the generated method bodies, which contain user-defined code from the previous version of the generated fluent interface (adding keywords: ◐). Evans and Fowler's fluent interfaces do not address conservative extensions (conservative extensions: N/A) In contrast, Garcia's EMF2JDT [Gar08] can generate checks from OCL-like constraints but without special guarantees (conservative extensions: ◖). None of the approaches address late semantic adaptations (late semantic extensions: N/A).

Dubochet [Dub06] and Odersky et al. [OSV07] embedded DSLs in Scala [Sca] that have good support for adding keywords but abstracting over semantics is not addressed (adding keywords: ◖). Conservative extensions are possible because Scala provides special features for type-safe extensions, such as traits, case classes, and dependent types. Safe conservative extensions are possible, but only if all pre- and post-conditions and invariants of language can be completely be specified on its types' interfaces. Since in the embedding approaches using Scala, a developer can only use Scala types for encoding the language syntax in the type's members, the type checker can only check if a conservative extension for possible syntax conflicts, but not for checking the domain's semantics. Rich semantic information cannot be expressed in Scala type interfaces. For semantic compositions, to prevent semantic conflicts by extension, Scala would require an additional design-by-contract solution (conservative extensions: ◖). But, they cannot support late semantic adaptations as there is no MOP in Scala (late semantic extensions: N/A).

The polymorphic embedding approach of Hofer et al. [HORM08, HO10] has equal or better qualities than the previous. In addition to the above techniques, Hofer et al. use Scala's support for *generics*, *path-dependent types* and imports for abstracting over the semantics for a program, which they call pluggable semantics. Basically, a program is implemented as a *generic class* that is parameterized by possible semantics for evaluating the program. For such a parameterized program, various semantics can be provided in a type-safe manner, whereby the return type of such a program is a path-dependent type (adding keywords: ●). They support similarly safe conservative extension as the other Scala approaches (conservative extensions: ◖). In addition, programs representation can be extended, as polymorphic embedding has support for multiple encoding of programs. But, polymorphic embedding does not address late semantic adaptations, because this would need to open up the Scala embedded types and to make adaptations to their interfaces that would violate their static guarantees (late semantic extensions: N/A).

4.1.2 Heterogeneous Embedding Approaches

Extensibility with Embedded Compilers: Heterogeneous embedding approaches are interesting since they try to address the weaknesses of homogeneous embedding approaches by being inspired from traditional non-embedding-based language implementation approaches.

Kamin's [Kam98] support implementing extensible generators, but programs that abstract over semantics are not in the focus (adding keywords: ◖). It does also not support conservative extensions (conservative extensions: N/A), and late semantic adaptations are not supported (late semantic extensions: N/A).

Elliot et al. [EFDM03] extend Kamin's technique, therefore they partially support also extensibility, but incremental extending is data types that represent AST nodes is not possible (adding keywords: ◖). However, the extensions they have implemented target at improving the execution time of the generated code, rather than targeting at syntax and semantics extensions. They do not discuss conservative extensions (conservative extensions: N/A) or late semantic adaptations (late semantic extensions: N/A).

Cuadrado et al. [CM07] apply Kamin's technique in dynamic languages to implement ad-hoc generators in Ruby. Their generators can be incrementally extensible, by adding new method to generator classes, but it is not possible to abstract over semantics of a program (adding keywords: ◖). Although they can generate OCL-constraints into embeddings, the Ruby interpreter does not type check the generators and generated code is also not checked (conservative extensions: N/A). Although they internally use reflection in Ruby to extend the generator, late semantic adaptation of generators is not addressed (late semantic extensions: N/A).

Extensibility with Source Transformation Languages: MetaBorg has an good support for modular and extensible languages with concrete syntax and modular semantics. It has limited support for abstracting over semantics of programs, but the developer has to reimplement the complete transformation implementations. Replacing the execution semantics for a program is only possible at pre-processing time. It is not possible to evaluate one instance of a program under different semantics, where the identity of the program representation is preserved (adding keywords: ◖). In MetaBorg, syntax extensions are supported, but no special type or encapsulation guarantees are given for languages and generated code (conservative extensions: ◗) and it does not address late semantic adaptations (late semantic extensions: N/A).

TXL has only a limited support for extensibility. To extend the syntax, the developer creates a new TXL module and imports another TXL module to extend, which makes all syntax rules from the other module available. In the extending module, it is possible to override imported syntax rules and add alternative productions to them. Abstracting over semantics has the same limitations as MetaBorg (adding keywords: ◖). In TXL, no special type or encapsulation guarantees are given for languages or generated code (conservative extensions: ◗). Unfortunately, TXL does not address late semantic adaptations by the user (late semantic extensions: N/A).

4.1.3 Roadmap: Extensibility in Non-Embedded Approaches

In contrast, embedded approaches use the extensibility mechanism of their host GPLs that are not specialized. Hence, it would be interesting to make specialized extensibility mechanisms available for embedded approaches, too.

Conversely, non-embedded approaches could learn from embedding approaches. In contrast to embedding approaches, these extensibility mechanisms are not implicitly available, user-defined abstractions to non-embedded mechanisms are not possible. The reason for this is that non-embedded approaches are mostly heterogeneous and not causally connected. Therefore, to cope with extensibility requirements, over the last decades, their meta-languages had to be invasively extended with extension mechanisms that were re-invented from GPL. First, grammar inheritance was adopted from OO languages [AMH90, KRV08, Par08]. Second, support for functions were adopted from functional languages [Cor06]. Third, recently aspect-oriented programming was made available to modularize crosscutting concerns in grammars [RMWG09]. Consequently, as mechanisms are frequently extended, it would be interesting if the mechanisms of non-embedded approaches would be extensible, like in embedding approaches [Mez97, HBA10].

In comparison, the major advantage of homogeneous embeddings over non-embedded approaches is that embedding are far less expensive in terms of implementation effort for the language developer. Another advantage of homogeneous embeddings is that they are homogeneous and causally connected, thus they can reuse advanced extensibility mechanism of the host. In contrast, the meta-languages of non-embedded approaches are not homogeneous and not causally connected with their target languages they generate code for. Therefore, for non-embedded approaches, it is not possible to easily reuse available mechanism in the meta-languages.

The remainder of this section summarizes interesting extensibility mechanisms in non-embedded approaches.

Adding Keywords: For a better support for adding keywords, embedded approaches could learn from *parser generators* and *compiler compiler*, as well as *extensible interpreters* and *extensible compilers*.

There is a myriad of parser generators with limited extensibility, such as *ANTLR* [Par93, Par08], and *Rats!* [Gri04, Gri06]. Similarly, there are numerous examples for compiler compilers that better support extensibility than parser generators, such as *SableCC* [GH98] and *JavaCC* [Kod04]. Often parser generators and compiler compilers use a grammar formalism and generative techniques to synthesize parts of the language front-end and back-end. A language developer can use a formalism to specify the syntax of a language, such as (E)BNF or SDF [HHKR89]. (E)BNF is a formalism that helps language developer

to reason about the syntax before implementing a new language, and this formalism is well-known to many language developers. The theoretical foundations of BNF and other formalisms help the developer specifying languages. Whereby, for a syntactic and semantic language composition, the developers can rely on the sound foundations and theories behind BNF, CFGs, and category theory.

The basic mechanisms for grammar specification are frequently augmented with special mechanisms for extensibility of languages. SDF [HHKR89, Vis97b] and ANTLR in version 3.1 [Par08] support importing other SDF modules to extend existing languages. MontiCore [KRV08] provides a declarative inheritance mechanism. What is common in these approaches is that they provide special mechanisms to override syntax rules from inherited languages, such as redefining and renaming keywords. Unfortunately, because in those approaches, the artifacts for syntax and semantic definition are coupled with each other, it is often hard to reuse both front-ends and back-ends for extensions.

Extensibility of embedded languages is in the same vein as extensibility of languages in extensible interpreter and compiler approaches. To address some of these issues, special techniques have been proposed. Often *attribute grammars* [Knu68, Paa95] are used to enables extensible syntax and semantics in compiler implementations. The literature discusses extensible languages in *grammar-oriented programming* or *syntax-directed languages*. A good overview of extensible languages is given in [KM09].

There are extensible compilers, that are supporting *multiple inheritance* enabling a better reuse, such as in *MontiCore* [KRV08], or ANTLR v3.1 [Par08]. Still, creating new extensible interpreters and compilers is expensive. Therefore, the investment is often only made for general-purpose languages, where extensions are more likely requested by a large group of end users. In contrast, in most cases, developers do not implement DSLs as extended compilers/interpreters, due to of the higher investment costs, and because often there is a relatively small group of end users for a DSL.

There are extensible interpreters that use parser combinators to implement extensible front-ends, e.g. in Haskell [Wad95, Fok95] or Scala [MPO08]. There are extensible interpreter back-ends, but they are hard to implement. Monads that have been discussed before, are often used for composable back-end implementations, also in non-embedded approaches. Examples for extensible interpreters are first-class interpreters [IJF92]. Other possible solutions are discussed in [SN05].

Extensible compilers of GPLs allow defining domain-specific extensions to GPLs. Examples of extensible compilers for Java are: *Polyglot* [NCM03], *JastAdd* [EH07b], or the *Java syntactic extender* [BP01], *LISA* [MLAZ00], or *Silver* [VWBH06, WKBS07, VWBGK08]. Furthermore, a *meta-object protocol* [KRB91] can also be seen as an extensible compiler that allows application-specific extensions [MHS05]. There are also extensible virtual machines that have just-in-time compilers that allows extensions, such as the JikesRVM [AFG+00, RG09] or the *virtual virtual machine* [FPS+02].

A particular interesting extensible compiler is JastAdd, since it is independent of the input language. JastAdd is based on *attribute grammars* to model syntax and semantics. There is support for base languages such as Java in *JastAdd* [EH07a], *Modelica* [ÅEH08], and a subset of *MatLab* [ADDH10a]. In JastAdd, *aspects* are used to build modular syntactic and semantic extensions to ASTs. JastAdd uses aspect-oriented features to introduce attributes into existing AST nodes. Aspects act as are a special means to implement an *attribute grammar*. JastAdd supports *synthesized attributes* that are attributes with values that are declared by equations (propagating information upwards the AST). It supports also *inherited attributes* that are attributes derived from a parent AST node to one of its children (propagating information downwards the AST). Extensions can access attributes of AST nodes, changed them, and add new attribute to AST node to store information. In particularly interesting is that JastAdd decomposes the access to attributes as small functional pieces of the language semantics that are automatically scheduled on demand. In JastAdd, language developers can relatively easily extend existing languages that are integrated as components. Interesting is that JastAdd supports gray-box extensions, where language extensions can adapt the output of their base languages invasively. Because of its gray-box extensibility, it is particularly well suited to implement special extensions to base languages.

Only a few extensible compilers are implemented for DSLs—mostly for DSLs with a large end user group, such as for *MatLab*. E.g., JastAdd has been used to modularly extend Java with an AspectJ [AET08] extension, and MatLab with aspects [ADDH10b].

Further, there are *commercial-of-the-shelf* approaches (such as XML [W3C06], UML [OMG04], EMF [SBP+09]) that provide good support for extensibility of DSL syntax, but there is a limited support for extending semantics. XML is targeted rather for structural DSL, than executable DSL. XML has easy support for syntax extensions through XML schema extensions [W3C06]. Semantics for XML can be provided when using XSL transformations for code generation [MHS05], but extending transformations is more complicated and requires special tools [ES01]. UML supports syntax and semantics extensions through UML profiles [OMG04], but to actually make a domain-specific model executable, a generator is used. Unfortunately, generators are implemented mostly in a monolithic way. They are heterogeneous because of using heterogeneous source-and-target transformations. They are not causally connected, because the source-model mostly does not have the same runtime as the target model, which is often executable code. For these reasons, generators are hard to extend.

Conservative Extensions: For better support for conservative extensions, embedded language could learn from non-embedded approaches that use formal syntax and semantics definitions, such as compilers generated from denotational semantics [JS80]. This is less a problem for pure functional embeddings and stage-based embeddings, which are type-safe. Nonetheless, since the language semantics in the other embedding approaches are rather implemented in an ad-hoc manner, integrating formal approaches would improve correctness of those approaches. After all, type-safety is only a minimal correctness guarantee when it comes to domain semantics, often domain-specific constraints cannot be encoded into the type system. Even when using a pure or staged embedding approach, there is no automatic guarantee that domain-specific constraints of a base language are met by extensions, since often such constraints cannot be encoded in the type-system of the host language. To address this, it would be interesting to use advanced type-systems that allow encoding application-level constraints on types. It could be interesting to use languages with *constraint types* or *refinement types* as host languages for embeddings, such as *Omega* [She04b].

Late Semantic Adaptations: To support late semantic adaptations, embedded language could learn from meta-level architectures and reflection in non-embedded approaches. There are language approaches that come with such a meta-level, such as extensible compiler and commercial-of-the-shelf (COTS) approaches.

There are extensible compiler approaches with a meta-level, such as *Reflex* [Tan04], or *Linglets* [Cle07], which have support for late adaptations. *Reflex* is a compile-time MOP for an object-oriented language, e.g. it allows implementing compile-time optimizations for dynamic aspects [Tan04], domain-specific aspect languages [TN05, FETD07], and compositions thereof [Tan06b]. *Linglets* [Cle07] uses a compile-time MOP to implement traversing strategies on AST nodes that are open for late adaptations, and non-local transformations. Extensible compilers with a meta-level allow late semantic adaptations of language implementations, but only at compile time.

Furthermore, there are homogeneous and uniform meta-levels available for general-purpose languages. Reflective programming languages allow programs written in general-purpose languages to reason about themselves [Mae87]. Meta-object protocols allow extensions of OO language semantics in the user domains [KRB91, Kic96]. Because reflective languages and MOPs are causally connected, their causally connected design is interesting from a language design perspective, unfortunately, they are not domain-specific.

When using reflection and meta-object protocols, an open problem is how to provide good safety guarantees for semantic adaptations. In particular, there is a conflict between providing static type-safety guarantees and reflection/MOPs. In general, it is hard to reconcile reflection with typing [Pie02]. When reflection and MOPs are allowed to manipulate the interface of objects at runtime, there is a

decidability problem to know what type an object has, in particular at compile-time, since the object interface may be changed in the future.

Specifically, type soundness is proven by means of *preservation* and *progress*. Preservation means that the type of an expression does not change during its evaluation. Progress means whenever an expression has a type, either it is a value, it can be evaluated further, or it raises one of the declared exceptions. In general, reflection conflicts with preservation, because the type of an expression can be adapted at any time [Pie02, ASSS09, GWTA10]. In other words, there can be no guaranteed type safety, when full reflection is allowed.

However, there are some concepts that try to address those issues by restricting reflection. For example, the issue is partially addressed by aspect-oriented programming, whereby according to Sullivan [Sul01] aspects can be understood as a MOP with a restricted flexibility. Since aspects allow adaptations only at certain points and using well-defined abstractions, there are additional opportunities for validation and also for optimizations.

Indeed most COTS-based approaches have a meta-level that allows user extensions. Most modern model-based approaches today come with a meta-level, which is heavily used to embed domain-specific models into general-purpose modeling languages or to extend the modeling notation with new means to express domain specific constraints. Examples are the *Meta-Object Facility* (MOF) that defines a meta-model for UML [OMG04], *Essential MOF* for the *Eclipse Modeling Framework* (EMF) [SBP+09], *XML Schema* for *XML* [W3C06]. Unfortunately, these meta-levels have only limited means to express semantics. UML supports constraints in its *Object Constraint Language* (OCL) and semantic transformations its *Query View Transform* (QVT) language. *Eclipse Model To Model* (M2M) a QVT-like language planned for EMF, XSLT supports semantics for XML. In academic community, there are similar modeling solutions and toolkits available, such as *MOFLon* [AKRS08] using graph transformation rules, *Generic Modeling Environment* [Dav03] that is especially interesting because it supports compositions of meta-models, and *Kermeta* [DFF+09] is a meta-programming environment for meta-model engineering. Although the above approaches have meta-levels, model-based approaches rather use their meta-levels for syntactic extensibility but rarely for semantic extensibility. Unfortunately, compared to homogeneous embeddings, the model-based approaches are often not causally connected, since in most of them, the models do not have a uniform compile- and runtime with the generated code.

Even if there is no flexible architecture available, there are still some general solutions to build software with built in variability. They allow building variation into language implementations. It is often possible to allow semantic adaptations with variability management tools for a language implementation, as for any other software. Variability management tools allow language developers to implement a language with extension points to which other language developers can provide extensions. On top of those extension points, developer can implement the variable features of the language. The resulting language implementation with the built in variability could be seen as a *software product-line* (SPL) [Bos00] of a language—*language product-line*, which can be configured for various domains.

Ideally, language developers organize variable features in a feature model, whereby they can define dependencies between those features. By checking the dependencies defined in the feature model, it is possible to automatically validate the correctness of a possible configuration. Finally, when customizing a new product of such a language product-line for a particular domain or an individual user, a domain expert who knows all available features configures the product-line by making a concrete selection on the desired features available from the product-line specification.

In theory every tool to build software product-lines could be used for building variable language implementations, however there are two major limitations: (1) *non-homogeneous*: to implement the product-line, the SPL approaches often use *meta-languages* or *meta-tools* that generate the product in another target language. Because their meta-languages are non-homogeneous, they disallow using the available tools of the target language when developing the product-line and its models, (2) *not causally connected*: SPL approaches often use generative techniques to generate products from the product-line

model, and thus the model and the products are disparate and do not have a uniform compile- and runtime.

Still, there are several relevant initiatives in the field of software product-lines that address these problems. First, a product-line developer can use special product-line approaches that have support for late variability [vG00, VGBS01]. By using such an approach to implement a language, late semantic adaptations could be possible. Although it has been shown that there is the benefit that late variability enables a better reuse [vG00], late variation has not yet been studied detailed enough for its application for adapting language implementations in user domains. Second, a product-line developer can implement software product-lines using special languages that provide advanced language features for implementing feature-oriented programming, such as *virtual classes* [GA07], *aspects* [MO04], and *layers* [CD08]. Moreover, multi-dimensional separation of concerns techniques have been used in the implementation of virtual machines [SHH09] to improve modularity of crosscutting features in VMs, such a garbage collection. Third, there is a new trend to maintain a part of the feature model and product-line models at runtime, such as in *dynamic software product-lines* [HHPS08, CD08, DMFM10], and *model at runtime* [IEE09, NB09]. It would be interesting to further investigate similar techniques, to enable better semantic adaptation in language embeddings.

4.2 Composability of Languages

There is an extensive body of ongoing research in the field of language composition. Embedded DSL are in particular interesting for language composition, because they have special properties that support composition. Similarly to extensibility, embedding approaches reuse the composability mechanisms of their host language in order to compose embedded languages Despite this the mechanisms have different qualities, as Section 4.2.1 argues that homogeneous embedding better supports semantic composition and heterogeneous embedding better support syntactic composition. Generally, there is good support for composing independent languages. Unfortunately, most existing embedding approaches fail to compose multiple languages that have complex syntactic and semantic interactions. Therefore, Section 4.1.3 outlines how to improve composability of embedding approaches inspired by non-embedded approaches.

In the following, for each approach, the review evaluates the available support for language composition. It reviews the available support for every identified composition scenario: (1) composition of languages *without interactions*, cf. Section 3.2.1, (2) with *syntactic interactions*, cf. Section 3.2.2.1, and (1) with *semantic interactions*, cf. Section 3.2.2.2.

4.2.1 Homogeneous Embedding Approaches

Not all homogeneous approaches discuss composing languages, but if, then generally there is good support for composability for independent languages. However, most embedding approaches have the limitation that compositions of dependent languages are not supported, and even if, there is only a limited support.

Composability with Functional Host Languages: In general, pure embedding approaches with functional host language have excellent support for black-box composition of multiple embeddings. Similarly to extensibility, higher-order functions and monadic composition help to compose languages from smaller pieces. An advantage is that functional decomposition guarantees the absence of side-effects between the composed languages.

Hudak [Hud96, Hud98] composes language from monads that strongly encapsulate each constituent language in a composition. In other words, Hudak uses a monad as a kind of language component (without interactions: ●). However, pure functional languages, such as Haskell, have a limited support to deal with syntactic and semantic interactions in embeddings. Restriction are mostly because, it is crucial for the functional paradigm to prevent interactions , which disallows invasive compositions of languages. E.g. in all approaches using Haskell, when composing two pure embedded languages, a user

needs to explicitly import functions and classes from different embedding into the same compilation unit using Haskell's sophisticated *import mechanism* A syntactic conflict occurs when two or more languages define a function with the same signature. The Haskell compiler detect such conflicts. To solve conflicts, when importing, the user need to explicitly qualify imported modules names and rename imported names form other compilation units. Unfortunately, explicit composition by users is less convenient for them who are not aware of conflicts and do not want to take care about resolving them. A problem is that Haskell uses the same dot (".") character to compose functions and also to *qualify* functions from imported modules. Therefore, when composing qualified functions, this can lead to syntactic ambiguities (syntactic interactions: ◖). Functional abstractions are inherently inadequate for invasive composition, since a function definition requires it to be a black-box. Due to this mismatch, pure embeddings do not support late semantic adaptations of functions by users (semantic interactions: N/A).

Carette et al. [CKS07, CKS09] do not discuss composition (without interactions, syntactic interactions, semantic interactions: N/A).

Atkey et al. [ALY09] use the power of the Haskell type system to implicitly compose functions from different embedded languages, or for convenience, Atkey allows explicitly composing languages by defining a new Haskell class that is a sub-class of two embedded languages (without interactions: ●). W.r.t. interactions, Atkey's unembedding approach has the same limitations as Hudak. (syntactic interactions: ◖, semantic interactions: N/A).

Composability with Dynamic languages: In general, dynamic host languages have exceptional support for composability, in particular, for unanticipated compositions. Because they can dynamically load and extend existing code, they are particularly flexible Further, because their dynamic type system can defer type checking until runtime, they can exploit that actual types are known, when the composition finally takes place. The downside is that dynamic host languages provide little static guarantees.

Peschanski [Pes01] does not discuss composition (without interactions, syntactic interactions, semantic interactions: N/A).

The embedding approaches that are based on dynamic scripting languages, such as Ruby [TFH09], TwisteR [AO10], and Groovy [KG07], allow ad-hoc compositions of languages that have been implemented independently and that may have ad-hoc interactions in their abstract syntax and semantics. For language composition, these languages use object inheritance and sometimes reflective features (without interactions: ●). Ruby can compose embedded languages by re-opening existing class definitions of embedded languages, and moreover it can dynamically mixin modules into classes. TwisteR can only compose aspect-oriented languages with the same restriction as its host language Ruby. Groovy can compose embedded languages using Groovy *categories* [KG07], or by adding methods to existing classes using meta-objects or dynamic mixins. Unfortunately, they do not provide adequate guarantees for ensuring correct compositions in case of syntactic or semantic interactions. In Ruby, TwisteR, and Groovy, syntactic conflicts are similar, they occur when two or more classes of embedded DSLs to be composed, or resp. modules or categories, define a member with the same signature. Ruby and TwisteR do not detect such conflicts. Although Groovy validates correctness of methods of inherited interfaces and classes, it does not of validate categories and mixins. When composing classes in Ruby and Groovy, internally when mixin composing the members of classes, these members are added to objects via their meta-objects, which holds a set of members that contains all members of the object it interprets. In Ruby and Groovy, whenever there is another adaptation of one of the members, it overrides previously defined members or adaptations, thus there is the danger of overriding conflicting expression types of embedded DSL classes that are encoded in the members. The problem is that overriding conflicting methods can happen unintended, whereby it remains unnoticed for the developer/user, and it can lead to incorrect compositions (syntactic interactions, semantic interactions: ◐).

The π language [KM09] can compose languages as a set of patterns, by composing their pattern definitions (without interactions: ●). To compose several languages, the language developer only needs to combine the code of the patterns and use the composed code together. For composing the syntax of languages, π allows composing π patterns that may have any concrete syntax of a CFG, which is only

possible because they use an scannerless Earley parser. When there are syntactic ambiguities, π resolves them by taking into account the return types of patterns, but a user defined resolution is out of scope (syntactic interactions: ◖). For composing semantics, there are limitations, while π allows black-box composition of patterns that have explicit interaction, i.e. a pattern can be parameterized by another pattern, which is a kind of *higher-order pattern*. Still, gray-box composition is not addressed, since in π higher-order patterns can only wrap around other patterns (semantic interactions: N/A). Further, in π, there is the limitation that there is no module concept for embedded languages. Since patterns are visible throughout their scope, there cannot be multiple modular languages in one program—there is only one language.

Renggli's Helvetia [RGN10] has good support for composing languages. For composing several languages in concrete syntax, in Helvetia, the developer combines the parser combinators of these several languages, which is easy when the languages have no interactions (without interactions: ●). Syntactic interaction are prevented, because the parser combinator library is based on *parsing expression grammars* (PEGs) [For04], which resolves ambiguities with implicit priorities. PEGs are a composable subset of *context-sensitive grammars*, albeit PEGs have no subset relation to CFGs (syntactic interactions: ◖). There is also support for semantic interactions, as embedded languages can change the semantics of other languages using Helvetia's rewrite patterns, that even can change Smalltalk semantics, such as the default method call dispatch and field access, which they demonstrate by implementing a DSL for transactional memory. Although semantics extension of the embedded languages and host language is possible, in Helvetia, there is the assumption that only one language may define a semantic interaction for a particular language construct or AST node. The problem is that Helvetia cannot compose interacting or conflicting rewrite rules of different languages (semantic interactions: ◖).

Composability with Staged Languages: None of the multi-stage approaches [COST04, SCK04, Tra08] currently addresses composition of multiple languages (without interactions, syntactic interactions, semantic interactions: N/A). With respect to syntactic interactions, the quoting mechanism can be only used to compose expressions of the host language and one embedded language. For the current research, it is not clear how the quoting mechanism can be used to reflect several ASTs of different languages at the same time In particular when ASTs are represented with GADTs, it is not clear how to compose ASTs, since without special mechanisms GADTs are closed. With respect to semantic interactions, the only semantic interactions that stage-based embedding address are interactions on the AST of the host language, which merely supports multiple optimizations on the same set of language constructs, but not of multiple languages.

Composability with OO Languages: In general, typed OO embedding approaches support composition of multiple embedded languages, but only well, when the host language provides an advanced inheritance mechanism, such as multiple inheritance. Still with the available mechanisms, there are limitations w.r.t. the support for language composition, since those mechanisms are not specialized for the language composition problems.

Evans [Eva03] and Fowler [Fow05] do not address composition of multiple languages. Garcia's EMF2JDT [Gar08] allows composition of an OCL-like constraint language with EMF models, but composition of other languages is not covered (without interactions, syntactic interactions, semantic interactions: N/A).

Dubochet [Dub06] and Odersky et al. [OSV07] discuss only the embedding of individual DSLs in Scala [Sca], but they do not discuss compositions (without interactions, syntactic interactions, semantic interactions: N/A).

In their polymorphic embedding approach, Hofer et al. [HORM08, HO10] demonstrate that developers can use Scala traits to compose independent languages. However, they explicitly exclude dependent languages from their discussion (without interactions: ●). Similarly, when embedding in Scala, a syntactic conflict occur when two or more types of embedded languages are extended or composes and if the types define members with the same signature. In particularly interesting is that users who know Scala well can easily compose expression from different languages and resolve conflicts. To prevent syntactic

conflicts, Scala *imports* allow qualifying, hiding, and renaming imported member names for possible compositions, similarly to Haskell. When importing several conflicting members for a composition, the Scala compiler automatically detect the conflicts and requires the user to resolve it. However, it is rather inconvenient that language end users have to resolve conflicts, it would be better if language developer could compose languages and resolve conflicts for them (syntactic interactions: ◖). Further, semantic interactions are not adequately handled, since the *linearization* of Scala's inheritance mechanism is not precise enough to compose interacting language compositions. Similar to mixins in dynamic language, only one semantics is taken into account of conflicting embedded types, namely the first of the conflicting members in the linearization, but once the order is established for a type, it cannot be changed anymore in subtypes (semantic interactions: ◖).

4.2.2 Heterogeneous Embedding Approaches

Heterogeneous approaches have different qualities in their support to compose languages.

Composability with Embedded Compilers: Kamin's [Kam98] does not address composition of multiple embedded languages (without interactions, syntactic interactions, semantic interactions: N/A).

Elliot et al. [EFDM03] support combining several optimizations of for the same embedded language, but composition of languages is out of the scope (without interactions, syntactic interactions, semantic interactions: N/A).

Cuadrado et al. [CM07] apply Kamin's technique in dynamic languages to implement ad-hoc generators in Ruby. Their generators can be incrementally extensible and they use multiple languages in parallel. However, compositions of languages are not in the focus (without interactions, syntactic interactions, semantic interactions: N/A).

Composability with Source Transformation Languages: *MetaBorg* has an exceptional support for composing the concrete syntax of multiple independent languages. This support is based on using a formalism for modular syntax definitions, namely SDF [Vis97b] as part of its Stratego language, and a scannerless GLR (SGLR) [Vis97a] parsing algorithm that supports composing arbitrary context-free grammars (CFGs). To compose languages, the language developer creates a new SDF module and imports all constituent languages (without interactions: ●). A syntactic conflict occurs, when two languages define the productions with the same lexical pattern (e.g. the same keywords). Nonetheless, MetaBorg makes it easy to resolve such conflicts, since it offers two mechanisms. First, either the developer handles conflicting productions from multiple languages explicitly at the syntax level, whereby the developer can rename the category names of imported conflicting productions. Second, the developer lets the parser simply recognize and parse the ambiguities, which result from the syntax conflict. Thanks to SGLR parsing MetaBorg obtains all possible resulting ASTs, which can then be filtered using so-called disambiguation filters [vdBSVV02]. These filters can take context information into account to resolve the ambiguities, by removing the invalid sub-trees from the AST (syntactic interactions: ●). MetaBorg allows writing rich semantics with complex rewrite rules that can be scoped. It is possible to apply a rule only in a certain phase of a transformation, to order rules, or to apply a rule only to some parts of a program. This allows developers implementing dynamic rules of which the rule application dependents on the AST context, or implementing *generic rewrite strategies*. Rewrites are intended to have side effects, at the cost that there is no guarantee that the rewrite rules are conflict-free. MetaBorg's rewrites are non-functional, therefore unfortunately transformations are *non-confluent*. However, there is no out-of-the-box support for complicated semantic compositions of languages in MetaBorg. Hence, for each composition, the composition semantics must be implemented from scratch (semantic interactions: ◖).

TXL has a good but limited support for composability. To compose several embedded languages, a language developer defines a new syntax module and imports the syntax modules of all constituent languages. Via the import, all syntax rules become implicitly available (without interactions: ●). Unfortunately, TXL has limits w.r.t. to its syntactic composability, because TXL does not support the full class of CFG. TXL implements a full-backtracking recursive descent parser that only supports the class *LL(*)*.

LL(*) is a subset of CFG. Unfortunately, LL(*) is not closed under composition [BV04]. In other words, in special situations, syntactic interactions can lead to erroneous parse trees, e.g. when the composed languages have same literals in their expression types (syntactic interactions: ◖). Composing language semantics is supported in TXL, due to the functional features and the fact that rules can easily be scoped. However, similarly to the latter approach, since there is no out-of-the-box support, composition semantics must be implemented from scratch (semantic interactions: ◖).

4.2.3 Roadmap: Composability in Non-Embedded Approaches

To improve the support for composability particularly of dependent languages, embedding approaches could learn from the available mechanisms in non-embedded approaches. In the following, we discuss inspiring composability mechanisms from compiler/interpreter approaches, extensible compiler/interpreter approaches, as well as, commercial-of-the-shelf approaches.

Compiler/Interpreter Approaches: Non-extensible compilers and interpreters are known to be hard to compose [MHS05]. Individually implemented compilers do not facilitate composition with other compilers.

In front-ends there is better support for composition. Various solutions that support partial compositions of front-ends of compiler and interpreters have been proposed. When *multiple inheritance* for grammars is supported, such as in [KRV08, Par08], this can be used to inherit from several grammars in order to compose them. One major challenge is to handle disambiguities, when composing expression types of different languages, e.g. by declaring *disambiguation rules* on grammar productions [vdBSVV02, Par07, KVW10] with priorities, associativity, (semantic) restrictions, rejections, or preferences. Alternatively, there are approaches that prevent disambiguities, e.g. by defining an implicit disambiguation by the order in which syntax rules have been defined [For04, Gri04, Gri06]. Despite the advances in the research in this field, most available parser generators and compiler compilers are not dimensioned for language composition. There are many problems, when language developers want to compose both syntax and semantics, as elaborated below.

First, it is not possible to generate code for constituent languages and then to compose the generated code. Therefore, a language developer needs to compose languages by composing their specifications.

Second, unfortunately, there are restrictions on syntax definitions in most meta-languages. Most parser generators are limited to a subclass of context-free grammars (CFGs). First, an examples class is LALR(1) that is supported by *YACC* [Joh75], *Flex/Bison* [LMB92], and *SableCC* [GH98]. Second, there is the class of LL(k) that is supported by *ANTLR* [Par93], *JavaCC* [Kod04]. What is problematic with those subclasses of CFGs is that they are not closed under composition [BV04, Gri06], when multiple grammars are combined. Consequently, LALR and LL parser generators are only dimensioned for single monolithic programming languages. In general, LALR and LL grammar specifications sometimes cannot be composed. Therefore, for language composition of CFGs, one need to use a composable subset of CFGs. There are several examples of such grammars. First, *regular expressions* is a composable subclass of CFGs, but this subclass is too limited for DSLs. Second, *parsing expression grammars* [Gri04, Gri06, VWBH06] are composable, but they are not a subclass of CFGs. Third, the *CYK algorithm* supports parsing CFGs, but they have to be given in *Chomsky normal form* (CNF) or transformed to CNF, which leads to a different AST when parsing. Most important, only *scannerless* parsers support the full class of CFGs without restrictions, such as *scannerless GLR* [Vis97a, BV04], GLL [JMS10], and *Earley parsers* [Ear68, Ear70].

Third, there are technical issues. For example, in *ANTLR* version 3.1, compositions are implemented with root parsers that import multiple parsers to combine their expression [Par08]. But a problem with the *ANTLR* solution is that the infrastructure of composed languages cannot be generated independently. When changing a root parser, it is not enough to re-generate only the root parser that imports constituent parsers in a composition, but for each combination of all constituent parsers, the complete infrastructure has to be re-generated, which makes is impossible to independently evolve the parsers. Another problem is that most parser generators leave it to the language developer to combine the languages' ASTs and

executing semantics. Because of these issues with current parser generators, it is not possible to generate a parser for a language, compile it, and share it with other language designers for composition.

Forth, compiler approaches have limited support for composing dependent languages. Most parser generators assume a specification that is free of syntactic and semantic conflicts, or they assume that possible conflicts are explicitly resolved by developers, such as *ANTLR*. Syntactic conflicts are prevented by using modular grammar definitions that support *namespaces* for expression types, such as in *SDF2* [Vis97b] and *Stratego/XT* [Vis04]. Semantics conflicts could be detected by taking into account formal specifications of the languages, such as TXL does. There is limited support to control dependencies between single language constructs. E.g., *ANTLR* allows controlling dependencies in name resolutions with semantic predicates. However, in most non-extensible compiler approaches, composition of whole dependent language components is not addressed. There is a lack of means to extend the internals of a language implementation for dependent compositions. Therefore, dependent compositions are often implemented as individual solutions [LK97, HH04] that are specialized to resolve composition conflicts.

In sum, with these issues, non-extensible compilers do not fully support composability of languages as components, but still it is the support for concrete syntax of CFGs that is inspiring for embedded languages.

Extensible Compilers/Interpreters: Developers can easily implement composable front-ends for compilers and interpreters with parser combinators, e.g. in Haskell [Wad95, Fok95] or Scala [MPO08]. Developers can compose a language front-end only by reusing existing the parser components from the constituent languages.

Most extensible compilers are implemented only for one GPL and do not allows composition of several languages. Extensible compilers/interpreters that support composability of languages are often based on attribute grammars, which also enable composability of attributes from different languages. There are extensible compilers that are specialized for composing a special subset of languages and a special subset of parts of the language implementations, such as *Reflex* that allows composing domain-specific aspect languages and handling weaving conflicts [Tan06b]. There are solutions that allow the language developers to declaratively model dependencies, such as *LISA* [MLAZ00]. The *AspectBench Compiler* [ATC+05] is based on *JastAdd* and allows composing AO extensions. Some solutions even allow to automatically schedule dependencies, such as the JastAdd Compiler [EH07b].

What is interesting with the JastAdd extensible compiler is that it especially targets compositions of languages and modular extensions. *Attribute grammars* [Knu68, Knu90] play a crucial role in JastAdd for enabling composition of several languages, by allowing composition by extending, i.e. with a common extension that imports all constituent languages. For a composition, a language developer defines a new module that imports and composes a set of composite languages. Each composite language is again module that other language developers can import and extend. In JastAdd, the composition of languages' attribute grammars is possible, because JastAdd has a general infrastructure for the language back-end. There can be several back-ends for different languages, and the developer can relatively easily integrate and compose them. To implement a new language syntax, JastAdd allows the developer to define a new AST and integrate it into the extensible compiler framework. To implement its semantics, JastAdd uses the declarative specifications of constituent languages' attributes, which can be declared as *lazy*, i.e. they are not immediately evaluated. To compose several languages, a developer can reuse the existing components of the constituent languages. First, the developer composes their syntax by explicitly defining the expression types of the composed language, whereby the composite language can reuse the syntactical categories of the constituent languages. Composing the semantics has special support, since JastAdd allows combining the declared attributes of different languages, whereby an exceptional feature of JastAdd is that the framework can automatically schedule the calculation of attributes that are defined for AST nodes, which is an important feature for a convenient and safe composition of multiple languages.

COTS-based Approaches: There are COTS-based approaches (such as XML [W3C06], UML [OMG04], EMF [SBP+09]) that provide good support for composability of DSL syntax, but they have limited support

for composing semantics. The COTS-based approaches come with structural representations that are syntactically homogeneous. For example, in XML, every XML syntax follows an XML Schema, which again is an XML document. For example, in MDSD, every UML model is an instance of a MOF meta-model, and respectively every EMF model is an instance of an eMOF meta-model. Because schemata, or models and meta-models are syntactically homogeneous, several instances of them can be easily composed.

In contrast for semantic composition, there are currently no complete homogeneous behavioral representations in COTS-based approaches, which are complete enough to make the structural representation executable. Currently, there is support for generating code from constraints expressed in *Object Constraint Language* (OCL). There is ongoing work to implement platform independent semantic transformations that are also composable, such as *Query View and Transformations* (QVT) for MOF/EMOF. But even if these rule-based language are composable, the COTS-based approaches usually compose models of several languages by transforming them. The transformed models are no longer homogeneous and causally connected.

When using XML for implementing DSLs, XML only has a good support for syntactic composition but not for composing execution semantics. The XML syntax of various DSL notations can be easily composed due to XML's generic syntax. When using XML for defining a DSL, its domain-specific primitives are defined as XML element types bound to a particular *namespace*[1]. A DSL program is an XML document that composes such XML elements. Namespaces help preventing syntactic conflicts when composing programs that use several DSLs. This is because, in a program that mixes XML elements from different DSLs, each domain-specific primitive always binds to only one well-defined XML namespace. In contrast, there is little support for composing semantics. Making a DSL executable is not in the focus of XML. Still, XML-based DSLs can be made executable by transforming DSL programs as XML documents with XSL transformations (XSLT) [W3C] into an executable form. When using XSLT to add execution semantics, however, composing XSL transformations is difficult and it can cause semantic conflicts. Component-based XML transformations [ES01, ELKP04] have been proposed that can compose XSL transformations, but composition conflicts have not been addressed in their body of work.

In comparison to embedded DSLs in COTS-based approaches, the host language plays a similar role to establish syntactic homogeneity as given in COTS-based approaches. Specifically, having a syntactic homogeneous representation of DSL expressions as host language expressions (in abstract syntax) is similar to having a syntactic homogeneous representation in form of XML Schemata or models in MOF and EMOF. However, an important advantage of embedded languages over COTS-based approaches is that programs are not required to be transformed to make them executable. Because the embedded programs are already semantically encoded through calls to the embedded library in the host language, their semantics are homogeneous and can be easily combined.

4.3 Enabling Open Composition Mechanisms

As a result of the current research in the field of language composition, special composition mechanisms were included into existing meta-languages, such as basic OO-like inheritance [AMH90, KRV08, Par08] and basic aspect-oriented features [HM03, RMWG09] for grammars. However, these basic extensibility and composability mechanisms for grammars have only limited support for what is possible by their ancestors.

Further, there are well known problems with existing OO and AO composition mechanisms. In particular OO inheritance mechanisms have shown to be inadequate to deal with special scenarios of object evolution, such as *name collisions* [Mez97]. As language evolution is more or less equally complex as object evolution, it is not enough to borrow existing mechanisms for grammars. Likely the problems of OO mechanism apply when adding those mechanisms to meta-languages. However, currently, there is little research to adopt the experience of dealing with OO and AO evolution problems in the context of

[1] W3C: Namespaces in XML 1.1 (Second Edition): `http://www.w3.org/TR/2006/REC-xml-names11-20060816/`

language composition. To design better mechanisms, research should take into account the conclusions of research results made with OO.

As a consequence for making better mechanisms available for language evolution, there are implications for the embedding styles. In case of a homogeneous embedding style, when using the available host language mechanisms to evolve embeddings, language developers should select a host language that does not suffer from evolution problems. For example, when using OO inheritance to evolve an embedding, e.g. an OO host language that supports resolution of name collisions. In case of a heterogeneous embedding style, meta-languages of parser generators, compiler compilers should be empowered with mechanisms that do not have those problems.

The review evaluates the quality of the composition mechanisms for each related approach. It checks whether there is a generic mechanism that is powerful enough to deal with all composition problems, or whether the mechanism can be extended for special compositions. The review validates the concrete support for the identified scenarios: (1) *conflict-free*: whether it detects conflicts and enforces conflict-free language compositions, cf. Section 3.3.1.1, page 20, (2) *renaming*: whether conflicting keywords can be renamed, cf. Section 3.3.1.2, page 20, (3) *linearization/priorities*: whether conflicting keywords can be disambiguated by using a partial order or them, cf. Section 3.3.1.3, page 21, (4) *crosscutting composition*: whether multiple DSLs can be composed, i.e. at least one DSL does semantically interact with another DSL, cf. Section 3.3.2.1, page 21, and (5) *composition conflicts*: whether, multiple semantically dependent DSLs can be composed, whereby resolving possible conflicts between them, cf. Section 3.3.2.2, page 21.

4.3.1 Homogeneous Embedding Approaches

Composition Mechanisms of Functional Host Languages: The pure embedding approaches of Hudak [Hud96, Hud98] and Atkey et al. [ALY09], thanks to functional composition, implicitly achieve conflict-free compositions (conflict-free: ◖), but the functional composition mechanism is closed and cannot be adapted e.g. to allow side-effects. To solve syntactic conflicts of homonymous functions, the import can rename conflicting functions (renaming: ◖), but the mechanism is closed, e.g. it is not possible to implicitly rename imported functions. In Haskell, it is not possible to abstract over two conflicting functions with the same signature. Because pattern matching is limited to one compilation unit, there is no meaningful way to compose the functions from two different compilation units (linearization/priorities: N/A). In the current pure embedding approaches, there are no mechanisms for invasive crosscutting compositions of functions, since this would require allowing invasively changing functions. Although there are special techniques [LHJ95] that allow changing functions that could be used, these techniques have been out of scope in current pure embedding approaches. In particular, for these techniques, it is not clear whether they do not violate the pureness of functional decomposition, which makes it unclear whether the techniques conflict with the pureness assumption made by pure embedding approaches [HO07] (crosscutting: N/A). There is no need to resolve composition conflicts, because side effect conflicts are not allowed and if the user wrongly composes functions or monads, the Haskell compiler will report a type error. However, with the approaches of Hudak and Atkey, there is also no mechanism that helps the user when the monad interpreter get stuck (composition-conflict resolution: N/A).

Carette's technique particularly guarantees that interpreters cannot get stuck. Still, this review does not classify the support for composition mechanisms in case of Carette et al. [CKS07, CKS09] since they do not discuss composition of languages, it is not clear whether composed interpreter will also not get stuck (conflict-free, syntactic interactions, linearization/priorities, crosscutting, composition-conflict resolution: N/A).

Composition Mechanisms of Dynamic languages: One exception is the work by Peschanski [Pes01] on jargons that does not discuss composition (conflict-free, syntactic interactions, linearization/priorities, crosscutting, composition-conflict resolution: N/A).

The other embedding approaches in dynamic scripting languages are all similar w.r.t. extensibility of their composition mechanisms. They use available host language extensibility mechanisms for composing embedded languages, but these mechanisms support composition only in an ad-hoc way, which can lead to incorrect compositions. Ruby [TFH09], TwisteR [AO10], and Groovy [KG07] are similar. All three approaches do not guarantee for compositions to be conflict free (conflict-free: N/A). Users can rename class members inside language embedding implementations, e.g. using the Ruby's *alias mechanism*, or using the special *expando meta-object* in Groovy that allows applying dynamic adaptation to any class (renaming: ◖). By default, when there are conflicting members in Ruby and Groovy, the last member defined (or added) is always the effective one. In Ruby, the effective member is always the one defined in the last *re-opening* of a class, or the last member definition that was dynamically mixed-in, or the last *alias*. In Groovy, the effective member is the last member that was last mixed-in via a *category* or *dynamic mixin* (linearization/priorities: ◖). Using always the last change that was made to a class is awkward, since normally multiple OO inheritance linearizes in the opposite order. Note that one can use reflection in Ruby and the MOP in Groovy to adapt some of the mechanisms effects, but this has not yet been addressed in the ad-hoc approaches. TwisteR uses reflection to adapt the semantics of aspects and composition conflicts between aspects, but not of other language constructs (crosscutting: ◖) Twister does not address resolving conflicts between aspects (composition-conflict resolution: N/A).

In π [KM09], there is no guarantee for compositions to be conflict free (conflict-free: N/A). Patterns can be renamed (renaming: ◖). Patterns have a lexical scope, thus π always uses the most enclosing pattern definition, which leads to a well-defined ordering but disallows user-defined priorities (linearization/priorities: ◖). Crosscutting composition for languages have been out of scope (crosscutting: N/A). It is an interesting question whether conflicting patterns can be composed by higher-order patterns, but using higher-order patterns to handle conflicts has been out of scope so far (composition-conflict resolution: N/A).

In Renggli's Helvetia [RGN10] when composing the syntax of multiple languages, developers can compose the constituent parsers as first-class objects, whereby one can compose the resulting combined parser with other parsers. Internally, the composition uses PEGs, and PEGs use the defined order to prioritize and compose expression types. However, the implicit ordering can be counter-productive, since it does not prevent unintended or incorrect ordering by the user, which remains unnoticed. Syntactic interactions are prevented, because the parser combinator library implicitly resolves ambiguities, whereby, the resolution is defined by the order the developer composes the parsers. Although PEGs are a composable subset of context-sensitive languages, albeit PEGs have no subset relation to CFGs (conflict-free: ◑). It is not clear whether there is a mechanism for renaming keywords that are used inside parser component (renaming: N/A). There is also support for semantic interactions, as embedded languages can change the semantics of other languages using Helvetia's rewrite patterns, that even can change Smalltalk semantics, such as the default method call dispatch and field access, which they demonstrate by implementing a DSL for transactional memory. Although semantics extensions of the embedded languages and host language is possible, in Helvetia, there is the assumption that only one language may define a semantic interaction for a particular language construct or AST node. When constructing a combined parser, to resolve conflicts, there is the possibility to reference syntax rules from other grammars, to define new rules from them, and to combine based on certain conditional (linearization/priorities: ●). In Helvetia, languages can be semantically invasively composed with the host language, but multiple sets of rewrite rules are possible, but they must be strictly independent (crosscutting: ◖). The problem is that Helvetia cannot compose interacting or conflicting transformation rules of different languages. When there are several transformation rules those patterns match the same condition, Helvetia raises an error (composition-conflict resolution: N/A).

Composition Mechanisms of Staged Languages: None of the existing multi-stage language embedding approaches [COST04, SCK04, Tra08] addresses composition of multiple languages and there is no means for extending composition mechanisms (conflict-free, syntactic interactions, linearization/priorities, crosscutting, composition-conflict resolution: N/A).

Composition Mechanisms of OO Languages: In general, embedded approaches using typed object-oriented languages use the closed composition mechanisms of OO type system, but in existing approaches the semantics of types are closed and cannot be changed. If the host language does not provide an adequate mechanism for composing several languages, there is no possibility that a language developer provides a new kind of composition mechanism or extends an existing one.

Fluent interfaces in Evans [Eva03] and Fowler [Fow05] do not address composition of multiple languages, and therefore, composition mechanisms are out of scope (conflict-free, syntactic interactions, linearization/priorities, crosscutting, composition-conflict resolution: N/A).

Dubochet [Dub06] and Odersky et al. [OSV07] do not address composition of embedded DSL in Scala (conflict-free, syntactic interactions, linearization/priorities, crosscutting, composition-conflict resolution: N/A).

In contrast, Hofer et al. [HORM08, HO10] polymorphic embedding discusses composition of independent languages and uses Scala type checker to guarantees type-safety (conflict-free: ●). Scala *imports* could be used to rename imported member names for possible compositions, but the user has to do it, not the language developer, which is awkward (renaming: ◖). Further, semantic interactions are not addressed, because Hofer explicitly excludes dependent compositions for which conflicting keywords must be composed (linearization/priorities: N/A). Compositions of semantically interacting languages, such as crosscutting composition are out of scope (crosscutting: N/A). Scala support only one fix linearization scheme to prevent conflicts. But the Scala host language features do not allow resolving complex conflicts (composition-conflict resolution: N/A).

4.3.2 Heterogeneous Embedding Approaches

Heterogeneous approaches have different qualities in their support to compose languages.

Composition Mechanisms of Embedded Compilers: Kamin's [Kam98], Elliot et al. [EFDM03], Cuadrado et al. [CM07] do not address composition of multiple embedded languages (conflict-free, syntactic interactions, linearization/priorities, crosscutting, composition-conflict resolution: N/A).

Composition Mechanisms of Source Transformation Languages: *MetaBorg* has good composition mechanisms to compose concrete syntax with and without conflicts. However, there is no guarantee for conflict-free composition of semantics (conflict-free: ◖). When there are conflicts between the syntax rules of two languages, the developer can rename the categories of imported syntax rules or define priorities to resolve conflicts. Further, there is the concept of disambiguation filters [vdBSVV02] that allows an excellent resolution of syntactic conflicts, therefore, there is no need to extend the disambiguation mechanisms (renaming, linearization/priorities: ●). For semantic composition, MetaBorg uses dynamic rules and traversal strategies for a controlled semantic composition, but there are limitations that come from its general architecture. Although MetaBorg can be combined with an external infrastructure to weave aspects, as Tanter tries by using MetaBorg to generate AO code for one individual general-purpose AO language [Tan06a]. With MetaBorg, it is not feasible to generate code that crosscuts multiple DSLs, because MetaBorg's architecture only supports integration with one target language, compiler, or runtime environment. Therefore, MetaBorg's current architecture is not adequate for crosscutting composing multiple DSLs (crosscutting: ◐). The developer can control composition conflicts of multiple rewrite rules by defining a dynamic condition for the application of the rules, but there are no mechanism to detect composition conflicts (composition-conflict resolution: ◐). In the end, in MetaBorg, all composition mechanisms are closed and not extensible.

TXL has good but limited composition mechanisms. When composing several embedded languages, TXL checks for syntax conflicts. Further, when composing the semantics of rewrite rules of different languages, TXL uses fixed-point compositional semantics. However, only the TXL functions are conflict-free not the rewrite rules (conflict-free: ◖). In TXL, one can rename keywords by overriding syntax rules (renaming: ◖). When there are conflicts, these can be resolved by overriding and reordering the syntax rules, to prevent the usual fix ordering of syntax rules. However, reordering by overriding is not

convenient for the developer (linearization/priorities: ◖). So far, crosscutting composition has been an non-issue in TXL (crosscutting: N/A), and even if, its architecture has the same limitation as the MetaBorg architecture. What is in particular interesting is that TXL supports building hierarchies of rewrite rules, sub-rules, and functions, and that it allows implicitly and explicitly scoping rules to control their applications, which are important mechanism to control conflicts (composition-conflict resolution: ◖). In the end, in TXL, all composition mechanisms are closed and not extensible.

4.3.3 Roadmap: Open Composition Mechanisms in Non-Embedded Approaches

Non-embedded approaches have good composition mechanism for independent languages. Often developers can use the available composition mechanisms without extensions. Nonetheless, existing composition mechanisms have shortcomings when using them for composing semantically dependent languages. While embedded languages can learn from existing dependent composition mechanisms, it is particularly interesting to study the limitations that non-embedded approaches for composing dependent language and languages that have crosscutting semantics.

Open Composition Mechanisms for DSLs: The COTS-based approaches to DSL implementation have been very active to find a solution for composing several dependent DSLs.

Model-driven engineering is a very active field of research, which focuses on visual languages. Visual languages are very different from textual executable languages that are in the focus of this report, still there are some interesting relations.

Model weaving [CH06] (also *model merging*) discusses compositions of several domain-specific models, but they focus on composing at the model-level, whereby the problem of composing the semantics is only moved into the generator for the woven model. However, composable generators are still an open problem that has been addressed only partially by QVT and other rule-based approaches. There are other approaches that are not base on rules, but allow complex dependent compositions, but they have practicable limitations, as elaborated below.

The *domain virtual machine* approach of Mélusine [EVI05a] discusses how to implement compositions of several dependent DSLs from scratch, but the composition has been defined for every new combination of domains. Reusable and generic compositions are not supported.

There are also aspect-oriented model weavers, such as XWeave [GV07, GV08]. But they do not support crosscutting composition of several DSLs. The problem is that they support only weaving of instances of the same meta-model. But, for crosscutting elements from the different domains, weaving is not supported.

Compiler/Interpreter Approaches: There are compiler and interpreter approaches that have generic or extensible composition mechanisms.

Heidenreich et al. [HJZ07, HHJZ09, HHJ+08] propose a generic weaver approach to compose programs of an individual language at the textual level, which they call *invasive software composition* [Aßm03]. They weave crosscutting concerns into the text representations of programs, but they do not crosscutting compose languages. Still, the approach is interesting because it supports weaving crosscutting concern into an arbitrary language. To define where to compose programs, language developers define hooks into the syntax of an individual language. At these hooks, programs can insert code fragments. What is special is that the weaver they use is a generic weaver for textual syntax. There are composition programs, that insert the defined code fragments at those hooks. Unfortunately, with invasive textual composition, it is only possible to weave aspects into programs of one particular DSL. While this supports invasive composition of programs, it does not address invasive composition of languages. When composing a program, after the pre-processor has produced woven code for it, its woven code is further processed by a traditional DSL compiler.

Wende et al. [WTZ10] propose to use role modeling at the meta-model level to invasively compose dependent languages using role-based interfaces. To define where languages can interact with each other, the language developers of a constituent language define a *role* for this language. A role is an explicit

language component interface that declares what each language provides and expects, which is defined in a *component specification language*. Further, there is a *composition language* in which a language developer can define a composition by describing how to compose the language components. For a language composition, the constituent languages need to have matching roles, which allows a composition tool to compose the modular component, w.r.t. the composition specification in the composition language. With such role-based language compositions, possible paths of evolution are anticipated at design time to allow a safe composition later on. Wende et al. applied their approach to re-implement OCL in a modular way, which allows them to define extensions to it. While developers can relative easily define extensions and compositions, they identified that the technique does not support more complex compositions of language extensions for OCL, which requires a more detailed specification than they could specify in their interfaces.

COTS-based Approaches: When using meta-models to define DSLs, their modeling notations (such as XML, UML or EMF) allow a relatively easy composition of syntax of domain-specific models through model weaving [EVI05b]. But semantic composition of several DSLs is more complicated. While COTS-based approaches provide extension points for syntax extensions (e.g. UML profiles) and tools for syntactic composition, there is a lack for semantic composition and tools for it.

To compose two models, first their meta-models must be composed. While it is rather easy to compose structural DSLs this way, it is rather hard to compose several executable DSLs, which are the focus in this report. Nonetheless, when designing a new architecture for language composition, one can learn from the problems that have been identified in the architectures of COTS-based approaches. In general, despite the fact that in model-driven approaches models and meta-models are syntactically homogeneous, it is hard to compose their semantics. A problem is that models often encode only details of the syntactic representation of their domains, but too little semantic information to allow semantic composition.

There are COTS-based composition approaches for models that have been proposed. Often, to compose several models, a meta-model is required that is shared between the models [GV07, EVI05b]. However in general each DSL has a different meta-model. There are only a few model-driven approaches that allow compositions of meta-models [Dav03, EVI05b]. Further, when composing models by weaving them, the composition logic must be often hand-written and there is little support for automatic composition [EVI05b]. Meta-model composition is discussed in [LNK+01, EI05, ES06]. [DSLB03] propose to use a joint action model and [SB05] proposes to define formalized interaction points between models to integrate them more easily. For a semantic composition of domain-specific models and their meta-models, the several generators would have to be combined. There is no general solution to this, except for special domains.

What is missing is a homogeneous approach that allows semantic compositions that are both homogeneous and causally connected—between its input and in its output. Unfortunately, COTS-based approaches often compose by transformation of one model into another model, but the input and output models are no more causally connected. But, in particular, homogeneous embedding approaches do not want to lose their advantage of being causally connected.

4.4 Support for Concrete Syntax

One of the biggest limitations in most embedding approaches is the missing support of concrete syntax in DSL programs [MHS05, KLP+08]. While concrete syntax is often a problem with homogeneous embedding approaches, only a few heterogeneous embedding approaches lack support for concrete syntax.

In the following, this review discusses whether there is support for (1) *concrete-to-abstract syntax*, cf. Section 3.4.1, (2) prefix, infix, suffix and mixfix operations—**-fix operations* for short—cf. Section 3.4.2, (3) *overriding host keywords*, cf. Section 3.4.3, (4) *partial syntax* for abstracting over concrete expression types in a language, cf. Section 3.4.4.

4.4.1 Homogeneous Embedding Approaches

Concrete Syntax in Functional Host Languages: All embedding approaches that use pure functional host languages [Hud98, CKS09, ALY09] have abstract syntax, none of them supports arbitrary concrete syntax (concrete-to-abstract syntax, overriding host keywords, partial syntax: N/A). Although functional languages, such as Haskell, allow defining prefix and infix operators, suffix and mixfix are generally not supported (*-fix operations: ◐).

Concrete Syntax in Dynamic languages: Peschanski's jargons [Pes01] does not support the scenarios (concrete-to-abstract syntax, overriding host keywords, partial syntax: N/A), except prefix operations are allowed (*-fix operations: ◐).

Ruby only supports only embedding with abstract syntax. Although Ruby is implemented as an AST-based interpreter, it is not possible for the developer to access the AST directly or to add new expression types to it (concrete-to-abstract syntax, overriding host keywords, partial syntax, *-fix operations: N/A).

TwisteR [AO10] does not have support for arbitrary concrete syntax, but interesting is that it addresses the problem of Ruby that embeddings cannot access the programs AST (in abstract syntax). TwisteR can convert concrete DSL programs in host language syntax to abstract syntax. For the conversion, TwisteR uses a fix pre-processor to convert Ruby syntax to abstract syntax in *S-expressions*. TwisteR needs this conversion to reflect on expressions at the basic-block level (such as if and loops expressions) in order to perform dynamic analysis. However, converting concrete DSL syntax to abstract syntax is out of scope (concrete-to-abstract syntax: ◐). The other cases of concrete syntax are not addressed (mixfix, overriding host keywords, partial syntax: N/A).

In Groovy, embeddings do not support concrete syntax (concrete-to-abstract syntax: N/A). Groovy supports overriding only a predefined set of operators, but defining new infix, suffix and mixfix operators in concrete syntax is not supported (*-fix operations: ◐). Still, Groovy supports *compile-time meta-programming* [Gro] intercepts compilation after parsing a Groovy file to rewrite its AST, before finally compiling it. The language developer can provide a custom AST visitor. At compilation time, such a visitor traverses the program AST and can rewrite its AST nodes, which allows overriding host languages keywords by other host languages expressions (overriding host keywords: ●). There is no support for partial syntax (partial syntax: N/A).

In π [KM09], patterns can recognize arbitrary concrete syntax of a context-free grammars. π supports abstract syntax, but it is not needed, since π can process concrete syntax directly (concrete-to-abstract syntax: ●). Therefore also special the cases are supported (mixfix, overriding host keywords: ●). Unfortunately π does not support defining partial syntax, when the π interpreter interprets a program it executes the program one line after the other, whereby every line can define new patterns with concrete syntax. What is in particular interesting is that π internally uses an Earley parser for CFGs that can be extended on demand. When defining a new pattern, this updates the grammars rules in the current parser, which will be taken into account when parsing the subsequent lines. This support incrementally extending the syntax of the π language in a program during its execution. But, π does not support defining partial syntax, every expression in a program must have a well-defined expression type of some defined pattern. π does not support abstracting over concrete expression types, and therefore does not support special parsing methods, such as *robust parsing* [Cor06] and *island grammars* [Moo01] or *union grammars* [Cor06] (partial syntax: ◐).

In Renggli's Helvetia [RGN10] parser combinators can parse any concrete DSL syntax of a parsing expression grammar (PEG). Helvetia intercepts the Smalltalk parser to convert a DSL program in concrete syntax, it parses its code, then it convert its AST representation in concrete syntax to abstract Smalltalk syntax, and finally, it let the default Smalltalk compiler continue to make the converted code executable (concrete-to-abstract syntax: ●). Therefore, all kind of operations and host language keyword can also be overridden (mixfix, overriding host keywords: ●). Defining new syntax with parser combinators is incremental, but it is an open question whether PEGs can be used for abstracting over expression types, e.g. for island grammars (partial syntax: ◐).

Concrete Syntax in Staged Languages: In the multi-stage language embedding approaches of Sheard et al. [SBP99], Czarnecki et al. [COST04], and Seefried et al. [SCK04] there is no support for arbitrary concrete syntax (concrete-to-abstract syntax, partial syntax: N/A). What is special it that because a stage can reify the AST at runtime—but only the AST of the host language. Further, the stage can manipulate the AST, by rewriting AST nodes to alternative AST nodes. Finally, the stage can reflect the manipulate AST the next stage. Such manipulation also allows overriding host languages keywords, but only with existing host expressions (overriding host keywords: ●). Although prefix and infix operators can be overridden, suffix and mixfix are generally not supported (*-fix operations: ◐).

Tratt [Tra08] is the only compile-time meta-programming approach that support languages that supports concrete syntax. To support concrete syntax, a language developer uses a BNF-like DSL to define concrete syntax. Then, the developer generates an Earley parser out of this syntax definition—with a full support of CFGs. In Converge, DSL code can be embedded as a string into quoted code blocks. For such a code block, Converge uses the generated parser to create its AST, which is rewritten with the rewrite rules that the developer has defined for the DSL semantics. Unfortunately, creating the abstract syntax from the concrete syntax is not a fully automatic process (concrete-to-abstract syntax: ◗). Because DSL block are quote, it is not problem to use host keywords inside the quotations (overriding host keywords: ●) Since Converge support the full class of CFGs, it support all kinds of mixfix operations (*-fix operations: ◐). Unfortunately, partial syntactic definition are out of scope (partial syntax: N/A).

Concrete Syntax in OO Languages: Evans [Eva03] and Fowler [Fow05] must comply with Java syntax and therefore do not adequately support the concrete syntax scenarios (concrete-to-abstract syntax, overriding host keywords, partial syntax: N/A), except in Java, one can define prefix operations as a method that uses Unicode in its method name (*-fix operations: ◐).

Dubochet [Dub06], Odersky et al. [OSV07], Hofer et al. [HORM08, HO10] do not support arbitrary concrete syntax (concrete-to-abstract syntax: N/A). Scala supports defining infix operations with Unicode syntax. To define mixfix operations, the developer can chain method calls on *case classes*, that would be normally qualified using the dot character (“.”), but it is possible to omit the dot for the calls, if it is unambiguous. However, custom suffix operations are not supported (*-fix operations: ◗). The embedding approaches in Scala do not support overriding Scala’s keywords (overriding host keywords: N/A). It is not possible to define partial concrete syntax, as the Scala parser cannot abstract over expressions (partial syntax: N/A).

4.4.2 Heterogeneous Embedding Approaches

Heterogeneous approaches have good support for concrete syntax, but there are different qualities.

Concrete Syntax in Embedded Compilers:

Kamin’s [Kam98] and Elliot et al. [EFDM03] uses ML’s/Haskell functional abstract syntax, but they do not address concrete syntax (concrete-to-abstract syntax, overriding host keywords, partial syntax: N/A). One exception is that infix operations are supported (*-fix operations: ◐).

Cuadrado et al. [CM07] also does not address concrete syntax, but since it uses Ruby, it has the same qualities as the ad-hoc embedding approach in Ruby (concrete-to-abstract syntax, overriding host keywords, partial syntax: N/A), as only infix operations are supported (*-fix operations: ◐).

Concrete Syntax in Source Transformation Languages: *MetaBorg* has support arbitrary context-free grammars with a SGLR parser. Ambiguities can be resolved using disambiguation filters (concrete-to-abstract syntax: ●). Because CFGs are supported, arbitrary prefix, infix, suffix and mixfix operations are supported (*-fix operations: ●). In MetaBorg, a DSL program cannot change the Stratego keywords (overriding host keywords: N/A). Although the Stratego language would support abstracting over the expression types, currently, MetaBorg does not address partial syntax for embedded DSLs, to embed a language, always the developer has to define the complete syntax of the embedded language (partial syntax: N/A).

TXL has a good but limited support for concrete syntax, it only support LL(*) but not full CFGs (concrete-to-abstract syntax: ◖). Except that limitation, TXL has the same qualities as MetaBorg (*-fix operations: ●). In TXL, a DSL program cannot override the TXL keywords (overriding host keywords: N/A). Although the TXL language would have support abstracting over concrete expression types, currently, this has no been discussed in the context of embedded DSLs and composing them (partial syntax: N/A).

4.4.3 Roadmap: Concrete Syntax in Non-Embedded Approaches

There are many different compiler and interpreter approaches that in general have excellent support for concrete syntax from which in particular homogeneous embedding can learn from. They mostly have comparable qualities for a concrete notation for an individual language. But, in many parser generator frameworks or compiler compilers, there are certain restrictions for language composition. A problem is that most tools support only a subclass of CFGs. Support for the full class of CFGs is crucial for composability and not for concrete syntax, because most subset of CFGs are not closed under composition [BV04].

With respect to overriding the special operations, such as mixfix, there are some challenges for traditional language approaches [Mos80, DN09]. This is because, mixfix requires full support of CFGs, using mixfix operation expression often lead to ambiguous parses that need to be disambiguated.

With respect to overriding the host keywords, there is generally no problem with compilers and interpreters, since most approaches have a front-end/back-end architecture, in which the input and the target languages are heterogeneous, and all keywords in programs are rewritten to the target language anyway.

With respect to partial syntax, there are only a few approaches that support abstracting over expression types that are not fully specified. There are special parser generators that support *island grammars* [Moo01] for partial parsing, an overview is given in [Lat03].

4.5 Support for Pluggable Scoping

In most of the embedded DSL approaches, support for pluggable scoping is not addressed. Regardless of the fact that whether embedded DSLs are homogeneous or heterogeneous, it is common that an embedding reuses the binding and scoping mechanism of its host language. This review explain the challenges with implementing and discusses are exceptions where scoping mechanism are defined on top of the host language.

To review the support for pluggable scoping, this section discusses whether (1) the embedding approach addresses to implement *dynamic scoping* for an embedded language, in particular implementing dynamic scoping in a lexically scoped host languages cf. Section 3.5.1, page 24, (2) whether it addresses to implement *implicit references*, such as `thisTurtle` cf. Section 3.5.2, page 24, and (3) whether it addresses to scope the *activation of language constructs*, such as dynamic aspects cf. Section 3.5.3, page 24.

4.5.1 Homogeneous Embedding Approaches

Pluggable Scoping in Functional Host Languages: All pure embedding approaches [Hud98, CKS09, ALY09] use the scoping mechanism from their functional host languages. All approaches use Haskell, which is statically/lexically scoped. Unfortunately, the developer cannot change the scoping of the Haskell, which is needed to allow a different scoping for the embedding (dynamic-scoping: N/A).

In Haskell, it is not possible to define implicit references as functions, because functions by definition cannot dependent on the dynamic context. A function cannot be used to implement an implicit references, because when resolving an implicit references in a different context, the return value would have to change, which is not allowed for a parameterless function (implicit references: N/A).

Current pure embedding approaches do not address the activation of embedded language constructs (activation: N/A).

Pluggable Scoping in Dynamic languages: Peschanski's jargons [Pes01] use Scheme that provides lexical scoping only (dynamic-scoping: N/A). Implicit references are not discussed and activation of language constructs (implicit references, activation: N/A)

Currently, in Ruby and Groovy, the ad-hoc embedding approaches do not exploit the flexibility of their host languages for scoping strategies (dynamic-scoping: ◖) and implicit references (implicit references: N/A). Activation of embedded language constructs is out of scope (activation: N/A).

TwisteR [AO10] only uses Ruby lexical scoping and does not support dynamic scoping (dynamic-scoping, implicit references: N/A). By combining Tanter's scoping strategies [Tan08] with the concept of the *meta-aspect protocol* [DMB09]. By this combination, they can realize scoping strategies for aspects (activation: ●).

In π [KM09] used to have support for dynamic scoping, but now uses lexical scoping. Currently, π has a closed scoping strategy does not support embedding different generic scoping strategies. However, Knoell et al. uses π for their *Pegasus framework* [KM06] in which they provide special scoping mechanisms for linguistics (dynamic-scoping: ◖). Therefore also special the cases are supported (implicit references: ●). In π, the program can always activate a new language construct that becomes active in its dynamic context, until it is undefined, which is a simple but effective activation mechanism, but dynamic activation is a limited solution in that it does not allow safe scoping of the construct (activation: ◖).

In Renggli's Helvetia [RGN10] does not address scoping strategies for embedded DSLs (dynamic-scoping: N/A). Their language boxes have support to lexically scope expression types, so that the rewrite patterns of a language component do affect only a certain scope, but rewriting expression does not support dynamic scoping strategies. Still, Smalltalk's meta-objects could be used to enable dynamic scoping [HCH08]. Implicit references are currently not addressed in Helvetia (implicit references: N/A). In Helvetia, because Smalltalk on-line programming techniques allow to define a new parser at runtime, this could be understood as a limited from of activation of language constructs, but Helvetia does not support dynamically activating and deactivating language constructs during a program run, because each embedded program is compiled only once (activation: N/A).

Pluggable Scoping in Staged Languages: None of the current multi-stage language embedding approaches [SBP99, COST04, SCK04, Tra08] addresses dynamic scoping, implicit references and dynamic activation (dynamic-scoping, implicit references, activation: N/A).

Pluggable Scoping in OO Languages: Since fluent interfaces in Evans [Eva03] and Fowler [Fow05] use Java with lexical scoping, none of the scoping feature is addressed (dynamic-scoping, implicit references, activation: N/A)

Dubochet [Dub06], Odersky et al. [OSV07], Hofer et al. [HORM08, HO10] use Scala that support only lexical scoping (dynamic-scoping: N/A). Using implicit references in embedded DSL is out of their scope (implicit references: N/A), but in Scala the end user could use Scala's import mechanism to implicitly statically bind to a member name of a certain type that is provided as a type parameters, which would enables replacing the reference dependent on its static context. In Scala, embedded language constructs cannot be dynamically activated or deactivated (activation: N/A).

4.5.2 Heterogeneous Embedding Approaches

Heterogeneous approaches have scoping mechanism in their meta-languages and their object languages, or respective target languages.

Pluggable Scoping in Embedded Compilers: Kamin's [Kam98], Elliot et al. [EFDM03], and Cuadrado et al. [CM07] use only the fixed scoping mechanism of their meta-languages. In those approaches, the generated code only uses the scoping mechanisms of its target language. No special scoping mechanism are provided or addressed (dynamic-scoping, implicit references, activation: N/A).

Pluggable Scoping in Source Transformation Languages: *MetaBorg* and TXL have similar qualities with respect to scoping. Stratego supports *generic traversals*, *generic transformations* can be used to pre-

cisely control the scoping of rewriting rules, the control- and the data-flow a program transformation rules, which enables developers to write context sensitive transformations. TXL uses its special implicit and explicit scoping mechanism to control the application of its functions and its transformation rules. However, because MetaBorg's and TXL's architecture is not homogeneous and not causally connected, the scoping mechanism of their meta-languages are not available inside embedded programs, which are compiled and executed in a target language. Both the scoping of the meta-language and the target languages are closed and developers cannot extend them. Still, in MetaBorg and TXL there is a workaround, since it would be possible for a language developer to switch the scoping mechanism of the embedded language by either generating to a dynamically or lexically scoped languages, but selecting a scoping strategy by the user is not addressed (dynamic-scoping: ◐). For an implicit reference, the language developer can implement a rewrite rule that rewrite the references name to a target language code that resolves the reference, depending on its context. MetaBorg uses this technique in its SWUL implementation to avoid that end users have repetitive write down a Java reference that is clear from the context (implicit references: ●). In TXL, scoping of functions can achieve the same (implicit references: ●). In MetaBorg and TXL, the dynamic activation of language constructs is not addressed. Theoretically, it would be possible to rewrite terms to target language code that contains indirections that allow to activate or to deactivate a certain language construct, but implementing rewrite strategy would be extraordinary complex. Unfortunately, after transformation, the indirection cannot be adapted anymore in the user domain (activation: ◐).

4.5.3 Roadmap: Scoping in Non-Embedded Approaches

The literature on providing new scoping mechanisms is mostly for interpreters and compiler.

Interpreter Approaches: Interpreters have the advantage that they have full access to runtime information, which allows them to relatively easy control scoping.

There are approaches that discuss scoping of paradigm-specific language constructs. To control aspects, in particular, Tanter [Tan09] has studied scoping strategies of language constructs in Scheme, such as variable bindings, aspects [Tan08], layers.

Most works address implementing various scoping schemes for interpreters, an overview is given in [AS96]. Experimental forms of scoping schemes have been implemented for interpreters, such as in *quasi-static abstractions* [IJF92, LF93].

There are many works for controlling paradigm-specific constructs in non-embedded-approaches, such as for aspects in GPLs, but there is little work on scoping in DSLs.

In particular interesting is the scoping and activation of aspects in DSLs, as dynamically scoped aspects are supported in the BPEL workflow language [CM04, Cha08].

Compiler Approaches: In general, when implementing a compiler/interpreter for a language, the language developer can determine the scoping mechanism of the implemented language. There are different techniques to implement scoping, such as *symbol* and *look up tables*, or *reference attributes* in attribute grammars [Knu90]. Further, for developers, it is at their hands to implement a name analysis using the corresponding compiler/interpreter approach. In most compiler frameworks, developers can implement a name analysis in form of a visitor that traverses AST nodes and that resolves e.g. the reference attribute during the compilation process.

But, it is more flexible if the compiler separates name analysis from other compiler phases, such as it is supported by the special compiler construction systems *Eli* [KW91]. Eli has systematic support for pluggable scoping, since it allows defining scoping rules, such as *static scoping rules*, and rules that take into account the *syntactic* and the *semantic context*.

4.6 Support for Pluggable Analyses

When language developers want to implement DSLs with requirements for syntactic or semantic analyses, because of the missing support for analysis in many embedding approaches, existing DSL surveys

and comparisons do not recommendation developers to use embedding approaches [MHS05, KLP+08]. While indeed it is very difficult to support analyses in homogeneous embedding approaches, many heterogeneous embedding approaches have some support for syntactic and semantic analysis.

To review the support for pluggable analyses, this section discusses whether (1) the embedding approach addresses the implementation of *syntactic analyses*, such as analyzing DSL program code for code conventions (cf. Section 3.6.1) and (2) *semantic analyses*, such as performing an abstract interpretation to validate a domain-specific constraint (cf. Section 3.6.2).

4.6.1 Homogeneous Embedding Approaches

Pluggable Analyses in Functional Host Languages: Hudak's approach [Hud98] does not address implementing custom syntactic or semantic analyses, the approach completely relies on the host language to analyze embedded programs (syntactic analyses, semantic analyses: N/A).

Carette et al. [CKS09] laid the basis for analyses of programs in pure embedding approaches. To implement an analysis, the language developer defines a function, which is an *explicit fold* over the HOAS in an alternative interpreter of the language. By abstracting over program semantics, the alternative interpreter is plugged onto the program to evaluate the fold. Folding over the HOAS is comparable to traversing an AST representation of the program, but developer need to implement a total function over all AST types, i.e. the complete AST must be known, which disallows composition of analyses for ASTs of different languages. Unfortunately, Carette et al. did not implement sophisticated analyses, and later, it was identified that implementing analysis via folds is complicated [ALY09] (syntactic analyses, semantic analyses: ◔).

In particular, Atkey et al. [ALY09] address the problem that other pure embedding approaches lack adequate support for analysis. They identified that when embedded DSL programs are encoded with HOAS, it is hard to implement analysis as explicit folds over the HOAS. They found that analysis can be easier expressed when the program is represented with de Bruijn indices. However, the HOAS encoding should not be totally abandoned, since still it is needed for an efficient interpretation of programs. They address this problem by supporting both encoding of programs—HOAS and de Bruijn—whereby they can map one encoding with an isomorphism into the other. Because their technique allows converting the program encoding, language developer can always choose the best encoding, which is the best for their needs. By unembedding HOAS to de Bruijn (first-order abstract syntax), they enable intensional analysis. Unfortunately, they do not demonstrate to what extend it is possible to write complex analyses, and similar to Carette analysis are implemented as a total function that must know the complete AST, which disallows composition of analyses for ASTs of different languages (syntactic analyses, semantic analyses: ◖).

Pluggable Analyses in Dynamic languages: Peschanski's jargons [Pes01], the ad-hoc embedding approaches in Groovy and Ruby, π [KM09], these approaches completely relies on the host language to analyze embedded programs custom syntactic or semantic analyses are out of scope (syntactic analyses, semantic analyses: N/A).

TwisteR [AO10] in particular addresses the missing support for dynamic analysis in Ruby. They use extend the concept of the meta-aspect protocol with so-called *meta-join points* to allow language developer to intercept the evaluation of expression types at the basic-block level in Ruby methods, which can be either Ruby expression types or expression types of the embedded language. To implement an analysis, the language developer implements aspect that intercept those join points an extract information at them. There is no different whether the aspect extracts syntactic or semantic information, or both. As TwisteR support dynamic aspects, the developer can dynamically activate and deactivate analyzes. To analyze a program, it is first pre-processed to instrument it, so that it exposes the meta-join points, and then at runtime the aspects dynamically compose the analysis logic into the running program. Unfortunately, with their technique, a program cannot be analyzed independently without executing the program. Consequently, TwisteR cannot analyze a program offline, before executing it, which disallows

checking programs before their execution. Because, TwisteR analyzes programs at runtime, there is always a runtime overhead due to executing the analysis logic (syntactic analyses, semantic analyses: ◖)

In Renggli [RGN10], Helvetia comes with a principle support for analyses, since Helvetia can intercept between parsing and compilation to Smalltalk. To analyze a program, theoretically, the developer could hook in before compilation and analyze the AST using transformation rules. This way, analyses can plug in custom semantic analyses before, instead of, and after the default semantic analysis of the host compiler. Composability of rules is limited, since there is no mechanism to control conflicts between transformation rules of multiple analyses. In the end, implementing analyses as transformation is awkward. It is not clear whether Helvetia supports global analyses that would need to combine analysis results from all transformation rules. Since they do not demonstrate concrete syntactic and semantic analysis, it is not clear whether Helvetia's support for analysis is adequate (syntactic analyses, semantic analyses: ◐).

Pluggable Analyses in Staged Languages: Multi-stage language embedding approaches [SBP99, SCK04, Tra08] (and MetaOCaml in [COST04]) discuss analysis, such as domain-semantic semantic analyses for execution and optimization, but in a very different sense than custom domain-specific analysis (syntactic analyses, semantic analyses: ◐).

In particular interesting is *intensional analysis* as Czarnecki et al. discuss for TemplateHaskell in [COST04]. Intensional analysis allows introspecting the code of embedded programs, but again the use of intensional analysis is very different from domain-specific analyses of syntax and semantics. Czarnecki uses pattern matching on algebraic data types used to represent expressions of DSL programs for analysis. Pattern matching on algebraic data types allows implementing syntactic and semantic analysis. For example, they have implement a semantic analysis as part of an optimization that identifies computational expensive expressions using pattern matching and then rewrite such expressions to corresponding optimized expressions. However, semantic analyses in abstract domains have been out of scope. Since an analysis needs to rewrite expressions to a distinct domain, which may be structurally equivalent to the analyzed AST, but one cannot manipulate the same AST instance. there is no general pluggability and composability (syntactic analyses, semantic analyses: ◐).

Pluggable Analyses in OO Languages: Evans [Eva03] and Fowler [Fow05] does not address implementing custom syntactic or semantic analyses with fluent interfaces (syntactic analyses, semantic analyses: N/A).

Garcia [Gar08] supports generating fluent interfaces which contains checks generated from OCL-like constraints, but this disallows an embedded program from being analyzed independently from its execution (syntactic analyses, semantic analyses: ◐).

For Dubochet [Dub06], Odersky et al. [OSV07], custom syntactic or semantic analyses are out of scope (syntactic analyses, semantic analyses: N/A).

In contrast, Hofer et al. [HORM08, HO10] use Carette's technique to abstract over syntax they can analyze programs. To make a program analyzable, the language developer has to implement an AST for it. The AST representation of a program can be obtained, by plugging an alternative evaluator to a program that creates the AST form the expressions. They adapt also use different encodings, similar to [ALY09], but add Church and Scott the set of supported encodings to it. These encodings have different properties w.r.t. extensibility and composability of analyses that are implemented from them. However, they have identified that none of these encodings allows both composition and extension at the same time. Unfortunately, in contrast to [ALY09], they do not support isomorphic converting between the encoding forth and back. Dynamic analysis like TwisteR supports is out of scope (syntactic analyses, semantic analyses: ◖).

4.6.2 Heterogeneous Embedding Approaches

Since heterogeneous approaches often require the language developer to implement an AST representation of the program, most of them support syntactic and also semantic analysis, but there are different qualities.

Pluggable Analyses in Embedded Compilers: Kamin's [Kam98] does not address implementing custom syntactic or semantic analyses of DSL programs (syntactic analyses, semantic analyses: N/A).

Elliot et al. [EFDM03] support semantics analysis with a different focus, namely only for optimizations. Unfortunately, a program cannot be analyzed independent from its execution (syntactic analyses, semantic analyses: ◐).

Cuadrado et al. [CM07] support generating programs with constraints, but this disallows an embedded program from being analyzed independently from its execution (syntactic analyses, semantic analyses: ◐).

Pluggable Analyses in Source Transformation Languages: *MetaBorg* [BV04] has exceptional good support for syntactic analysis, but there are little limitations with respect to semantic analysis. MetaBorg can generate analyzers for programs of in arbitrary syntax of a CFG (syntactic analyses: ●). After parsing, there is an extensible AST representation, with AST nodes that can be annotated with arbitrary information, which realizes an attribute grammars, where analyzes can store intermediary values and result values of modular analyzes. Program generation can be organized into modular phases, whereby incrementally adding information into the AST representation of a program. Technically, there is no different whether analysis process syntactic or semantic information. However, unfortunately, MetaBorg does not adequately support semantic analyses, since it are not integrated into the semantic analyses phase of the target compiler, not all semantic information of the target program is not available. To still enable semantic analyses for heterogeneous embedding approaches, there have been experiments with *MetaBorg* to integrate it with an extended type analysis in a host language [dG05]. (semantic analyses: ◖).

TXL [BV04] has good support for syntactic and semantic analysis, but there only little limitations. TXL can analyze LL(*) but not all CFGs. After parsing, there is an extensible AST representation. On the AST, rich syntactic and semantic analyses can be implemented as functions. Each function pattern that matches AST nodes, it can retrieve any information for them. If a function's pattern does not match it returns the unchanged scope. In other words, each function is total on any AST. Analysis can store information in the AST and replace nodes. Because of the functional properties of analyses, multiple of them can be easily composed with fix-point semantics. Still, in TXL, there is also no integration with the target compiler, thus not all semantic information of the target program is available (syntactic analyses, semantic analyses: ◖).

4.6.3 Roadmap: Analyses in Non-Embedded Approaches

Often in non-embedded approaches, custom analyses are implemented with OO extensions mechanisms, but there are practical issues. Custom analyses often use the standard *visitor pattern* [GHJV95] or some extension of it. The standard visitor pattern has well-known design issues that cannot be solved in with single inheritance. When composing several languages it is hard to add new AST nodes into a hierarchy with existing visitors, because of the bad extensibility of the standard visitor pattern. Therefore, often some extended from of visitor pattern is used.

Interpreter Approaches: The most important advantage of the interpreter approach is that it is simple and flexible to implement such analyses. Interpreters support both static and dynamic program analysis [TCL+00]. Interpreters have the advantage compared to other approaches that they have access to run-time information of the interpreted program and to access the language internals (such as the call stack, control flow information). Therefore, interpreters can enable analyses that are undecidable before runtime [Ayc03] (e.g., in contrast to static analyses). A disadvantage is that when interpreters are manually implemented [AS96], semantic analyses must be also hand-written, and often they are not

implemented modularly. Moreover, in the interpreter approach, often there is no special facility for incremental extensible and pluggable semantic analyses. Still, in comparison to homogeneous embedding, most interpreters are not meta-circular and not causally connected, which is needed to exchange objects between the interpreter and the running program, which can simply implementing analyses.

Compiler Approaches: In general, compilers have good support for syntactic and semantic analyses. Analyses in compilers have a long tradition, in particular implementing analyses for particular compiler phases and for *abstract interpretations* [CC77, Cou96]. An extensive overview of compiler techniques for implementing analyses is given by [FS03].

Technically, often compiler approaches partially generate analyses as AST walkers from a formal specification, such as *ANTLR*. Often the generated classes implement a visitor pattern. Alternatively, sometimes the language developer has to extend a special framework class to implement a new AST visitor, such as in *SableCC*. Both syntactic (i.e. lexical) and semantic analyses are supported. Analyses can be context-dependent, such as in *ANTLR* that support semantic predicates to make an analysis context-dependent.

Compiler approaches for DSLs (e.g., most parser generators) allow multiple syntactic analyses on the ASTs of a program before the rewritten program is finally converted to an executable form (e.g., by a host compiler), such as in ANTLR. Whereby, an analysis can rely on the results of the previous analysis, and incrementally stores its own results in AST nodes. A problem of multiple analyses to the same AST nodes is the presence of side effects. The theoretical background of composing modular analyses is discussed in depth in [CC02].

There are several works that propose to extend compilers by new mechanisms. For example, in [LJ05], propose an extension to the Haskell compiler and that allows implementing analysis as libraries for the extended compiler. Implementing analysis requires the developer to use the extended features, but implementing analysis as a libraries is in the same vein as embedding analysis.

In sum, pluggable analyses are more or less well supported by compilers. In contrast to interpreters, generally it is harder to plug in analyses with compilers. But also with interpreters, it is not possible to plug in analyses in the user domain, because both approaches are not homogeneous with the target languages and their architecture are not causally connected with the runtime in the user domain.

COTS-based approaches: COTS-based approaches have principle support for analyses. While they have good support only syntactical analyses, semantic analyses are hard to implement.

When using XML for DSLs, in principle, semantic analyses of XML document can be implemented using XSLT, but there are important practicable limitations that lead to non-maintainable implementations because of the poor readability of XML and XSLT [CH06].

COTS-based approaches (such as XML, UML, EMF) provide support for syntax analyses though standard and custom components for validation, syntax analyses and transformations in XML and UML/EMF.

When using XML for DSLs, XML document can be syntactically analyzed using XQuery [BCF07] and XPath [W3C07]. Semantic analysis requires special tools, or XSL transformations, but it is rather awkward to implement a semantic analysis via transformations.

For UML and EMF, there are syntactic analyses to validate the syntactic correctness of models. Further, for semantic analysis, there is the possibility to define constraints on models, e.g. using the *Object Constraint Language* (OCL), for which there are special tools and simulators that validate the constraints.

4.7 Support for Pluggable Transformations

Current DSL surveys and comparisons do attribute embedding approaches with similar limitations for transformation as for analysis, and therefore do not recommend the use embeddings when transformation is required [MHS05, KLP+08]. While indeed it is very difficult to support transformation in homogeneous embedding approaches, transformation is natural to heterogeneous embedding approaches.

To review the support for pluggable transformations, this section discusses whether (1) the embedding approach addresses the implementation of *syntactic/semantic transformations*, such as optimizing a DSL program (cf. Section 3.6.1) and (2) *dynamic transformations*, such as adaptive-dynamic optimizations (cf. Section 3.6.2).

4.7.1 Homogeneous Embedding Approaches

Pluggable Transformations in Functional Host Languages: Hudak's approach [Hud98] does not address implementing custom program transformations (syntactic/semantic transformations, dynamic transformations: N/A).

With abstracting over semantics, Carette et al. [CKS09] would support transformations, but implementing custom transformations are out of scope (syntactic/semantic transformations, dynamic transformations: ◐).

Atkey et al. [ALY09] addresses the problem to transform programs from HOAS to de Bruijn encoding and back, however they do not address custom the implementation of transformations (syntactic/semantic transformations, dynamic transformations: ◐).

Pluggable Transformations in Dynamic languages: Peschanski's jargons [Pes01], the ad-hoc embedding approaches in Groovy and Ruby, π [KM09], these approaches do not address implementing custom program transformations (syntactic/semantic transformations, dynamic transformations: N/A).

TwisteR [AO10] targets dynamic analysis, but not static or dynamic transformation. TwisteR instruments every embedded program with a pre-processor to expose meta-join-points, but implementing custom transformations are out of scope (syntactic/semantic transformations: ◐). Pre-processing happens at start-up time. Although developer could use dynamic aspects to adapt running programs, implementing concrete transforms has been out of scope (dynamic transformations: ◐).

Renggli's Helvetia [RGN10] heavily relies on transformations to transform concrete to abstract syntax. Moreover, they discuss syntactic/semantic transformations to instrument programs for support for transactional. However, in general, it is not possible to combine several transformations, because developers cannot combiner transformation rules (syntactic/semantic transformations: ◖). Although, static transformation happens on-demand, it is not dynamic since every program is transformed only once, and it is not clear how their techniques can be used for transformations that depend on the runtime context (dynamic transformations: N/A).

Pluggable Transformations in Staged Languages: Multi-stage language embedding approaches support transformation of a set of AST nodes from one stage to another set of AST nodes in another stage, whereby current approaches consider only transformations on the same AST. To transform the AST, every stage can reify part of the AST, manipulate it by replacing some AST node by other AST node of the same AST, and by reflecting is to the next stage. Custom transformations can be provided as new stages (syntactic/semantic transformations: ●). Since every reflective access to the AST must be statically quoted to reflect it, it is not possible to dynamically change reflective access to the AST at runtime (dynamic transformations: N/A).

Tratt [Tra08] compile-time meta-programming naturally support static transformation with its rewrite rules (syntactic/semantic transformations: ●), however dynamic transformations are not supported, since the already compiled converge code cannot be adapted anymore (dynamic transformations: N/A).

Pluggable Transformations in OO Languages: Evans [Eva03] and Fowler [Fow05] does not support implementing custom transformations of DSL programs (syntactic/semantic transformations, dynamic transformations: N/A).

Garcia [Gar08] transforms EMF models to fluent interfaces in Java code, but he does not support implementing custom transformations (syntactic/semantic transformations, dynamic transformations: N/A).

For Dubochet [Dub06], Odersky et al. [OSV07], custom transformations are out of scope (syntactic/semantic transformations, dynamic transformations: N/A).

In contrast, Hofer et al. [HORM08, HO10] support modular syntactic and semantics transformations In [HO10], they address the problem of composing several transformations of one language. In particular, they use four kinds of encodings that have different qualities, in particular, none of the encodings is at the same time extensible, composable, and supporting local as well as global transformations. Further, they cannot compose transformations from multiple languages, because they cannot handle syntactic conflicts between two languages (syntactic/semantic transformations, dynamic transformations: ◖).

4.7.2 Heterogeneous Embedding Approaches

Since heterogeneous approaches often require the language developer to implement an AST representation of the program, most of them support syntactic and also semantic analysis, but there are different qualities.

Pluggable Transformations in Embedded Compilers: Kamin's [Kam98] does not address implementing custom syntactic or dynamic transformations of DSL programs (syntactic/semantic transformations, dynamic transformations: N/A).

Elliot et al. [EFDM03] support semantics analysis with a different focus, namely only for optimizations. Unfortunately, a program cannot be transformed independent from its execution, which disallows custom transformations (syntactic/semantic transformations: ◗), and dynamic transformations are not possible at compile-time (dynamic transformations: N/A)

Cuadrado et al. [CM07] support generating programs with constraints, but they do not support several transformations of an embedded program (syntactic/semantic transformations: ◗), and dynamic transformations also are not possible (dynamic transformations: N/A)

Pluggable Transformations in Source Transformation Languages: *MetaBorg* [BV04] has exceptional good support for transformations. MetaBorg synthesizes program generators for arbitrary CFGs. Developers can implements transformations in modules and they can compose several modular transformations. MetaBorg can transform the AST representation of a program with declarative rewrite rules. Such rules can be explicitly scoped via dynamic conditions to enable context-sensitive transformations. Further, generic rewriting strategies can be used to abstract over syntax. However, there is no implicit mechanism that scopes rewrite rules or that prevents side-effects, which developers need to prevent by explicitly restricting their rules to an exclusive context. Unfortunately, MetaBorg does not support to plug in new transformation in the user domain (syntactic/semantic transformations: ◖). Context-sensitive transformations depend on the static context, but they cannot take into account the dynamic context of program execution. This is because, MetaBorg's architecture is not integrated with the target code's compiler, and there is no causal connection, i.e. between the running code and the code of the meta-program (dynamic transformations: N/A).

TXL [BV04] also has exceptional good support for transformations. Although it is limited to transformations of LL(*), in particular interesting are TXL's rewrite rules, functions and their scopes, that implicitly take into account rule hierarchies and that can be explicitly scoped to local and global transformations. Unfortunately, TXL does also not support to plug in new transformation in the user domain (syntactic/semantic transformations: ◖), and TXL's architecture is also not causally connected with the runtime of the generated program (dynamic transformations: N/A).

4.7.3 Roadmap: Transformations in Non-Embedded Approaches

The related work on non-embedded DSLs can be distinguished w.r.t. if there is a DSL for which (1) custom transformation are implemented only once and (2) generic DSL approaches that allow to add new transformations to an existing DSL.

Custom Transformations: Custom transformations for DSLs usually are implemented from scratch and it is a rather large investment to take into account the details of the domain semantics. But, when the returned benefits are large, it is considered worth to invest the additional costs for transformations.

For example, dynamic transformations enable special dynamic optimizations for SQL [DD89]. Today, most *SQL* implementations of database management systems perform dynamic and adaptive optimizations of SQL queries at runtime. But, such optimizations itself are hard-wired in the SQL implementation. But implementing hard-wired transformation is only feasible in special domains, where there are enough end users benefits to outweigh the costs. Hard-wired transformations are hard to implement and expensive to maintain. They cannot be easily adopted and reused in other domains.

Unfortunately, there is a lack of systematic approach to build dynamic transformations into new languages.

Generic Transformations: Often in non-embedded approaches, similarly to analyses, custom transformation are implemented with OO extensions mechanisms. In general, compilers have good support for pluggable transformations with AST visitors, such as in *ANTLR*, and *SableCC*. While pluggability of transformations is supported, extensibility of transformations is a problem because of the aforementioned problems. Therefore, often some an extended from of the visitor pattern is used also for transformations [NCM03], in order to allow both pluggable and extensible traversals. An overview of transformations with compiler techniques is given by [FS03].

Technically, transformations are implemented similarly to analyses. Most approaches support only static transformations (e.g. compilation to a GPL). Some approaches allow making transformation context-dependent, such as *ANTLR* uses semantic predicates for scoping transformation rules.

It is rather hard to compose several transformations, since one transformation may violate the assumptions of another transformation. Often semantic dependencies have to be controlled by the developers for which they can use *semantic predicates*, such as provided in *ANTLR*.

Complicated combinations of multiple AST transformations need be scheduled without circularities to be correct. Only a few approaches support developer by scheduling transformations.

There are automatically scheduling interpreters. For example, Jourdan [Jou84] proposes an evaluator that is based on attribute grammars. To allow implicit correct composition of semantics, the evaluator detects circularities for synthesized or inherited attribute at run-time and implements an optimal dynamic *evaluation by need* of those attributes.

There are also automatically scheduling compilers. For example, JastAdd [HM03, EH07b] also uses attribute grammars and implicitly resolves attribute at compile-time. Unfortunately, it is not possible to build transformations that depend on the runtime context.

In sum, transformations are more or less well supported by interpreters/compilers, but generally, there is little work on dynamic transformations and it is not possible to plug-in transformations in the user domain.

COTS-based Transformations: COTS-based DSL approaches support transformations in XML and UML/EMF.

DSL programs as XML documents can be rewritten through XSL transformations (XSLT) [MHS05], e.g., in [MRD08], XSLT is used to transform a workflow DSL into executable code. Still, the applicability of XSLT for DSL implementation is rather limited by their generic syntax, because XML transformations are less readable for humans [MHS05].

For UML and EMF, there are special tools and simulators that validate the standard syntax of UML models. Some tools allow domain-specific extensions in form of UML profiles for various domains, e.g. for real-time [AdSSK06]. While it is relatively easy to provide custom syntax analyses due to the underlying generic syntax and available tools, semantic extension often need to be implemented from scratch. Syntactic transformations are addressed by model-to-model transformations [CH06]. Model-driven approaches address the problem of readability of models and transformations by defining bijective mappings from the generic model representations to an alternative textual representation with better human-readable syntax (e.g., TextUML Toolkit for UML[2], EMFText for EMF [HJK+09]).

[2] The TextUML Toolkit Homepage: `http://abstratt.com/`

For semantic transformations, UML/EMF use model generators that semantically analyze models and generate executable code out of them. QVT [Gro08] enables semantic analyses for UML models [CH06]. QVT and stepwise transformations are heavily used in the *model-driven architecture* [MSUW02] approach which promises platform independent transformations. QVT transformations are composable, but it has been shown that there are high costs to develop complete transformations, to maintain them, and to understand them for maintenance [KKS07].

Fortunately, there are domain-specific generators available for UML model extensions that take into account the additional information from UML profiles, e.g. through which the generator can incorporate addition logic for handling real-time concerns [AdSSK06]). Domain-specific models can be checked using model checking and simulation, which allows arbitrary semantic domain constraints to be validated.

For embedded languages, what can be learned from COTS-based approaches is that they allow users to relatively easy to plug in custom semantic analyses, pluggable transformations should be as easy as it is possible to add new XSLT or UML template. Further, there is a strong need for declarative query and transformation languages. Finally, one can learn from UML's tool support. But, in particular embedded approaches do not want to loose the advantage that they are causally connected with their runtime, which in COTS-based approaches is often not the case.

4.8 Summary

This chapter has given a review of support for the desirable properties by the related work on embedding approaches. While the review identified open issues and limitations with current techniques, it highlighted the most important mechanisms in existing embedding approaches. To overcome the current shortcoming of embedded approaches, the review proposes a road map for the research on embedded domain-specific languages, whereby it draws conclusions from studying the available support for the desirable properties in related work on DSLs and GPLs.

Table 4.1 gives the detailed result of this review. For the top most desirable properties (gray lines), for each sub-property, the detailed results are aggregated, such that, the best support that is available for *all* scenarios is taken into account. Whereby, when one or more scenarios are not supported at all, the corresponding cell indicates the support as only partial (◐).

From the review result, one can conclude that there is only limited support for the desirable properties. Existing embedding approaches have only full support for a few properties that are expected by language developers. In current embedding approaches, there is a lack of support for many requirements in language implementation.

For extensibility, there is good support for extensibility, except for semantic adaptations in the user domain.

For composability of languages, most embedding approaches do not allow to compose interacting languages at all. Heterogeneous embedding approaches have good support to address syntactic interactions, but they have limited support to address semantic interactions.

For composition mechanisms, in all embedding approaches, the set of composition mechanisms is closed and none of the approaches allows defining new composition operators for special language composition scenarios. In homogeneous embedding approaches, the general-purpose composition operators provided by their host languages do not adequately address many composition scenarios, particularly scenarios with interaction. In heterogeneous embedding approaches, often the general-purpose composition mechanisms supports language compositions in a declarative way. Because heterogeneous embedding approaches do not support defining new composition mechanisms that facilitate sophisticated compositions scenarios, language developers have to implement the composition semantics for every language composition.

For concrete syntax, only a few homogeneous approaches support concrete syntax while most have strong limitations. In contrast, there are heterogeneous embedding approaches that have excellent support for concrete syntax. So far, heterogeneous embedding approaches do not employ partial definition of concrete syntax. Therefore, there is no saving of the syntax costs by heterogeneous embedding approaches compared to homogeneous embedding approaches. Hence, the heterogeneous embedding approach cannot leverage one of the most competitive advantages of embedding.

For pluggable scoping, special scoping strategies for language constructs are not in the focus of today's homogeneous embedding approaches, but there is only little support by heterogeneous approaches.

For pluggable analyses, there are some but few homogeneous and heterogeneous approaches that have good support for analyses. The most important limitations are the lack of user-defined analyses and pluggable analysis at runtime.

For pluggable transformations, there are some but few homogeneous and heterogeneous approaches that have good support for transformations. The most important limitations are the lack of user-defined transfromation and support for dynamic transformations.

With this limited support for the properties, language developers cannot used current embedding approaches as an alternative to traditional non-embedded approaches for implementing sophisticated DSLs.

Due to the lack of the support for these properties, the adoption of the concept of language embedding is currently limited. To overcome current shortcomings of embedded DSL approaches, for each desirable property, researchers can follow the corresponding roadmap that this section proposes and that suggests how the embedded DSL approaches can learn from traditional non-embedded approaches.

Table 4.1: Review of the Supported Properties by Embedding Approaches

Desirable Property (◑ : partial support, ◖ : important limitations, ●: full support)	Pure Embed. [Hud96]	Tagless Embed. [CKS09]	Unembedding [ALY09]	Jargons [Pes01]	EDSL Groovy [KG07]	EDSL Ruby [TFH09]	TwisteR [AO10]	π Pattern Lang. [KM09]	Helvetia [RGN10]	Staged Interpr. [COST04]	Ext. Meta-Prog. [SCK04]	Converge [Tra08]	Fluent [Eva03, Fow05]	EMF2JDT [Gar08]	Scala EDSL [OSV07]	Polymorphic [HORM08]	Embed. Gen. [Kam98]	Embed. Compiler [EFDM03]	Ruby Gen. [CM07]	MetaBorg [BV04]	TXL [Cor06]
3.1. Extensibility	◑	◑	◑	◑	◖	◖	◖	◖	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑
3.1.1. New Keywords	●	●	●	◖	●	●	●	●	◖	◑	◖	◑	◑	◑	●	●	◖	◖	◖	◖	◖
3.1.2. Semantic Extensions	◑	◑	◑	◑	◖	◖	◖	◖	◑	◑	◑		◑	◑	◑	◑				◑	◑
3.1.2.1. Conservative Extensions	●	●	●	◑	◖	◖	◖	◖	◑	●	●		◑	◖	◖	◖				◑	◑
3.1.2.2. Semantic Adaptations					◖	◖	◖	◖	◑	◑											
3.2. Composability of Languages	◑		◑		◑	◑	◑	◑	◖				◑	◑	◑	◑				◖	◖
3.2.1. Languages without Interactions	●		●		●	●	●	●	●				◖	◖	●	●				●	●
3.2.2. Languages with Interactions	◑		◑		◑	◑	◑	◑	◖											◖	◖
3.2.2.1. Syntactic Interactions	◖		◖		◑	◑	◑	◖	◖											◖	◖
3.2.2.2. Semantic Interactions					◑	◑	◑		◖											◖	◖
3.3. Composition Mechanisms	◑		◑		◑	◑	◑	◑	◑				◑	◑	◑	◑				◑	◑
3.3.1. Syntactic Interactions	◖		◖		◑	◑	◑	◑	◑				◑	◑	◖	◖				◑	◑
3.3.1.1. Conflict-Free Compositions	◖		◖					◖	◑				◖	◖	◖	◖				●	◖
3.3.1.2. Resolving with Renaming	◖		◖		◖	◖	◖	◖							◖	◖				●	◖
3.3.1.3. Resolving with Priorities					◖	◖	◖		●						◖	◖				●	◖
3.3.2. Semantic Interactions							◑		◑											◑	◑
3.3.2.1. Crosscutting Composition							◖		◖											◑	
3.3.2.2. Resolving Composition Conflicts																				◑	◖
3.4. Concrete Syntax	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑
3.4.1 Concrete-to-Abstract							◑	●	●								◑			●	◖
3.4.2 Mixfix Operators	◑	◑	◑	◑	◑	◑	◑	●	●	●	◑	◑	◑	◑	◖	◖	◑	◑	◑	●	●
3.4.3. Overriding Host Keywords					●			●	●	●	●	●									
3.4.4. Partial Syntax																					
3.5. Pluggable Scoping				◑			◑	◑												◑	◑
3.5.1. Dynamic Scoping				◑																◑	◑
3.5.2. Implicit References								●												●	●
3.5.3. Activation of Constructs							◑	◑												◑	◑
3.6. Pluggable Analyses		◑	◖				◖		◑	◑	◑	◑				◖		◑	◑	◖	◖
3.6.1. Syntactic Analyses		◑	◖				◖		◑	◑	◑	◑				◖		◑	◑	●	●
3.6.2. Semantic Analyses		◑	◖				◖		◑	◑	◑	◑				◖		◑	◑	◖	◖
3.7. Pluggable Transformations		◑	◑				◑		◑	◑	◑	◑				◑		◑	◑	◑	◑
3.7.1. Static Transformations		◑	◑				◑		◖	●	●	●				◖		◑	◑	◖	◖
3.7.1.1. Syntactic Transformations		◑	◑				◑		◖	●	●	●				◖		◑	◑	◖	◖
3.7.1.2. Semantic Transformations		◑	◑				◑		◖	●	●	●				◖			◑	◖	◖
3.7.2. Dynamic Transformations							◑														

5 Conclusion

In this report, a set of desirable properties for language embedding approaches have been identified by related work on non-embedded language implementation approaches. The report's premise is that a language embedding approach only facilitates the language developers with their tasks in an adequate way, when it supports all these properties for developing and evolving languages.

Table 5.1 presents a summary of the review's result. The aggregated values clearly indicate the current limitations of the language embedding approaches, which miss important support for properties that are often supported by non-embedded approaches.

Table 5.1: Review Summary of the Supported Properties by Embedding Approaches

Desirable Property (◐: partial support, ◖: important limitations, ●: full support)	Pure Embed. [Hud96]	Tagless Embed. [CKS09]	Unembedding [ALY09]	Jargons [Pes01]	EDSL Groovy [KG07]	EDSL Ruby [TFH09]	TwisteR [AO10]	π Pattern Lang. [KM09]	Helvetia [RGN10]	Staged Interpr. [COST04]	Ext. Meta-Prog. [SCK04]	Converge [Tra08]	Fluent [Eva03, Fow05]	EMF.JDT [Gar08]	Scala EDSL [OSV07]	Polymorphic [HORM08]	Embed. Gen. [Kam98]	Embed. Compiler [EFDM03]	Ruby Gen. [CM07]	MetaBorg [BV04]	TXL [Cor06]
3.1. Extensibility	◐	◐	◐	◐	◖	◖	◖	◖	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐
3.2. Composability of Languages	◐		◐		◐	◐	◐	◐	◖				◐	◐	◐	◐				◖	◖
3.3. Composition Mechanisms	◐		◐		◐	◐	◐	◐	◐				◐	◐	◐	◐				◐	◐
3.4. Concrete Syntax	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐
3.5. Pluggable Scoping				◐			◐	◐												◐	◐
3.6. Pluggable Analyses		◐	◖				◖		◐	◐	◐	◐				◖		◐	◐	◖	◖
3.7. Pluggable Transformations		◐	◐				◐		◐	◐	◐	◐				◐		◐	◐	◐	◐

There is a lack of support for many requirements in language implementation, particularly for: extensibility with semantic adaptations, composition of interacting languages, new composition mechanisms, partial concrete syntax, pluggable scoping, analyses and transformations. Unfortunately, due to the lack of this support, the adoption of the concept of language embedding is currently limited. The future research on language embedding needs to address these drawbacks in order to make embedding an competitive approach to traditional techniques.

Bibliography

[AAB+07] Assaf Arkin, Sid Askary, Ben Bloch, et al. Web Services Business Process Execution Language 2.0, OASIS Standard. Technical report, Organization for the Advancement of Structured Information Standards (OASIS), April 2007.

[ADDH10a] Toheed Aslam, Jesse Doherty, Anton Dubrau, and Laurie Hendren. Aspectmatlab: an aspect-oriented scientific programming language. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, New York, NY, USA, 2010. ACM.

[ADDH10b] Toheed Aslam, Jesse Doherty, Anton Dubrau, and Laurie Hendren. Aspectmatlab: an aspect-oriented scientific programming language. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, New York, NY, USA, 2010. ACM.

[AdSSK06] L. Apvrille, P. de Saqui-Sannes, and F. Khendek. TURTLE-P: a UML profile for the formal validation of critical and distributed systems. *Software and Systems Modeling*, 5(4):449–466, 2006.

[ÅEH08] J. Åkesson, T. Ekman, and G. Hedin. Development of a Modelica Compiler Using JastAdd. *Electronic Notes in Theoretical Computer Science*, 203(2):117 – 131, 2008. Workshop on Language Descriptions, Tools, and Applications (LDTA'07).

[AET08] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. Modularity First: A Case for mixing AOP and Attribute Grammars. In *AOSD*, pages 25–35, 2008.

[AFG+00] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA*. ACM Press New York, NY, USA, 2000.

[AKRS08] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. Metamodeling with MOFLON. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science (LNCS)*, pages 573–574, Heidelberg, October 2008. Springer Verlag. tool description.

[Alm] D. Almear. SQL AOP becomes SQXML AOP. `http://almaer.com/blog/sql-aop-becomes-sqxml-aop`.

[ALSU07] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools (2nd Edition)*. Addison-Wesley, 2007.

[ALY09] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 37–48, New York, NY, USA, 2009. ACM.

[AMH90] Mehmet Akşit, Rene Mostert, and Boudewijn Haverkort. Compiler Generation based on Grammar Inheritance, 1990.

[AO10] Michael Achenbach and Klaus Ostermann. A Meta Aspect Protocol for Developing Dynamic Analyses. In *Proceedings of the Runtime Verification Conference*, 2010.

[AS96] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1996.

[Asp] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[Aßm03] U. Aßmann. *Invasive software composition*. Springer-Verlag New York Inc, 2003.

[ASSS09] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022, New York, NY, USA, 2009. ACM.

[ATC+05] P. Avgustinov, J. Tibble, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, and G. Sittampalam. ABC: An Extensible AspectJ Compiler. In *Conference on Aspect-oriented Software Development*, pages 87–98, 2005.

[Atk09] Robert Atkey. Syntax for free: Representing syntax with binding using parametricity. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-02273-9_5.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[BCF07] Scott Boag, Don Chamberlin, and Mary F. Fernández. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery/`, 2007.

[BdGV06] Martin Bravenboer, Renï£i de Groot, and Eelco Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In Ralf Lï£immel, Joï£io Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 297–311. Springer Berlin / Heidelberg, 2006. 10.1007/11877028_10.

[BHMO04] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD*, pages 83–92, 2004.

[Bon03] J. Bonér. AspectWerkz - Dynamic AOP for Java. `htt://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf`, 2003.

[Bos00] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley Professional, 2000.

[BP01] Jonthan Bachrach and Keith Playford. The java syntactic extender (jse). *SIGPLAN Not.*, 36(11):31–42, 2001.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*. ACM, 2004.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252, New York, NY, USA, 1977. ACM.

[CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis. *Compiler Construction*, 2304:263–283, 2002. 10.1007/3-540-45937-5_13.

[CD08] Pascal Costanza and Theo D'Hondt. Feature Descriptions for Context-oriented Programming. In *2nd International Workshop on Dynamic Software Product Lines (DSPL'08)*, 2008.

[CH05] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-Oriented Programming: An Overview of ContextL. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Cha08] Anis Charfi. *Aspect-Oriented Workflow Management: Concepts, Languages, Applications*. PhD thesis, TU Darmstadt, 2008. Darmstadt, Techn. Univ., Diss., 2007.

[CHHP91] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. *Computer Languages*, 16(1):97 – 107, 1991.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5(2):56–68, 1940.

[CKS07] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally Tagless, Partially Evaluated. In *Asian Symposium on Programming Languages and Systems*, 2007.

[CKS09] J. Carette, O. Kiselyov, and C. Shan. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[Cle07] T. Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2007.

[CM04] Anis Charfi and Mira Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. In *European Conference on Web Services*, 2004.

[CM07] Jesús Sánchez Cuadrado and Jesús García Molina. Building Domain-Specific Languages for Model-Driven Development. *IEEE Softw.*, 24(5):48–55, 2007.

[Cor06] James R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190 – 210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04).

[COST04] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in metaocaml, template haskell, and c++. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-25935-0_4.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[Dav03] James Davis. Gme: the generic modeling environment. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–83, New York, NY, USA, 2003. ACM.

[DD89] C.J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley New York, 1989.

[DDLM07] Marcus Denker, Stéphane Ducasse, Andrian Lienhard, and Philippe Marschall. Sub-Method Reflection. *Special Issue TOOLS Europe 2007*, 6(9), 2007.

[DFF+09] Z. Drey, C. Faucher, F. Fleurey, V. Mahé, and D. Vojtisek. Kermeta language, 2009.

[dG05] R de Groot. Design and Implementation of Embedded Domain-Specific Languages. Master's thesis, Utrecht University, The Netherlands, 2005.

[DMB09] Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. The Art of the Meta-Aspect Protocol. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD)*, pages 51–62, New York, NY, USA, 2009. ACM.

[DMFM10] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A Dynamic Software Product Line Approach Using Aspect Models at Runtime. In *First Workshop on Composition and Variability'10 Rennes*, 2010.

[DN09] N.A. Danielsson and U. Norell. Parsing Mixfix Operators. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL'08)*. Springer Berlin / Heidelberg, 2009. (to appear).

[DSLB03] Peter Denno, Michelle Potts Steves, Don Libes, and Edward J. Barkmeyer. Model-driven integration using existing models. *IEEE Softw.*, 20(5):59–63, 2003.

[Dub06] G. Dubochet. On Embedding Domain-Specific Languages with User-friendly Syntax. In *ECOOP Workshop on Domain-Specific Program Development*, 2006.

[Ear68] J. Earley. *An Efficient Context-Free Parsing Algorithm*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, USA, 1968.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[EFdM00] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, pages 9–26. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-45350-4_5.

[EFDM03] C. Elliott, S. Finne, and O. De Moor. Compiling Embedded Languages. *Journal of Functional Programming*, 13(03):455–481, 2003.

[EH07a] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*, pages 1–18, New York, NY, USA, 2007. ACM.

[EH07b] Torbjï£ırn Ekman and Gï£ırel Hedin. The JastAdd System – Modular Extensible Compiler Construction. *Science of Computer Programming*, 69(1-3):14–26, 2007. Special Issue on Experimental Software and Toolkits.

[EI05] Jacky Estublier and Anca Daniela Ionita. Extending uml for model composition. In *Proceedings of the 2005 Australian conference on Software Engineering (ASWEC '05)*, pages 31–38, Washington, DC, USA, 2005. IEEE Computer Society.

[ELKP04] Johann Eder, Marek Lehmann, Christian Koncilia, and Horst Pichler. Using ontologies to compose transformations of schema based documents. In *INTEROP-EMOI Workshop (at 16th Conference on Advanced Information Systems Engineering, CAiSE 2004)*, pages 315–318, Riga, Latvia, 6 2004. Faculty of Computer Science and Information Technology.

[ES01] Johann Eder and Walter Strametz. Composition of xml transformations. In *2nd International Conference on Electronic Commerce and Web Technologies (EC-Web 2001)*, pages 71–80, Munich, Germany, 9 2001. Springer Verlag.

[ES06] M. Emerson and J. Sztipanovits. Techniques for metamodel composition. In *6th Workshop on Domain Specific Modeling (at OOPSLA)*, pages 123–139, 2006.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[EVI05a] J. Estublier, G. Vega, and A.D. Ionita. Composing domain-specific languages for wide-scope software engineering applications. *LNCS*, 3713:69, 2005.

[EVI05b] Jacky Estublier, German Vega, and Anca Daniela Ionita. Composing Domain-Specific Languages for Wide-scope Software Engineering Applications. In *Proceedings of MODELS/UML*, 2005.

[FC09] Jose Falcon and William Cook. Gel: A generic extensible language. In Walid Taha, editor, *Domain-Specific Languages*, volume 5658 of *Lecture Notes in Computer Science*, pages 58–77. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03034-5_4.

[Fel90] M. Felleisen. On the Expressive Power of Programming Languages. *3rd European Symposium on Programming (ESOP'90)*, 432:134–151, 1990.

[FETD07] Johan Fabry, Éric Tanter, and Theo D'Hondt. ReLAx: Implementing KALA over the Reflex AOP Kernel. In *Workshop on Domain-Specific Aspect Languages*, 2007.

[Fok95] Jeroen Fokker. Functional Parsers. *Advanced Functional Programming*, 925:1–23, 1995. 10.1007/3-540-59451-5_1.

[For04] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *ACM SIGPLAN Notices*, 39(1):122, 2004.

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

[FP06] Steve Freeman and Nat Pryce. Evolving an Embedded Domain-Specific Language in Java. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, pages 855–865, New York, NY, USA, 2006. ACM.

[FPS+02] B. Folliot, I. Piumarta, L. Seinturier, C. Baillarguet, C. Khoury, A. Leger, and F. Ogel. Beyond flexibility and reflection: The virtual virtual machine approach. In Dan Grigoras, Alex Nicolau, Bernard Toursel, and Bertil Folliot, editors, *Advanced Environments, Tools, and Applications for Cluster Computing*, volume 2326 of *Lecture Notes in Computer Science*, pages 277–280. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-47840-X_2.

[FS03] Thomas Fahringer and Bernhard Scholz. *Advanced Symbolic Analysis for Compilers: New Techniques and Algorithms for Symbolic Program Analysis and Optimization*. Springer-Verlag, Berlin, Heidelberg, Germany, 2003.

[GA07] Vaidas Gasiunas and Ivica Aracic. Dungeon: A Case Study of Feature-Oriented Programming with Virtual Classes. In *Proceedings of the 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 2007.

[Gar08] Miguel Garcia. Automating the Embedding of Domain Specific Languages in Eclipse JDT. `http://www.eclipse.org/articles/Article-AutomatingDSLEmbeddings/`, July 2008.

[GH98] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an Object-Oriented Compiler Framework. *Technology of Object-Oriented Languages, International Conference on*, 0:140, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[Gri04] R. Grimm. Practical Packrat Parsing. Technical Report TR2004-854, New York University, Dept. of Computer Science, 2004.

[Gri06] R. Grimm. Better Extensibility through Modular Syntax. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–51, New York, NY, USA, 2006. ACM.

[Gro] The Groovy Home Page. `http://groovy.codehaus.org/`.

[Gro08] Object Management Group. Meta object facility (mof) 2.0 query/view/transformation specification. version 1.0 (needham, ma, e.d). `http://www.omg.org/spec/QVT/1.0/`, April 2008.

[GV07] Iris Groher and Markus Voelter. XWeave: Models and Aspects in Concert. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling (AOM '07)*, pages 35–40, New York, NY, USA, 2007. ACM.

[GV08] I. Groher and M. Voelter. Using Aspects to Model Product Line Variability. In *Early Aspects Workshop (at SPLC)*, 2008.

[GWTA10] R. Garcia, R. Wolff, É. Tanter, and J. Aldrich. Featherweight Typestate. Technical Report CMU-ISR-10-115, Carnegie Mellon University, 2010.

[HBA08] W. Havinga, L. Bergmans, and M. Aksit. Prototyping and Composing Aspect Languages using an Aspect Interpreter Framework . In *ECOOP*, pages 180–206, 2008.

[HBA10] Wilke Havinga, Lodewijk Bergmans, and Mehmet Aksit. A model for composable composition operators: expressing object and aspect compositions with first-class operators. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 145–156, New York, NY, USA, 2010. ACM.

[HCH08] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with contexts. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-88643-3_9.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, pages 26–35, New York, NY, USA, 2004. ACM.

[HHJ+08] J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Aßmann. Extending Grammars and Metamodels for Reuse: the Reuseware Approach. *IET Software*, 2(3):165–184, 2008.

[HHJZ09] Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On Language-Independent Model Modularisation. *LNCS*, 5560:39–82, 2009.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. *Computer*, 41:93–95, 2008.

[Hir01] R. Hirschfeld. AspectS: AOP with Squeak. In *OOPSLA Workshop on Advanced Separation of Concerns in OO Systems*, 2001.

[Hir09] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer Berlin / Heidelberg, 2009. 10.1007/3-540-36557-5_17.

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 114–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[HJZ07] F. Heidenreich, J. Johannes, and S. Zschaler. Aspect Orientation for Your Language of Choice. In *Workshop on Aspect-Oriented Modeling (at MoDELS)*, 2007.

[HM03] G. Hedin and E. Magnusson. JastAdd - An Aspect-Oriented Compiler Construction System. *Science of Computer Programming*, 47(1):37–58, 2003.

[HO07] C. Hofer and K. Ostermann. On the Relation of Aspects and Monads. In *Workshop on Foundations of Aspect-Oriented Languages*, pages 27–33, 2007.

[HO10] Christian Hofer and Klaus Ostermann. Modular Domain-Specific Language Components in Scala. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*. ACM, 2010.

[HORM08] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic Embedding of DSLs. In *Generative Programming and Component Engineering (GPCE'08)*, pages 137–148. ACM New York, NY, USA, 2008.

[Hud96] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28(4es):196–196, 1996.

[Hud98] Paul Hudak. Modular Domain Specific Languages and Tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[IEE09] IEEE. Special issue on models@run.tim. *IEEE Computer*, 42(10), 2009.

[IJF92] John Wiseman Simmons II, Stanley Jefferson, and Daniel P. Friedman. Language Extension via First-class Interpreters. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.1396`, 1992.

[JMS10] Adrian Johnstone, Peter Mosses, and Elizabeth Scott. An agile approach to language modelling and development. *Innovations in Systems and Software Engineering*, 6:145–153, 2010. 10.1007/s11334-009-0111-6.

[Joh75] S. C. Johnson. Yacc: Yet Another Compiler Compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1975. Computing Science Technical Report.

[Jou84] Martin Jourdan. An Optimal-Time Recursive Evaluator for Attribute Grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer Berlin / Heidelberg, 1984. 10.1007/3-540-12925-1_37.

[JS80] Neil Jones and David Schmidt. Compiler generation from denotational semantics. In Neil Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 70–93. Springer Berlin / Heidelberg, 1980. 10.1007/3-540-10250-7_19.

[Kam98] S.N. Kamin. Research on Domain-Specific Embedded Languages and Program Generators. *Electronic Notes in Theoretical Computer Science*, 14:149–168, 1998.

[KAR+93] G. Kiczales, J. M. Ashley, L. Rodriguez, A. Vahdat, and D.G. Bobrow. Metaobject Protocols: Why we want them and what else they can do. *Object-Oriented Programming: The CLOS Perspective*, pages 101–118, 1993.

[KG07] D. König and A. Glover. *Groovy in Action*. Manning, 2007.

[Kic96] G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, 1996.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. In *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 285–294, New York, NY, USA, 2007. ACM.

[KL05] S. Kojarski and D.H. Lorenz. Pluggable AOP: Designing Aspect Mechanisms for Third-Party Composition. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 247–263. ACM, 2005.

[KL07] Sergei Kojarski and David H. Lorenz. AweSome: an Aspect Co-Weaving System for Composing Multiple Aspect-Oriented Extensions. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 515–534, New York, NY, USA, 2007. ACM.

[KLP+08] T. Kosar, M. López, E. Pablo, P.A. Barrientos, and M. Mernik. A Preliminary Study on Various Implementation Approaches of Domain-Specific Language. *Information and Software Technology*, 50(5):390–405, 2008.

[KM06] Roman Knöll and Mira Mezini. Pegasus: first steps toward a naturalistic programming language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 542–559, New York, NY, USA, 2006. ACM.

[KM09] R. Knöll and M. Mezini. π: a Pattern Language. *ACM SIGPLAN Notices*, 44(10):503–522, 2009.

[Kni07] G. Kniesel. Detection and Resolution of Weaving Interactions. *TAOSD: Dependencies and Interactions with Aspects*, LNCS, 2007. Special Issue on Aspect Dependencies and Interactions, edited by R. Chitchyan.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[Knu90] D.E. Knuth. The Genesis of Attribute Grammars. In *Proceedings of the International Conference WAGA on Attribute Grammars and their Applications*, page 12. Springer, 1990.

[Kod04] Viswanathan Kodaganallur. Incorporating Language Processing into Java Applications: A JavaCC Tutorial. *IEEE Software*, 21:70–77, 2004.

[KRB91] Gregor Kiczales, Jim d. Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Vi£ilkel. Monticore: Modular development of textual domain specific languages. *Objects, Components, Models and Patterns*, 11:297–315, 2008. 10.1007/978-3-540-69824-1_17.

[KVW10] L.C.L. Kats, E. Visser, and G. Wachsmuth. Pure and Declarative Syntax Definition: Paradise Lost and Regained. *Proceedings of Onward! 2010*, 2010.

[KW91] U. Kastens and W. M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28:539–558, 1991. 10.1007/BF01463944.

[Lan66] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[Lat03] Mario Latendresse. RegReg:a Lightweight Generator of Robust Parsers for Irregular Languages. *Reverse Engineering, Working Conference on*, 0:206, 2003.

[LF93] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: sharing variable bindings across multiple lexical scopes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'93)*, pages 479–492, New York, NY, USA, 1993. ACM.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, New York, NY, USA, 1995. ACM.

[LJ05] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. *SIGPLAN Not.*, 40(9):204–215, 2005.

[LK97] C. Lopes and G. Kiczales. D: A Language Framework for Distributed Programming. Technical report, Northeastern University, Boston, MA, USA, 1997.

[LLMS00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 108–118, New York, NY, USA, 2000. ACM.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Conference on Domain-Specific Languages (DSL'99)*, page 9, Berkeley, CA, USA, 1999. USENIX Association.

[LM00] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. *SIGPLAN Not.*, 35(1):109–122, 2000.

[LMB92] John Levine, Tony Mason, and Doug Brown. *Lex & yacc, 2nd edition*. O'Reilly, 1992.

[LNK+01] Ákos Lédeczi, Greg Nordstrom, Gábor Karsai, Péter Völgyesi, and Miklós Maróti. On Metamodel Composition. In *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA '01).*, pages 756–760, 2001.

[LY99] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, Boston, 1999.

[Mae87] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[ME00] J. Melton and A. Eisenberg. *Understanding SQL and Java together: a guide to SQLJ, JDBC, and related technologies*. Morgan Kaufmann, 2000.

[Mez97] Mira Mezini. Dynamic object evolution without name collisions. In Mehmet Aksit and Satoshi Matsuoka, editors, *Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, page 190. Springer Berlin / Heidelberg, 1997.

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

[MLAZ00] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Compiler/Interpreter Generator System LISA. *Hawaii International Conference on System Sciences*, 8:8059, 2000.

[MO04] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136, New York, NY, USA, 2004. ACM.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[Moo01] L. Moonen. Generating Robust Parsers using Island Grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society, October 2001.

[Mos80] Peter Mosses. A Constructive Approach to Compiler Correctness. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 449–469. Springer Berlin / Heidelberg, 1980. 10.1007/3-540-10003-2_91.

[MPO08] A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, KU Leuven, 2008. CW Reports.

[MRD08] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 815–824, New York, NY, USA, 2008. ACM.

[MSUW02] Stephen Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-driven architecture. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 233–239. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46105-1_33.

[NB09] Robert France Nelly Bencomo, Gordon Blair, editor. *1st International Workshop on Models@run-time*, 2009.

[NCM03] N. Nystrom, M.R. Clarkson, and A.C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag.

[oD65] Department of Defense. COBOL Edition 1965. Technical Report Formnr. 0-795-689, US-Government Printing Office, 1965.

[OMG04] OMG. UML 2.0 Superstructure. Technical report, Object Management Group, 2004.

[OSV07] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, Mountain View, CA, 2007.

[Paa95] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[Par93] T.J. Parr. *Obtaining Practical Variants of LL (k) and LR (k) for k > 1 by Splitting the Atomic k-Tuple*. PhD thesis, PhD thesis, Purdue University, 1993.

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

[Par08] T. Parr. The Reuse of Grammars with Embedded Semantic Actions. In *Proceedings of the 16th IEEE Conference on Program Comprehension (ICPC 2008)*, pages 5–10, 2008.

[Par10] T. Parr. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. *The Pragmatic Bookshelf*, 2010.

[PE88] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. *SIGPLAN Not.*, 23(7):199–208, 1988.

[Pes01] F. Peschanski. Jargons: Experimenting Composable Domain-Specific Languages. In *Workshop on Scheme and Functional Programming*, Firenze, Italy, 2001.

[PGA01] A. Popovici, T. Gross, and G. Alonso. Dynamic Homogenous AOP with PROSE. Technical report, Department of Computer Science, ETH Zürich, Zürich, Switzerland, March 2001.

[Pie02] B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[PS83] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, 1983.

[RG09] I. Rogers and D. Grove. The Strength of Metacircular Virtual Machines: Jikes RVM. *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*, page 235, 2009.

[RGN10] L. Renggli, T. Gîrba, and O. Nierstrasz. Embedding languages without breaking tools. *ECOOP*, 6183:380–404, 2010.

[RMH+06] D. Rebernak, M. Mernik, P.R. Henriques, D. da Cruz, and M.J.V. Pereira. Specifying Languages using Aspect-Oriented Approach: AspectLISA. In *28th International Conference on Information Technology Interfaces*, pages 695–700, 2006.

[RMHP06] D. Rebernak, M. Mernik, P.R. Henriques, and M.J.V. Pereira. AspectLISA: an aspect-oriented compiler construction system based on attribute grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):37–53, 2006.

[RMWG09] D. Rebernak, M. Mernik, H. Wu, and J. Gray. Domain-Specific Aspect Languages for Modularising Crosscutting Concerns in Grammars. *Software, IET*, 3(3):184–200, june 2009.

[Rub] Ruby Programming Language. `http://www.ruby-lang.org/`.

[SB05] Athanasios Staikopoulos and Behzad Bordbar. A Comparative Study of Metamodel Integration and Interoperability in UML and Web Services. *Lecture Notes in Computer Science*, 3748:145–159, 2005.

[SBP99] Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic. Dsl implementation using staging and monads. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 81–94, New York, NY, USA, 1999. ACM.

[SBP+09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, Rochard C. Gronback, and Mike Milinkovich. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition, 2009.

[Sca] The Scala Programming Language. `http://www.scala-lang.org/index.html`.

[SCD03] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust Multilingual Parsing using Island Grammars. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research*, pages 266–278. IBM Press, 2003.

[SCK04] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising embedded dsls using template haskell. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 201–211. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30175-2_10.

[She04a] Tim Sheard. Evolving Domain Specific Languages. `http://web.cecs.pdx.edu/~sheard/proposals/EvolvDsl.ps`, 2004.

[She04b] Tim Sheard. Languages of the Future. *SIGPLAN Not.*, 39(12):119–132, 2004.

[SHH09] Hans Schippers, Michael Haupt, and Robert Hirschfeld. An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC'09)*, pages 1944–1951, New York, NY, USA, 2009. ACM.

[SN05] J.E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann Pub, 2005.

[Ste99] G.L. Steele. Growing a Language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[Sul01] G.T. Sullivan. Aspect-Oriented Programming using Reflection and Metaobject Protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[Tan04] É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, Université de Nantes, France, 2004.

[Tan06a] É. Tanter. An Extensible Kernel Language for AOP. In *AOSD Workshop on Open and Dynamic Aspect Languages*, 2006.

[Tan06b] E. Tanter. Aspects of Composition in the Reflex AOP Kernel. *LNCS*, 4089:98, 2006.

[Tan08] Éric Tanter. Expressive Scoping of Dynamically-Deployed Aspects. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Sevelopment (AOSD'08)*, pages 168–179, New York, NY, USA, 2008. ACM.

[Tan09] Éric Tanter. Beyond static and dynamic scope. *SIGPLAN Not.*, 44(12):3–14, 2009.

[TCL+00] S. Thibault, C. Consel, J.L. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[TFH09] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2009.

[TN05] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language aop. *Generative Programming and Component Engineering*, 3676:173–188, 2005. 10.1007/11561347_13.

[Tra08] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):1–40, 2008.

[vdBSV97] M. van den Brand, M. Sellink, and C. Verhoef. Obtaining a Cobol Grammar from Legacy Code for Reengineering Purposes. *Proceedings of the Second International Workshop on Theory and Practice of Algebraic Specifications*, 1997.

[vdBSVV02] Mark van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 21–44. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45937-5_12.

[vdBvDK+96] M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van Der Meulen. Industrial Applications of ASF+SDF. *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology (AMAST'96)*, 1101:9–18, 1996.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[vG00] J. van Gurp. *Variability in Software Systems: The Key to Software Reuse*. PhD thesis, Dept. of Software Engineering & Computer Science, Blekinge Institute of Technology, Karlskrona, Sweden, 2000.

[VGBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[Vis97a] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, University of Amsterdam, The Netherlands, 1997.

[Vis97b] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Universiteit van Amsterdam, The Netherlands, 1997.

[Vis04] Eelco Visser. Program Transformation with Stratego/XT. *Domain-Specific Program Generation*, 3016:315–349, 2004. 10.1007/978-3-540-25935-0_13.

[Vis08] E. Visser. WebDSL: A case study in domain-specific language engineering. *Generative and Transformational Techniques in Software Engineering II*, 5235:291–373, 2008.

[VWBGK08] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116, 2008.

[VWBH06] E. Van Wyk, D. Bodin, and P. Huntington. Adding syntax and static analysis to libraries via extensible compilers and language extensions. In *2nd International Workshop on Library-Centric Software Design (LCSD'06)*, page 35, New York, NY, USA, 2006. ACM.

[W3C] W3C. XSL transformations (XSLT) version 2.0. `http://www.w3.org/TR/xslt20/`.

[W3C06] W3C. XML 1.1 (Second Edition), W3C Recommendation, (Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Franï£ios Yergeau, John Cowan, ed.). `http://www.w3.org/TR/2006/REC-xml11-20060816/`, August 2006.

[W3C07] W3C. XML Path Language (XPath) 2.0, W3C Recommendation, (Anders Berglund and Scott Boag and Don Chamberlin and Mary F. Fernández and Michael Kay and Jonathan Robie and Jérôme Siméon, ed.). `http://www.w3.org/TR/xpath20/`, January 2007.

[Wad90] Philip Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming (LFP'90)*, pages 61–78, 1990.

[Wad95] Philip Wadler. Monads for Functional Programming. *Advanced Functional Programming*, 925:24–52, 1995. 10.1007/3-540-59451-5_2.

[WGM09] Hui Wu, Jeff Gray, and Marjan Mernik. Unit Testing for Domain-Specific Languages. *Domain-Specific Languages*, 5658:125–147, 2009.

[WKBS07] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. Attribute Grammar-Based Language Extensions for Java. *ECOOP 2007–Object-Oriented Programming*, 4609:575–599, 2007.

[WTZ10] Christian Wende, Nils Thieme, and Steffen Zschaler. A role-based approach towards modular language engineering. In Mark van den Brand, Dragan GaŽevic, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 254–273. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-12107-4_19.

[Zha06] G. Zhang. Towards Aspect-Oriented State Machines. In *Asian Workshop on Aspect-Oriented Software Development (AOAsia'06)*, 2006.