Designing Syntax Embeddings and Assimilations for Language Libraries

Martin Bravenboer and Eelco Visser

Software Engineering Research Group, Delft University of Technology,
The Netherlands
martin.bravenboer@gmail.com, visser@acm.org

Abstract. *Language libraries* extend regular libraries with domain-specific notation. More precisely, a language library is a combination of a domain-specific language *embedded* in the general-purpose host language, a regular library implementing the underlying functionality, and an *assimilation* transformation that maps embedded DSL fragments to host language code. While the basic architecture for realizing language libraries is the same for all applications, there are many design choices to be made in the design of a particular combination of library, guest language syntax, host language, and assimilation. In this paper, we give an overview of the design space for syntax embeddings and assimilations for the realization of language libraries.

1 Introduction

Software libraries provide reusable data structures and functionality through the built-in abstraction facilities of a programming language. While functionally complete, the interface through regular function or method calls is often not appropriate for efficiently and understandably expressing programs in the domain of the library. *Language libraries* extend regular libraries with domain-specific notation. More precisely, a language library is a combination of a domain-specific language *embedded* in a general-purpose host language, a regular library for the underlying functionality, and an *assimilation* transformation that maps embedded DSL fragments to host language code.

In recent years case studies of language libraries have been conducted for a number of host languages and types of applications, including concrete object syntax for meta-programming [3,24], embedding of domain-specific languages in general purpose languages (MetaBorg) [11,6], and syntax embeddings for preventing injection attacks (StringBorg) [7]. While there is a common architecture underlying all these language libraries, there are many design choices to be made in filling in the parameters to the architecture. For example, a recent innovation is type-based disambiguation of syntax embeddings [26,10], which uses the type system of the host language to disambiguate quoted code fragments, thus allowing a more lightweight syntax.

In this paper, we present an overview of the design space for syntax embeddings and assimilations for the realization of language libraries. The contribution of this paper is an overview of the state-of-the-art, providing insight in the design space, and research questions in language library realization, in particular, the identification of research issues for realizing an independently extensible language library framework. In the next

H. Giese (Ed.): MoDELS 2007 Workshops, LNCS 5002, pp. 34–46, 2008.

section we give an overview of the various types of applications of language libraries illustrated with examples. In the remaining sections we discuss technical considerations and trade-offs in the realization of language libraries.

2 Applications of Language Libraries

In this section we consider different types of applications, which are characterized by the target of assimilation. We distinguish four types of language libraries; libraries for transformation, generation, string generation, and DSL embedding. We consider each of these categories in turn and show typical examples. Figure 1 gives an overview of some libraries that we have realized.

Transformation of structured data is typically used for program transformation, but also for transformation of other structured data, such as XML documents. Direct manipulation of the structures through their API can lead to tedious coding in which the natural structures are hard to recognize. Syntax embeddings can be used to provide concrete syntax for patterns used to match and construct code fragments [24]. The target of assimilation in these applications is an API for analysing and constructing abstract syntax trees. For example, consider the following Stratego rewrite rule that defines the desugaring of the `sum` construct in terms of a `for` loop with an auxiliary (fresh) variable:

	host language			
guest lang.	Stratego	Java	Perl	PHP
Stratego	T			
Tiger	T			
ATerm	T	T		
C	G(T)			
Java	G(T)	G		
XML	G(T)	G		
SQL		S	S	S
Shell		S	S	S
XPath		S	S	S
Swul		D		
Regex		D		

Fig. 1. Examples of realized embeddings with host languages in columns and embedded languages in rows. The letters indicate the type of embedding with G = generation, T = transformation, S = string generation, D = DSL embedding.

```
DefSum :
  |[ sum x = e1 to e2 ( e3 ) ]| ->
  |[ let var y := 0
      in (for x := e1 to e2
           do y := y + e3); y end ]|
  where y := <newname> x
```

The terms between `|[` and `]|` are quotations of Tiger code patterns that are used both to pattern match *and* to compose code. For example, the left-hand side `|[ sum x = e1 to e2 (e3) ]|` of the rewrite rule is assimilated to the term pattern `Sum(Id(x), e1, e2, e3)`, where `x`, `e1`, `e2`, and `e3` are *meta-variables*, i.e. variables that match subterms when the rule is applied. (`newname` is used to create a fresh variable in order to avoid accidental capture of free variables.)

A similar idea can be used with Java as host language. While Stratego natively supports terms for representing abstract syntax trees, Java requires such structures to be defined using objects. A syntax embedding of terms in Java (JavaATerm) can be used to make analysis and construction of term structures in Java programs easier. For example, the following is a code fragment from a transformation from an ATerm representation of bytecode files to a BCEL representation of bytecode:

```
private void addInstructions(ATerm code) {
  ATerm optlocals = null, optstack = null;
```

```
ATerm is = null, excs = null;
match code with
  Code(MaxStack(optstack), MaxLocals(optlocals)
  , Instructions([...] and is)
  , ExceptionTable([...] and excs)
  , Attributes(_));
```

The embedding provides (among other things) concrete syntax for term patterns and the `match <expr> with <term>` construct, which is assimilated to statements that implement the matching by analyzing the ATerm argument against the declared pattern.

An important requirement for the use of syntax embeddings in transformations is that the structure of the quoted pattern coincides with the structure of the program to which it is applied. This does not hold in scenarios where the abstract syntax tree of a program is heavily analyzed, modified, and/or attributed before being transformed. For example, abstract syntax trees for C and Java require static semantic analysis before they can be properly transformed.

Code generation involves the composition of programs from templates based on input such as a DSL program or user input. Construction of large pieces of code using string concatenation is tedious and error-prone. Templates are not checked statically for well-formedness and meta characters need to be escaped. Furthermore, a textual representation does not allow further processing. Use of an API provides a structured representation, but is not suitable for encoding (large) templates. Syntax embeddings allow encoding templates in the concrete syntax of the language, but at the same time producing structured code. Embedded code fragments are assimilated to API calls for *constructing* structured data (such as ASTs). The API does not need to support *transformation* of patterns derived from concrete syntax. For example, the back-end of the Stratego compiler uses rules such as

```
TranslateStrat(|S,F) :
  |[ s1 < s2 + s3 ]| ->
  stm|[ { ATerm ~id:x = t;
          ~stm:<translate-strat(|Next,F')>s1
          ~stm:<translate-strat(|S,F)>s2
          ~id:F' : t = ~id:x;
          ~stm:<translate-strat(|S,F)>s3 } ]|
  where x := <new>; F' := <new>
```

to implement schemes for translating Stratego to C. In this example *two* languages are embedded in the host language; the left-hand side of the rule is a Stratego code fragment, while the right-hand side is a C code fragment. The right-hand side template uses recursive calls to generate code for the subterms of the left-hand side pattern. The results are integrated by *anti-quotations* such as `~stm:t` and `~id:t` that convert a Stratego term into the right syntactic sort. Note that quotations and antiquotations are *tagged* with syntactic sorts such as `stm|[` and `~id:`, this is necessary to avoid ambiguities in parsing.

The same technique is used for other host languages. For example, the following fragment shows a Java method generating Java code represented using the Eclipse JDT.

```
public CompilationUnit run(ClassDescriptor cd) {
  CompilationUnit result = |[
```

```
    import java.util.*;
    public class #[ cd.getName() ] {
      #[ attributes(cd) ]
    }
  ]|;
  return result;
}
```

Note that unlike in the Stratego case, the (anti)quotations are not tagged; this information can be deduced from the types of the host language (e.g., `CompilationUnit`) [10].

String generation is commonly used in libraries supporting domain-specific languages such as XML, XPath, regular expressions, and SQL to provide an interface that uses character strings to communicate programs in these language. For example, a method `execQuery` from an SQL library might take a string containing an SQL query as argument, such as

```
execQuery("SELECT * FROM Users where username = \'" + name + "\'");
```

Insertion of run-time data (e.g., user input) is done by composing queries using string concatenation from constant parts and dynamic parts provided by variables. The approach suffers from a number of problems. First, strings are not statically guaranteed to be well-formed, which may lead to run-time errors. Second the approach requires escaping of meta-symbols (such as the quotes in the example above), which can lead to tedious and obfuscated code. Worst of all, the approach leads to software that is prone to *injection attacks*, where ill-formed user input may lead to breakdown or security compromises. For example, insertion of the string `' OR 1=1--` in the query above would lead to dumping a list of all users.

These problems can be avoided by using syntax embeddings instead of string concatenation [7]. By quoting a query, as in the following code fragment

```
SQL.QueryExpr q = <| SELECT * FROM Users WHERE username = $str{arg} |>;
execQuery(q);
```

the query is syntactically checked at compile-time, no spurious escaping is needed, and insertion of run-time data is guaranteed to be properly escaped to avoid injection attacks. Embedded queries are assimilated to calls to methods in a *syntax API*, which provides for each production in the syntax definition a corresponding method. This ensures that well-formedness of queries is verified by the type system of the host language.

Domain-specific language (DSL) embedding is concerned with providing better notation for programming in a certain domain, and are typically designed around an existing library. The DSL abstracts over normal usage of the library, and assimilation is to sequences of library calls. For example, JavaRegExp is a DSL built on top of the Java regular expression library. In the first place it provides a syntax for quoting regular expressions without spurious escaping, similar to the string generation examples above. Building on this basic embedding, JavaRegExp provides a DSL for defining and combining string rewrite rules in Java. For example, the following code fragment defines several string rewrite rules for escaping HTML special characters, their composition with a choice operator `<+`, and the application of the rules to the contents of a string in variable `input`:

```
regex amp = [/ & /] -> [/ &amp; /];
regex lt  = [/ < /] -> [/ &lt;  /];
regex gt  = [/ > /] -> [/ &gt;  /];
regex escape = amp <+ lt <+ gt;
input ~= all(escape);
```

Another example in this category is JavaSwul, an embedding in Java of a dedicated language for creating Swing user-inteface widgets, following the hierarchical structure of the class hierarchy of Swing [11,6].

3 Syntax Embedding and Assimilation

Language libraries are realized by means of *syntax embeddings and assimilations* as illustrated by the architecture diagram in Figure 2. An implementation typically consists of four components, *i.e.* a parser, typechecker, assimilator, and pretty-printer. Together these components transform programs in the extended language to programs in the host language only. Each of the components is parameterized with data that are specific for the syntax embedding at hand. The *parser* is parameterized with the syntax of the extended language. This requires syntax definitions for the *host language* and the *guest language*, and a definition of how guest language fragments are inserted in host language programs. The parser converts a textual representation of a program in the combined language to a tree structure that is suitable for further processing. *Type rules* extend the *typechecker* of the host language to check extended programs. This component is optional. Having an extensible typechecker avoids type error messages expressed in terms of assimilated programs. The *assimilator* transforms embedded guest language fragments to an implementation in the host language. The assimilator is parameterized with a set of *assimilation rules* that define the translation schemes for the guest language. For certain applications the assimilation rules may be generic in the guest language. A *pretty-printer* converts the tree structure produced by the assimilator to text, which can be fed to a host language compiler or interpreter. Pretty-printing can be avoided if transformations are expressed directly on parse trees, rather than abstract syntax trees. Another option is not to produce a textual representation of the assimilated program at all, but instead link assimilation into the host language compiler. This is done for example in Stratego, where assimilation of concrete object syntax is built into the compiler [24]. Finally, the assimilator may generate code that makes calls to an *API corresponding to the guest language* (the 'run-time system' for the embedding).

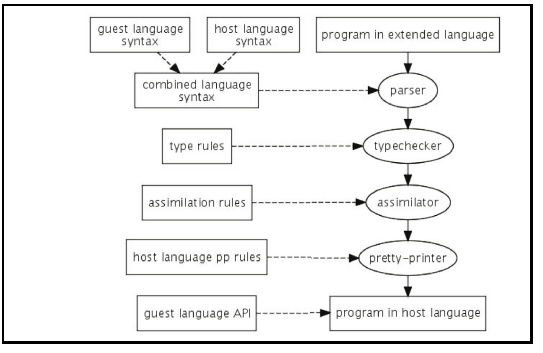

Fig. 2. Components for syntax embedding and assimilation. Solid arrows denote data-flow, dashed arrows parameters of tools in the chain.

3.1 Syntax Embedding

A syntax embedding is an extension of the syntax of the host language with the syntax of a guest language. Such an extension is achieved by an extension of the grammar of the host language, which introduces the language constructs of the guest language at specific places in the host language. To illustrate the basic principles of syntax embeddings, we use an embedding of SQL in PHP using SDF [23], a modular syntax definition formalism for defining the lexical as well as context-free syntax of language in a single formalism. Figure 3 shows the syntax definition for a tiny subset of SQL, the guest language of this example. We omit the syntax definition of the host language. The module `TinySQL` defines the lexical and context-free syntax of a stylized subset of SQL in reversed EBNF notation, namely simple queries 3 with where clauses 4, equivalence expressions 5, identifiers 1 and character strings with escaped quotes 2.

```
module TinySQL exports
lexical syntax
  [A-Za-z]+ -> Id                                  1
  [A-Za-z0-9\ \"\-\;] -> Char
  "\'" ("\'\'" | Char)* "\'" -> CharString         2
context-free syntax
  "SELECT" Id* "FROM" Id Where?-> Query            3
  "WHERE" Expr -> Where                            4
  Expr "=" Expr -> Expr {left}                     5
  CharString -> Expr
  Id -> Expr
```

Fig. 3. Syntax definition for a tiny subset of SQL

The syntax of TinySQL is embedded in PHP by creating a new SDF module (Figure 4) that imports the syntax definitions of PHP as well as TinySQL 6 and defines how the languages are combined, *i.e.* where TinySQL can be used in PHP and vice versa. The productions of this module use *parameterized non-terminals*, *e.g.* `Expr[[PHP]]` and `Expr[[SQL]]`, which are used to indicate the language of the non-terminal. The first production 7 of `SQL-in-PHP` defines the *quotation* of SQL queries, i.e. that they can be used as PHP expressions between the quotation tokens `<|` and `|>`. The second 8 and third 9 productions define the *anti-quotations* of this embedding, which allow an SQL expression or character string to be constructed by an arbitrary PHP expression. In this way, queries can be composed dynamically. From this combined syntax definition a parser is generated, which is used to parse PHP programs that use the SQL syntax extension. After that, the resulting parse tree or abstract syntax tree is transformed to a plain PHP program by an assimilation.

```
module SQL-in-PHP imports PHP TinySQL 6 exports
context-free syntax
  "<|"    Query[[SQL]] "|>" -> Expr[[PHP]]          7
  "${"    Expr[[PHP]]  "}"  -> Expr[[SQL]]          8
  "$str{" Expr[[PHP]]  "}"  -> CharString[[SQL]]    9
```

Fig. 4. Embedding of syntax for SQL in syntax for Java

3.2 Assimilation

The assimilation phase provides the actual implementation of the embedded syntax of the domain-specific language. In this phase the embedded language constructs are

removed from the source program by a translation to the host language, using any necessary translation scheme. The implementation of the assimilation phase largely depends on the application for which the embedded language is intended. The complexity of the implementation depends on the particular combination of the embedded language and the host language. The design of the assimilation is influenced by the requirements to make embeddings (a) easy to understand, (b) composable, and (c) analyzable.

Assimilation rules are the basic transformation steps of the assimilation that have a pattern of guest code at their left-hand side and produce a pattern of host code at their right-hand side. In our examples the assimilation rules are implemented in the Stratego program transformation language. Figure 5 illustrates two assimilation rules for the embedding of Java in Java. The first rule [10] assimilates an array type by generating invocations of the Eclipse JDT API for the representation of Java programs in Java. The second rule [11] assimilates a field access by creating a `FieldAccess` object and invoking some methods on it for initialization. The assimilation rules are applied by an *assimilation strategy* [13], which traverses the program and applies the rules where necessary. Most assimilation strategies have the same structure: they traverse the program topdown and apply the assimilation rules if a quotation is found. If an anti-quotation is found [12], then assimilation is stopped and the assimilation strategy is invoked recursively.

```
Assimilate : 10
  ArrayType(type) -> e|[ _ast.newArrayType(e) ]|
  where e := <Assimilate> type

Assimilate : 11
  Field(e,y) -> |[
    {| FieldAccess x = _ast.newFieldAccess() ;
       x.setExpression(e1); x.setName(e2);
     | x |}
  ]|
  where <newname> "expr" => x;
    [e1,e2] := <map(Assimilate)> [e,y]

Assimilate = 12
  ?AntiQuote(<assimilate-strat>)

assimilate-strat = 13
  alltd(?Quote(<Assimilate>))
```

Fig. 5. Assimilation of Java in Java

4 Design Issues

There are many variation points in the realization of syntax embedding and assimilation for a language library. In this section we give an overview of the main issues. An in depth discussion of the issues and the range of solutions and their relative advantages and disadvantages is beyond the scope of this paper.

4.1 Syntax Embedding

The grammar formalism that is used for defining the syntax embeddings should allow **modular extension of grammars**. A modular grammar formalism implies support for

the **full class of context-free grammars**, since that is the only class that is closed under composition. In particular, grammar classes such as LL and LR, which are supported by conventional parser generators are not adequate. Similarly, the lexical syntax is usually defined using a collection of regular expressions. However, the class of regular grammars is also not closed under composition. Solutions to **composition of lexical syntax** that are used in practice are: (1) Ignoring the problem, which entails that the lexical syntax of host and guest language are merged, e.g. keywords of one become keywords of the other. Used in Bali, from the pioneering AHEAD/JTS tool set [3] (see [11] for a discussion). (2) Use lexical states to distinghuish the context in which a program fragment should be interpreted. In language libraries for program generation such as Meta-AspectJ [26] and JavaJava [10] with explicit quotation symbols this is a possible solution (3) Control the state of the lexical analyzer from the parser. While this a major complication of the interface between the scanner and the parser and seems to be rather unpopular in practice, it has recently been applied in the Silver extensible compiler system [22]. (4) Let the scanner produce all possible interpretations of tokens. This is only possible if the token boundaries of the host and guest languages are exactly the same, which is often not the case. (5) Integrate lexical syntax and context-free syntax in the same formalism as is realized in the syntax definition formalism SDF, which implemented by using scannerless parsing This means that a separate lexical analysis phase is omitted and the parser operates directly on the individual characters of a source file. This is the only solution that can gracefully deal with combinations of languages.

Requiring support for the full class of context-free grammars reduces the number of **parsing algorithms** that can be used. Currently, the best studied algorithms for parsing possibly ambiguous context-free grammars are generalized-LR [21,18] and Earley [12] parsing. Scannerless parsing [19], which is an important feature for syntax embedding, has been integrated with generalized-LR in the implementation of SDF [23]. Known issues with (S)GLR are the poor quality of error messages and the absence of error recovery. Packrat parsing [13] is another candidate for parsing syntax embeddings. Packrat parsers are used to parse languages defined by *parsing expression grammars*, which are closed under composition, intersection and complement. Scannerless packrat parsers are available and have been shown to perform very well [14] compared to current GLR parser implementations. Unfortunately, packrat parsers are not able to produce all possible alternatives for ambiguities.

When the basic requirements for modularly combining grammars are met, there are additional **modularity features** a grammar formalism should have to support syntax embeddings. A notion of namespaces is needed to manage the scope of definitions in grammars to be combined. The host and guest language grammar may use the same non-terminal, which will become a single syntactic category when combined. To keep such non-terminals separated and to control exactly which non-terminals are identified, a namespace or renaming mechanism is necessary. A related issue is that the set of identifiers that should be considered as reserved keywords depends on the context. For example, `class` is a keyword in Java, but not in SQL. Also the rules for layout (whitespace and comments) are different depending on the context. For example, `//` is a comment symbol in Java, but an operator in XPath. In a declarative syntax definition

of AspectJ [8], the notion of **grammar mixins** was introduced to deal with these issues (except for modular layout). The parameterized sorts and modules of SDF, along with a mixin generator, were used to implement grammar mixins. A proper language design for grammar mixins is needed.

In code and string generation, **quotations and anti-quotations** are often highly ambiguous if the quotations of the various guest non-terminals all use the same syntax. There are three solutions to this problem. (1) The number of quotations can be restricted. For example, if the embedding allows the quotation of a list of statements as well as a single one, then a quotation of a single statement will always be ambiguous. In some applications, having only a quotation for the list of statements might be sufficient. (2) The quotations and anti-quotations can be explicitly tagged with their syntactic type, basically introducing different quotation symbols for every quotable non-terminal. This solves the ambiguity problem, but the user now needs to know when and how to use these tags. Furthermore, the tags obfuscate the code and often feel redundant to the user. In some cases explicit tagging can be made less unattractive by using keywords from the guest language. For example, the anti-quotation of an optional where clause of an SQL query can be tagged using `WHERE?`, which looks like the keyword `WHERE` from SQL.

```
"WHERE" "?" "${" E "}" -> Where⟦SQLCtx⟧?
```

(3) The ambiguities can be preserved by parsing the ambiguous source file to a parse forest. The ambiguity can then be dealt with at a later phase (for instance, by leveraging the type system of the host language [10]), or can even be ignored if the exact representation is irrelevant. For the user this is by far the most attractive solution, since the embedding is not restricted and no knowledge about the ambiguities is required.

Having a single extension of a host language available is useful, but many applications require **multiple extensions** to be available in a single source file. For example, in web applications, the main application domain of StringBorg, XML, SQL, Javascript, and Shell, are often used together. Extensions should preferably not be deployed as closed extensions of the host language, but rather as separate plugins that can be selected by the user and combined *on the fly* by the system. This requires **independent extensibility** of the host language, that is truly modular implementations of language libraries.

4.2 Assimilation

The **scope** of a transformation [25] indicates the parts of the source and target program that are involved in the transformation. Scope has a major impact on the complexity of assimilations. We distinguish the following types of scopes. (1) *Local-to-local* assimilations map guest code fragments directly to host code fragments, which are at the exact same place in the abstract syntax tree as the original guest code. Local-to-local rules are easy to implement, since the rules are all independent and do not influence the assimilation strategy. (2) *Local-to-global* assimilations do not just locally replace guest code fragments, but in other places in the abstract syntax tree as well. These assimilations are typically needed when fragments to be generated cannot be expressed locally. For example, executing statements, declaring new methods, or introducing import declarations. (3) *Global-to-local* assimilations need context information from the

original program for a local assimilation, such as the current package, class, method, or even type information. (4) *Global-to-global* assimilations need context information from the source program and also produce global host code. Dynamic rewrite rules [9] are typically used to implement context-sensitive transformations, that is all but those with local-to-local scope.

To avoid name capture, assimilation rules should be **hygienic**, that is, generated names should all be unique. This can be achieved easily with a gensym-like mechanism that guarantees generation of unique names.

If the embedding is part of a family of embeddings that all implement a similar assimilation, the assimilation may be **generic** in the guest language. For example, the representation of object programs in Stratego is directly based on the syntax definition of the object language, thus the transformation from the guest code to Stratego is implemented generically [24]. In some cases the assimilation can even largely be *host language* independent [7].

If the host language can be typechecked statically, then it is useful to **typecheck before assimilation**, to keep the error reports as close to the original source as possible. This requires extension of the typechecker of the host language with type rules for the guest language.

5 Related Work

Syntax Embedding. Macro systems usually restrict the syntax that can be introduced. For an extensive review of the relationship to JSE [1], Maya [2], Metafront [5], JTS [3], and camlp4 [20] we refer to [11] and [10]. There are a number of parser generators that could be applied for the parsing of syntax embeddings. Harmonia's Blender [4] and Cardelli's extensible syntax have been discussed in detail in [11]. Packrat parsing [13,14] has been discussed in Section 4.1. The Polyglot parser generator [17] supports modular adaptation of LR-grammars, which has been discussed in [7].

Furthermore, there is a wide range of applications that apply a form of syntax embedding. In these particular applications (Template-Haskell, MetaOCaml, Meta-AspectJ, SafeGen, etc.) the parsing problem is often reduced because the host language is embedded in itself or development time for parsing this combination of languages is acceptable. Hence, no general method for language libraries is required, though they could profit from such a facility. Recently, support for quoting expressions has been introduced in C#. C# does not introduce a syntax for quotation, but infers quotations from the type of variables and parameters. One of the main purposes of this facility is to use C# expressions to express query expressions, *e.g.* SQL, without introducing syntax (see also [7]).

Assimilation. Macro systems usually only allow a straightforward local-to-local mapping from the introduced syntax to the host language. Our assimilation rules and strategies allow local-to-global and global-to-local transformations, which is somewhat related to the desire for macros to reach out and touch somewhere [15], one of the ideas leading to AspectJ. Macro systems that allow context-free grammars as macro

arguments often provide more advanced facilities for transformations. For example, Metafront [5] associates transformations to all productions and additionally guarantees termination of the transformation.

Open Compilers. Open compilers such as Polyglot [17] are not designed to facilitate DSL embedding in particular. Instead, they are designed for the introduction of new language features that invade the language, such as new types, optimizations, concurrency features, etc. The requirements for a system that supports the implementation of syntax embedding and assimilation by non-compiler experts are rather different. More experience with extensions is needed to give definite answers on the requirements.

Methodology. In [16] the authors state that work is needed on DSL development methodologies, since the use of a DSL can provide major benefits, but the development of DSLs often raises many questions without clear answers or resources on the right decisions to make. Though our work chooses a particular approach to DSL development, our analysis of the design space in this area contributes to the discussion on methodologies. Also, a major focus of our work is to make the implementation of extensions as easy as possible by not imposing technical constraints on the syntax embedding and assimilation.

Domain-Specific Embedded Languages. The term domain-specific embedded (or internal) language (DSEL/EDSL) is often used for a DSL defined within the host language, that is, without syntactically extending the host language. DSELs are popular in languages that have a flexible syntax, such as Haskell (user-definable infix operators) and Ruby. In the case of Ruby, there is extensive support for introspection, intercession, code generation, evaluation and bindings, which reduces the need for interpreting the DSL. The heavy use of run-time code generation is largely cultural (since similar functionality is available in languages such as Perl and Python) and due to seemingly irrelevant details, such as multi-line string literals. Considering the syntax, the main disadvantages of embedded domain-specific language approaches is that the syntax cannot be clearly defined separately, but has to be crafted carefully, based on the constraints imposed by the host language.

Acknowledgments. This research was supported by NWO/JACQUARD projects 638.001.201, *TraCE: Transparent Configuration Environments*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

1. Bachrach, J., Playford, K.: The Java syntactic extender (JSE). In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001), pp. 31–42. ACM, New York (2001)
2. Baker, J., Hsieh, W.: Maya: multiple-dispatch syntax extension in java. In: Programming Language Design and Implementation (PLDI 2002), pp. 270–281. ACM, New York (2002)
3. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: tools for implementing domain-specific languages. In: International Conference on Software Reuse (ICSR 1998), pp. 143–153. IEEE, Los Alamitos (1998)

4. Begel, A., Graham, S.L.: Language analysis and tools for input stream ambiguities. In: Language Descriptions, Tools and Applications (LDTA 2004). ENTCS, Elsevier, Amsterdam (April 2004)
5. Brabrand, C., Vanggaard, M., Schwartzbach, M.I.: The metafront system: Extensible parsing and transformation. In: Language Descriptions, Tools and Applications (LDTA 2003), ACM, New York (April 2003)
6. Bravenboer, M., de Groot, R., Visser, E.: MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 297–311. Springer, Heidelberg (2006)
7. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In: Lawall, J. (ed.) Generative Programming and Component Engineering (GPCE 2007), pp. 3–12. ACM, New York (October 2007)
8. Bravenboer, M., Tanter, E., Visser, E.: Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-LR parsing. In: Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2006), ACM, New York (October 2006)
9. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fundamenta Informaticae 69(1–2), 123–178 (2006)
10. Bravenboer, M., Vermaas, R., Vinju, J.J., Visser, E.: Generalized type-based disambiguation of meta programs with concrete object syntax. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 157–172. Springer, Heidelberg (2005)
11. Bravenboer, M., Visser, E.: Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In: Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2004), pp. 365–383. ACM, New York (October 2004)
12. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM 13(2), 94–102 (1970)
13. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In: International Conference on Functional Programming (ICFP 2002), pp. 36–47. ACM, New York (2002)
14. Grimm, R.: Better extensibility through modular syntax. In: Cook, W.R. (ed.) Programming Language Design and Implementation (PLDI 2006), ACM, New York (June 2006)
15. Kiczales, G., Lamping, J.: L.H.R., Jr., Ruf, E.: Macros that reach out and touch somewhere. Technical report, Xerox Corporation (December 1991)
16. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
17. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for Java. In: Hedin, G. (ed.) CC 2003 and ETAPS 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
18. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam (1992)
19. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) parsing of programming languages. ACM SIGPLAN Notices 24(7), 170–178 (1989); In: PLDI 1989 (1989)
20. de Rauglaudre, D.: Camlp4 reference manual (September 2003)
21. Tomita, M.: Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Dordrecht (1985)
22. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Lawall, J. (ed.) Generative Programming and Component Engineering (GPCE 2007), pp. 63–72. ACM, New York (October 2007)
23. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)

24. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)
25. van Wijngaarden, J., Visser, E.: Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University (May 2003)
26. Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ programs with Meta-AspectJ. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 1–19. Springer, Heidelberg (2004)