Growing a Language Environment with Editor Libraries

Sebastian Erdweg*   Lennart C. L. Kats†   Tillmann Rendel*
Christian Kästner*   Klaus Ostermann*   Eelco Visser†

* University of Marburg
† Delft University of Technology

Abstract

Large software projects consist of code written in a multitude of different (possibly domain-specific) languages, which are often deeply interspersed even in single files. While many proposals exist on how to integrate languages semantically and syntactically, the question of how to support this scenario in integrated development environments (IDEs) remains open: How can standard IDE services, such as syntax highlighting, outlining, or reference resolving, be provided in an extensible and compositional way, such that an open mix of languages is supported in a single file?

Based on our library-based syntactic extension language for Java, SugarJ, we propose to make IDEs extensible by organizing editor services in *editor libraries*. Editor libraries are libraries written in the object language, SugarJ, and hence activated and composed through regular import statements on a file-by-file basis. We have implemented an IDE for editor libraries on top of SugarJ and the Eclipse-based Spoofax language workbench. We have validated editor libraries by evolving this IDE into a fully-fledged and schema-aware XML editor as well as an extensible LaTeX editor, which we used for writing this paper.

Categories and Subject Descriptors   D.2.6 [*Programming Environments*];   D.2.13 [*Reusable Software*];   D.3.2 [*Language Classifications*]: Extensible languages

General Terms   Languages

Keywords   language extensibility, library, DSL embedding, language workbench

1. Introduction

Programming language extensibility is an old research topic that has gained new relevance by the trend toward domain-specific languages and the vision of language-oriented programming [6, 32]. Researchers have proposed a variety of different approaches to extend the syntax and semantics of languages and to embed languages in other languages, such as libraries [9, 14], extensible compilers [8, 21, 28], macro systems [1, 3, 24, 25], and meta-object protocols [22, 23]. However, while languages themselves have gained flexibility, tool support in the form of integrated development environments (IDEs) cannot keep up with the rapid development and composition of new languages.
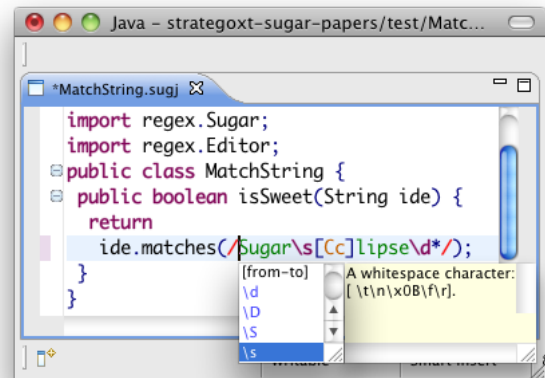
Figure 1. regex.Sugar provides a syntactic extension for regular expressions; regex.Editor provides the according IDE extension.

IDEs assist programmers, who spend a significant amount of time reading, navigating, adapting and writing source code. They provide *editor services* that improve a program's layout and support programmers in performing changes to the code, including syntax highlighting, code folding, outlining, reference resolving, error marking, quick fix proposals, code completion, and many more. The quality of IDE support for a language is hence a significant factor for the productivity of developers in that language. It is therefore desirable to provide the same level of tool support for extended and domain-specific languages that programmers are familiar with from mainstream programming languages.

However, as our own and the experience of others show, developing tool support for a new or extended language requires significant effort [5, 16, 20]. Although there are several advances to generate tool support from declarative specifications [7, 17], generation has to be repeated for every combination of language extensions because the generated editor services neither compose nor grow with the language.

Composable and growable editor services are especially important in the context of growable languages that support flexible and composable extensions, e.g., for the embedding of multiple domain-specific languages. In prior work, we developed *SugarJ*, a variant of Java which is extensible via sugar libraries [9]. A sugar library can export, in addition to ordinary types and methods, a syntactic extension and a transformation from the extended syntax back into the syntax of the base language. Sugar libraries are imported via the usual import mechanism of the base language. Multiple syntactic extensions can be composed by importing them

into the same file, allowing a local mix of multiple embedded languages.

In this paper, we propose *editor libraries* to further generalize library-based extensibility towards IDEs. Editor libraries compose: When multiple languages are mixed within the same file (such as XML, SQL and regular expressions within Java), we import and thereby combine all corresponding editor services. Editor libraries (as other libraries) are self-applicable, that is, editor libraries can be used to develop other editor libraries. Furthermore, editor libraries encourage a generative approach through *staging*: We generate editor services from high-level specifications (yet another domain-specific language) at one stage and use the generated services at a later stage. Staging enables the coordination of editor services that span several source files or languages.

We have developed an Eclipse-based IDE with support for editor libraries called *Sugarclipse*. For each file, Sugarclipse considers all editor libraries in scope, interprets the associated editor services and presents the decorated source code and editing facilities to the programmer. Sugarclipse is based on the *Spoofax* language workbench [17], which supports the generation and dynamic reloading of Eclipse-based language-specific editors from declarative editor configurations. In Figure 1, we illustrate an example usage of Sugarclipse: The import of regex.Sugar activates a syntactic extension for regular expressions, which integrates regular expression syntax into the surrounding Java syntax (instead of the usual string encoding). The import of the editor library regex.Editor enables corresponding editor services for regular expressions such as syntax coloring and code completion. Sugarclipse automatically composes the editor services of the host language, here Java, and of the extension, here regular expressions, to provide uniform IDE support to the programmer. While in Sugarclipse and in this paper, we focus on editor libraries for SugarJ, the concept of editor libraries is similarly useful for embedded languages in syntactically less flexible languages (cf. Section 7).

With several case studies, we demonstrate the practicality of editor libraries and the power of their composition. Beyond small editor libraries such as regular expressions illustrated above, we implemented fully-fledged editor libraries for XML (including XML Schema) and Latex. We used the latter for writing this paper.

In summary, we make the following contributions:

- We introduce the novel concept of *editor libraries* for organizing IDE extensions in libraries of the object language, in particular, to provide IDE support for embedded DSLs.

- Editor libraries are activated using the host language's standard import mechanism, and editor libraries *compose* to support multiple DSLs and the base language in a single file.

- We describe a pattern of *editor library staging* to generate editor services from high-level specifications and to coordinate editor services between several source files or languages.

- We present *Sugarclipse*, an extensible IDE for SugarJ based on the Spoofax language workbench. Our growable IDE complements the syntactic extensibility of SugarJ with the capability of providing domain-specific editor services that conform to the embedded DSLs.

- We validate our approach through realistic *case studies* of fully-fledged editors for XML and Latex. We demonstrate how our IDE supports domain-specific and programmer-defined editor configuration languages as well as deriving editor services from language specifications.

Sugarclipse is an open source project that is publicly available at http://sugarj.org/editor. Our case studies and, in particular, the source of this paper are available at the same location.

2. An overview of Sugarclipse

Sugarclipse, as shown in Figure 2, consists of an editor that features services such as syntax coloring, error marking and code completion. Sugarclipse has built-in support for Java syntax only, but all of Sugarclipse's editor services are user-extensible: Additional syntax and editor services can be imported from libraries.

2.1 Using Sugarclipse

A Sugarclipse user activates editor support for an additional language by importing a corresponding library. For example, in Figure 2, the sugar library xml.Sugar provides a grammar for embedded XML documents, and the editor library xml.Editor provides editor services for XML. This editor library specifies syntax colouring, outlining, folding etc., for embedded XML documents without invalidating the built-in services for Java. For example, the resulting editor contains code folding and outlining for both Java and XML combined. The additional editor support only affects the XML part of the document and leaves the remaining editor support intact, most visible in Figure 2 from syntax highlighting (including correct highlighting of quoted Java code nested inside XML).

We can extend the XML example by requiring that the embedded XML should conform to a specific schema. Knowing that schema, Sugarclipse provides even more tailored editor support for the embedded XML, including content completion providing a list of all valid tags and error reporting for validation errors. To activate the additional editor support, the user imports the editor library xml.schema.BookSchema derived from the schema.

2.2 Editor services

A Sugarclipse user can also assume the role of editor-service developer, because editor services are specified declaratively within Sugarclipse. This is more expressive than setting options in the Eclipse menu and significantly easier than manually extending Eclipse by writing a corresponding plugin. In addition to error marking, Sugarclipse lifts and extends eight different editor services from Spoofax [17]. Each service can be declaratively specified in a domain-specific language.

- *Syntax coloring* highlights source code using a colored, bold or italic font.

- *Code folding* supports collapsing part of the source code to hide its details.

- *Outlining* gives a hierarchical overview over the current document and enables fast navigation.

- *Content completion* provides proposals for complementing the current source code.

- *Reference resolving* resolves a construct (typically a name) to its declaration and provides facilities to navigate to the declaration directly ("CTRL-click").

- *Hover help* displays documentation as a tooltip when hovering over a documented entity with the mouse.

- A *refactoring or projection* applies a transformation to (parts of) the source code and writes the result either in the original or a separate file.

- *Parentheses matching* marks matching parentheses in the source code and adds closing parentheses automatically. This service is also essential for *automatic indentation* after line breaks.

Conceptually, editor services can be understood as procedures that decorate syntax trees, for example, with coloring information. Sugarclipse then interprets these decorated trees and maps the decorations to the original source code or other means of visualization such as a separate outline window or a completion proposal viewer.
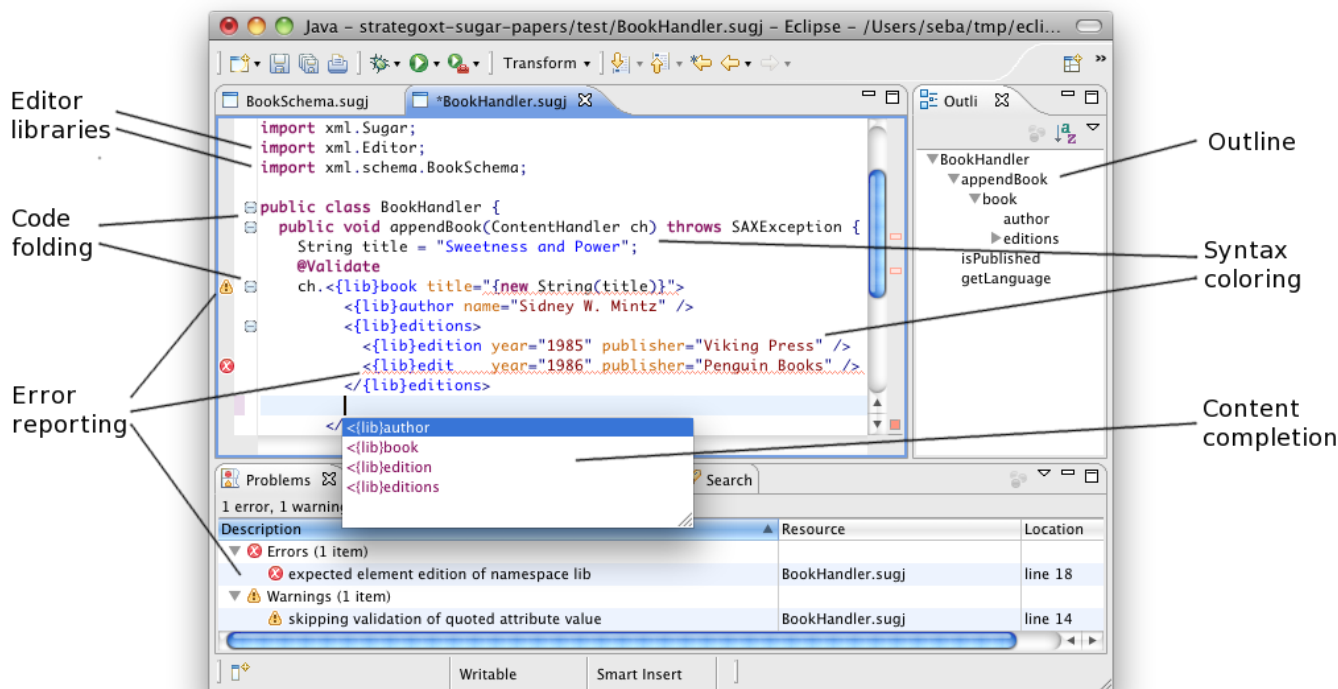
Figure 2. Sugarclipse extended through the imported editor libraries. The quoted Java expression **new** String(title) is highlighted using the typical Java coloring, while the surrounding XML code uses an XML-specific coloring service.

```
package xml;

import editor.Colors;
import xml.XmlSyntax;

public editor services Editor {
  colorer
    ElemName : blue (recursive)
    AttrName : darkorange (recursive)
    AttValue : darkred (recursive)
    CharData : black (recursive)

  folding
    Element

  outliner
    Element
}
```

Figure 3. Editor library for coloring, folding and outlining of XML code.

Since editor services are mere tree decorators, their definitions are fairly simple in most cases (the definition of refactorings and projections being an exception). To reflect this simplicity in editor service implementations, we use an extended version of the declarative editor service configuration language provided by Spoofax [17].

Developers can bundle multiple editor-service specification in an editor library (declared with **public editor services**). For example, the xml.Editor library shown in Figure 3 provides editor ser-

vices for coloring, folding and outlining XML documents using declarative tree decoration rules. Each tree decoration rule specifies a syntax tree pattern to match against and the decoration to apply to matched trees. For example, the XML coloring rules match on trees of the kind ElemName, AttrName, AttValue and CharData, that is, trees derived from these non-terminal sorts as defined by the imported sugar library xml.XmlSyntax. The coloring rules thus declare that XML element names are shown in a blue font, XML attribute names in a dark orange font, etc., and that the coloring recursively applies to all nodes in the matched trees. Similarly, the folding and outlining services declare that XML elements are foldable and XML documents show up in the outline of source files.

We specifically support the development of editor libraries by providing, bundled with Sugarclipse, an editor library for writing editor libraries. In similar fashion, we encourage other developers of language embeddings to accompany their embeddings with editor support in the form of editor libraries.

3. Editor libraries

The basic use of editor libraries, as described in the previous section, is to serve as containers for editor service specifications. Before discussing the composability of editor libraries in detail, we describe a number of advanced usage patterns for editor libraries in SugarJ.

3.1 Domain-specific editor configuration languages

SugarJ supports syntactic abstraction over all of its ingredients, that is, Java code, syntactic sugar, static analysis specifications and, now as well, editor configurations. This design enables the

```
import xml.schema.XmlSchema;

public xmlschema BookSchema {
  <xsd:schema targetNamespace="lib">
    <xsd:element name="book" type="Book" />

    <xsd:complexType name="Book">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="author" type="Person" />
        <xsd:element name="editions" type="Editions" />
      </xsd:choice>
      <xsd:attribute name="title" type="string" />
    </xsd:complexType>
  </xsd:schema>
}
```

Figure 4. An excerpt of the Book XML Schema. The xml.schema.XmlSchema library provides validation and editor services for XML schemas themselves.

development of customized and potentially domain-specific editor service configuration languages. For example, we have applied SugarJ's syntactic extensibility to provide an XML-specific editor service configuration syntax in the style of Cascading Style Sheets (CSS):

```
import editor.Colors;
import xml.CSS;
import xml.XmlSyntax;

public css CSSEditor {
  Element { folding; outlining }
  ElemName { rec−color : blue }
  AttrName { rec−color : darkorange }
  AttValue { rec−color : darkred }
  CharData { rec−color : black }
}
```

This CSS-style editor configuration corresponds and, in fact, desugars to the editor configuration in standard editor service syntax shown in Figure 3; CSS is just another syntax for configuring editor services.

3.2 Staged editor libraries

Many editor services are not static, but rather depend on the contents of the file being edited and imported files. For example, hover help for non-local Java methods depends on the method definitions in other files and code completion for XML elements depends on the corresponding schema. Hand-written IDEs support such editor services by managing a set of files as a project, explicitly coordinating between the information retrieved from each file. Unfortunately, neither SugarJ nor Spoofax have a notion of projects: In Spoofax, editor services for different files are independent, and in SugarJ, files are processed one after another. Sugarclipse, however, supports separate *generation and application stages* for editor libraries from different source files, which enables rich patterns of interaction between editor services of individual source files.

The central idea of our *staging pattern* is to first generate editor services from domain-specific declarations in one file and to later use them in another file. The generated editor services may well be of auxiliary nature such as a mapping from method names to the documentation of these methods, which a hover help editor service can query to display documentation of a method as a tooltip. In general, Sugarclipse employs the transformation language Stratego [29] for auxiliary editor services, and an import statement brings the generated editor services into scope.

For example, we applied the staging pattern to promote XML schemas as domain-specific declarations for XML editor services

that are specific to an XML dialect. Such editor services include XML validation and tag completion. Figure 4 shows an excerpt of the Book XML schema, which declares a dialect of XML for describing books. From this schema, we generate the definition of a static analysis as well as code completion. For the former, we desugar an XML schema into a set of Stratego rules that traverse a given XML document to check whether this document conforms to the schema. In other words, we generate a type checker for each XML schema. The result of applying the XML Book type checker is shown in Figure 2, where quoted Java expressions within an XML document are marked but ignored otherwise. Furthermore, our XML Schema embedding desugars each schema into a set of schema-specific completion templates. For instance, the following completion template results from desugaring the above Book schema.

```
completion template : Content =
  "<{lib}book title=\"" <string> "\">"
  "</{lib}book>"
```

Accordingly, when importing the Book schema, Sugarclipse recognizes the accompanying editor services and provides code completion to the programmer as shown in Figure 2.

As this case study illustrates, Sugarclipse supports the implementation of editor services that involve multiple files using a generative approach; the staging pattern effectively facilitates data flow from one source file to another.

3.3 Self-applicability

Like conventional libraries, editor libraries are self-applicable, that is, editor services can be used during the development of other editor libraries. For example, we have implemented code completion for the code completion editor service using an editor library:

```
public editor services Editor {
  completions
    completion template : EditorServiceCompletionRule =
      "completion template" " : " <Sort> " =\n\t"
        "\"" <prefix> "\" <" <placeholder> ">"
}
```

This template provides content completion for completion templates themselves. Completion templates are represented as sequences of strings and placeholders such as <Sort>, which Sugarclipse marks for the user to replace. The above completion template expands into the following code on selection, where the underlined fragments are placeholders:

```
completion template : Sort =
  "prefix" <placeholder>
```

More generally, we provide full editor support for writing editor libraries in Sugarclipse using editor libraries.

4. Editor composition

A key feature of Sugarclipse is the ability to compose editor libraries. For example, we can import support for regular expressions and XML in the same document. The editor then supports both language extensions with corresponding syntax highlighting, and other facilities. Editor libraries cooperate to present a coherent user interface even though their respective authors might not have planned for their editor library to be used in that exact combination of editor libraries.

We can compose editor libraries developed independently, such as regular expressions and XML, but we can also develop editor libraries that extend other libraries and editor libraries that explicitly interact with other editor libraries through extension points. Let us illustrate such interaction with an example from the domain of

text documents (which we will describe in more detail in Sec. 6.2): We express a bibliography database in one language (e.g., Bibtex-like) and write the text with references to bibliography items in another language (e.g, Latex-like). When composing both languages, we would like to add editor services to navigate from bibliography references to their definitions, to suggest available references with content completion, to provide hover help, and so forth. These editor services need to bridge elements in different files and from different languages.

Although different kinds of interactions and even conflicts between editor services are possible, we argue that editor services are largely independent and have local effects. In addition, for many services, interactions can be implicitly resolved using generic strategies. Finally, for intended interactions as in the bibliography example, we apply the staging pattern for explicitly coordinating editor services.

4.1 Local variation and global consistency

Editor libraries extend the *local* behavior of the SugarJ editor. There are different notions of locality:

- Editor libraries affect only files that import them explicitly. In these files, only the part after the import is affected.

- Editor libraries that extend distinct editor services compose naturally. For example an editor library defining syntax coloring will not conflict with another editor service providing content completion.

- Editor libraries usually reason about small and local subtrees of the abstract syntax tree. For example, an editor library typically defines syntax highlighting for specific syntactic forms, not for the overall program, and editor libraries that accompany a DSL embedding reason over tree fragments of that DSL only. Editor libraries that act on different parts of the abstract syntax tree naturally compose. For example, the XML editor library shown in Figure 3 only decorates XML fragments of the syntax tree and does not affect Java fragments.

The *global* behavior of the SugarJ editor, however, is fixed and cannot be extended by editor libraries. For example, the SugarJ editor supports a fixed set of editor services such as syntax highlighting, reference resolving, hover help, etc. as discussed in Section 2.2. The SugarJ editor presents a coherent user interface to access these editor services. For example, key bindings or the visual appearance of error markers are defined by the SugarJ editor directly and are therefore consistent across error libraries.

Together, global consistency and local variation go a long way ensuring that Sugarclipse supports arbitrary languages while still providing a coherent user interface. Some interactions between editor libraries cannot be resolved by locality, however, and require implicit or explicit coordination between editor libraries.

4.2 Implicit coordination

Although most editor libraries work locally, their results can conflict or overlap. For most editor services, conflicts can be resolved implicitly following generic strategies: aggregation and closest match.

For most editor services, aggregating results of different editor libraries is sufficient. For example, in our XML embedding, both Java and XML code completion services would respond to a prefix ch., which could be followed by a Java method name or an XML element. Sugarclipse simply shows all completion proposals. Aggregation works similarly for code folding, outlining and error marking.

For some other services, primarily syntax highlighting and hover help, simple heuristics can resolve conflicts implicitly. For example, when one editor library specifies that all tokens in assignments should be blue, whereas another editor library specifies that all tokens in while loops should be red, Sugarclipse needs to coordinate between these editor libraries and decide in which color to display tokens in an assignment nested within a while loop. As heuristic, we propose a closest-match rule, as used for style sheets in HTML: Color information, hover help, and other specifications on an AST node overrule corresponding specifications of the parent node; always the most specific information is used for presentation. For our example above, the closest-match rule displays the assignment blue, because the match on assignments is more specific (closer to the tokens in question) than the match on the while loop.

Aggregation and the closed-match rule resolve most conflicts implicitly in a natural way. Explicit coordination is usually necessary only for intended interactions.

4.3 Explicit coordination

Not all editor libraries are supposed to be independent. Editor libraries might explicitly extend the behavior of other libraries or interact with them in controlled ways.

An editor library can add additional editor-service specifications to another library. For example, the XML-Schema library builds on top of the XML library and extends it with code completion and error checking. In addition, different editor libraries can interact explicitly through the staging pattern to share data and coordinate editor services. The staging pattern, described in Section 3.2, enables communication from one editor library to another through the generation of auxiliary editor services. In our example, the bibliography database shares information about all known entries by generating an auxiliary editor service (technically: Stratego rules) that maps entry names to their definitions:

```
bibtex−entry : "Hudak98" −>
   BibtexEntryStm("@inproceedings",
     BibtexEntryName("Hudak98"), ...)
```

Any other editor library can use this information to integrate with the bibliography editor library. For example, our Latex editor library supplies hover help and content completion for citations (\cite{...}), and checks for undefined references.

4.4 Limitations

Although editor-library composition is usually straightforward in practice, there are limitations. Most significantly, we cannot provide modular guarantees about editor services in hostile environments.

Editor services use a global namespace without hiding. In principle, editor libraries could access (auxiliary) services of all other imported editor libraries and extend them. We discourage uncontrolled sharing and use naming conventions (similar to fully qualified names in Java) to avoid accidental name clashes. The staging-based communication between editor libraries relies on conventions and implementation patterns; there is no explicit scoping concept for staged services yet.

Furthermore, editor services should make little assumptions about the global structure of the AST. Editor services are used in a context where the AST of a file typically contains structures from different languages. For example, navigating from an AST element to its direct parent should be avoided, instead one should search for a direct or indirect parent of the expected type. Such strategies make editor libraries more robust against additional language extensions. However, Sugarclipse currently does not enforce locality and cannot detect violations modularly.

Building a module system to provide explicit namespaces and checked interfaces for Sugarclipse and the underlying SugarJ is an
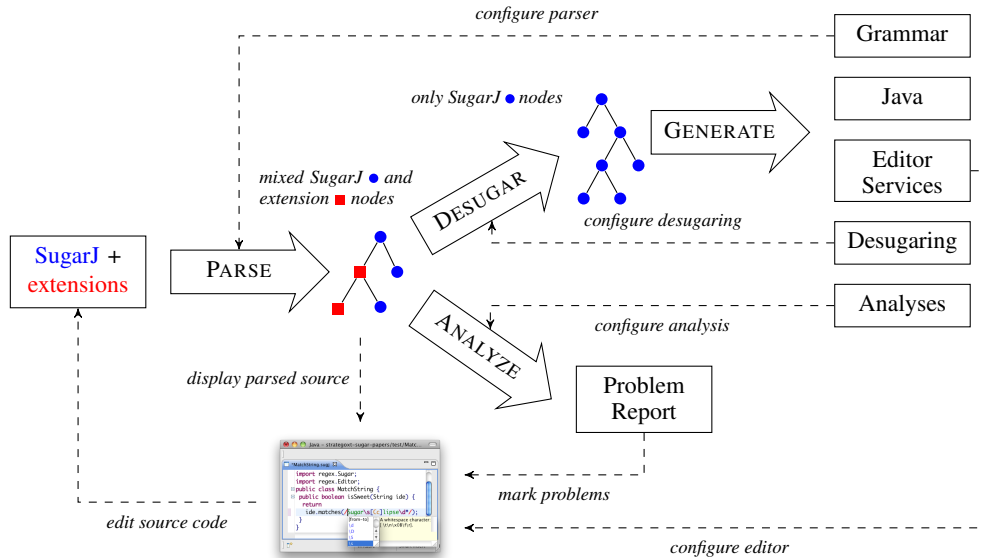
Figure 5. Data flow in Sugarclipse. The results of the processing pipeline (⟹) are used to configure (- -►) the earlier stages.

interesting avenue for future work. Such a module system should prevent name clashes and control what kind of information (technically: which Stratego rules) can be shared between editor libraries. To a large degree this seems to be a straightforward adoption of concepts from other module systems, such as the compilation manager in Standard ML [2]. On top, semantic interfaces could enable modular detection of conflicts between two editor libraries, so we would report an error when importing both of them.

In our experience, conflicts between editor libraries are rare and patterns for explicit coordination are easy to implement when required. Naming conventions and implementation patterns seem sufficient to avoid conflicts in practice. Hostile environments (deliberate attacks against editor libraries) are currently not a practical concern for editor extensions. Sugarclipse appears useful for many practical tasks, even without modular guarantees. Nevertheless, more experience in a broader set of applications is necessary to assess the difficulty of editor library composition.

5. Technical realization

In Sugarclipse, we combine the sugar libraries of SugarJ [9] with the IDE foundation of Spoofax [17] to support editor libraries for growing an IDE. SugarJ parses a file incrementally, because each declaration can extend the grammar of the rest of the file, and like in Spoofax, we use a generic editor component which can be configured to support different languages. Sugarclipse adds editor libraries into the mix: sugar libraries can desugar source code into editor libraries, and editor libraries in scope reconfigure the editor while a source file is edited. Together, these components enable to grow the IDE with editor libraries.

5.1 Architecture

Source code documents are often processed in many stages, compile-time and run-time being traditionally the most well-known. A library can affect several of these stages. For example, a Java class library contains, among other things, type definitions and method bodies. Clients of the library are type-checked against the type definitions in the library at compile-time, but method calls to method definitions in the library are executed at run-time. In our previous work on sugar libraries in SugarJ, we have broadened the

applicability of libraries by considering additional stages: parsing, desugaring and analysis. Sugar libraries contain grammar or desugaring rules to affect these stages of the SugarJ implementation. In the present work on editor libraries, we consider an integrated development environment as an integral part of the language implementation, that is, we consider an additional editor stage, which can be affected by editor libraries.

The interaction of these stages in Sugarclipse is shown in Figure 5. The editor stage is depicted by the Sugarclipse screenshot, all other stages are depicted as block arrows (⟹). The parsing stage transforms a source code document into an heterogeneous abstract syntax tree with nodes from different language extensions. The desugaring stage expands all nodes corresponding to language extensions into nodes of the base language, and the generation stage transforms the resulting homogeneous abstract syntax tree into separate source code artefacts containing grammar extensions, desugaring rules, editor services and so on. At the same time, the analysis stage checks the heterogeneous abstract syntax tree and produces a problem report listing all found errors and warnings.

The results of compilation can configure earlier stages as depicted with dashed arrows (- -►) in Figure 5. For example, generated grammars configure the parsing stage for clients of a sugar library and the generated analyses are applied in the analysis stage. In addition to these stages, the results of compilation also configure the editor, as we detail in the following subsections. In particular, the editor displays the input file's content with syntax highlighting according to the parsed source code, marks problems found by the analysis stage and behaves according to the editor services currently in scope. When the programmer changes code in the editor, the processing pipeline is run again to produce updated grammars, desugarings, etc., and any changes in these artifacts are reflected in the various stages.

5.2 Incremental parsing

Sugarclipse supports languages with extendible syntax by relying on SugarJ for incremental parsing. Parsing with SugarJ is an incremental process because import declarations and syntax definitions can change the syntax for the rest of the file. To this end, SugarJ repeatedly parses a single top-level entity (e.g., import or class dec-

laration) followed by the remainder of the file as a string. For each such parse, SugarJ extends the grammar according to the parsed entitiy before continuing to parse the remainder of the file.

In the context of Sugarclipse, two additional concerns arise. First, the parser must associate every node of the abstract syntax tree with position information which the editor needs for marking errors, moving the cursor for reference resolving or outline view navigation, and so on. Second, the parser must associate some nodes of the abstract syntax tree with tokens that are used for syntax highlighting.

To reconcile incremental parsing of SugarJ with creating tokens and collecting position information, we use the same tokenizer for each parse. After each parse, we partially retract the tokenizer to ignore all tokens after the top-level entity and to reset the parser position accordingly. After parsing, we combine the trees of all top-level entities and ensure that the tree nodes have pointers to corresponding tokens and position information.

5.3 Dynamic loading of editor services

Sugarclipse supports editor libraries by relying on Spoofax to provide a generic Eclipse-based editor which can dynamically load and reload editor services. Although Spoofax still distinguishes the building and loading of editor services into separate phases, its dynamic loading capability forms the basis for editor services that are transparently built and loaded with library imports in Sugarclipse.

In the context of Sugarclipse, two additional concerns arise. First, parse tables and editor services need to be adapted on-the-fly whenever the corresponding language or editor libraries change. This is accomplished by running the full processing pipeline whenever a file has been changed and needs to be reparsed. The editor then dynamically reloads the possibly regenerated editor services. To ensure optimal responsiveness of the editor, generation and reloading happens in a background thread. Any services that were already loaded and parse tables that were already built are cached. Second, in Sugarclipse, each file determines the required language components and editor components by means of library imports. Sugarclipse therefore needs to maintain a separate set of editor services for each file. In contrast, Spoofax normally uses a language-level factory class. We subclass that factory with a specialized implementation that loads editor services in a file-specific fashion.

To conclude, in the present section we presented Sugarclipse's architecture, which augments SugarJ's processing pipeline with an additional editor stage that can be configured via editor libraries. The editor stage connects to the processing pipeline through presenting the parsed syntax tree, marking errors and loading the (possibly staged) editor services. The following section reports on our experiments with this realization of Sugarclipse.

6. Case studies

We applied our Sugarclipse implementation to demonstrate the practicability of editor libraries. We have developed editor libraries for a small number of simple language extensions such as regular expressions, where editor services only act locally and no explicit coordination is necessary. These simple editor services compose with the basic SugarJ editor services and other simple editor services through implicit coordination. For example, our regular expression editor library would compose easily with an editor library for SQL to provide editor services for regular expressions nested within SQL statements, because each library acts on syntax tree of the respective DSL only.

In addition to these simple editor libraries, we have conducted two realistic case studies to evaluate the practicability and composability of editor libraries for larger languages: XML and LaTeX. In both case studies, we demonstrate Sugarclipse's support for the
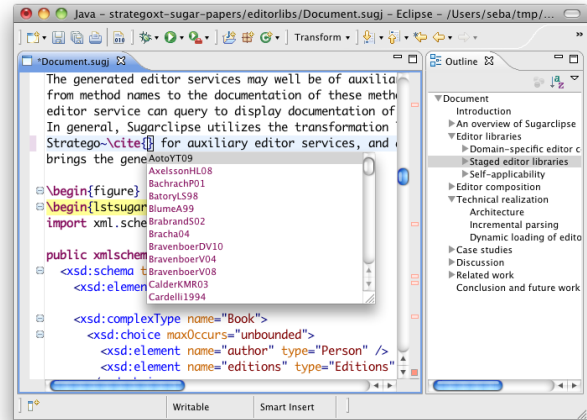

Figure 6. Editor services for Latex in Sugarclipse: outline, nested syntax coloring, citation completion, reference checking, code folding.

staging of editor services, and in the Latex case study we additionally apply explicit coordination to compose editor libraries.

6.1 Growing an XML IDE

XML and XML Schema demonstrate many interesting facets of editor libraries, including domain-specific editor configuration languages and editor library staging as described in Section 3. Although the XML Schema editor library extends the editor library for XML with schema-specific tag completion and validation, both libraries compose with editor services such as Java or SQL. This composability is based on locality and implicit coordination in the form of aggregation and the closest match rule (cf. Section 4).

In summary, we have grown Sugarclipse through the use of syntactic extensions and editor libraries into an XML-aware IDE that features coloring, folding, outlining, schema-specific tag completion and XML validation. Several potential editor services have not been implemented so far, but qualify as future student projects, for example, reference resolving according to XML Schema references or hover help to display documentation from the schema within the XML document.

6.2 Growing a Latex IDE

Language extensions such as XML or regular expressions extend the Java fragment of SugarJ and provide editor services that compose with Java services. Compared to Java, these language extensions are relatively small and do not cross-cut Java programs too much. Therefore, we also wanted to gain experience with incrementally growing a language from scratch by composing multiple sublanguages and their editor services into one unified language. To this end, we grew Sugarclipse into a Latex IDE by composing a Latex core with libraries for mathematical formulas, listings of (parsed and IDE-supported) source code and Bibtex bibliographies and citations. However, we only provide an IDE frontend for the Latex language and its libraries: Latex code in Sugarclipse compiles to regular Latex files, which use regular Latex libraries. In Figure 6, we show a screenshot of our Latex IDE.

The basic Latex language consists of macro calls **emph**{arg}, environments **begin**{abstract}...**end**{abstract}, structure declarations **section**{A}, **paragraph**{B}, and so forth, and of course text. We support these concepts in our core Latex syntax definition and editor library, which, for example, highlights section headers in a

bold, blue font, proposes code completions for macro calls, or provides a structural document outline.

In separate libraries, we define the syntax and editor support for various extensions of the Latex core:

First, the math library introduces a new language construct for formulas $n \to n + 1$ and according editor services (e.g., highlighting). These services act locally and thus compose with other Latex extensions.

Second, the listings library supports source code listings in a document. Typically, such source code listings are unparsed, unchecked and, often enough, erroneous. Within the code listing, all language-specific editor services are available, if the corresponding editor libraries are in scope. This way, we compose the Latex editor services with editor services for Java, services for editor libraries and services for language extensions such as XML Schema. For example, while writing this paper, Sugarclipse provided us syntax coloring and error checking for the schema in Figure 4, as shown in the screenshot of Figure 6.

Third, we separately implemented a syntactic extension and editor library for Bibtex, which, for instance, provides reference resolving and hover help for string constants within a bibliography. Bibtex and Latex interact via citations \cite{...} that occur in a Latex document and refer to Bibtex entries. However, the according editor services do not compose automatically in a meaningful way; explicit coordination is necessary to provide code completion, hover help or checking for undefined references. We provide these editor services for citations by generating and explicitly coordinating services as described in Section 4.3.

The key feature of our Latex IDE is its extensibility: users can extend the IDE through syntax definitions and editor libraries to support, for instance, vector-graphics libraries. Staging and explicit coordination of editor services provides the conceptual means for implementing a wide range of powerful IDE extensions.

7. Discussion

In this paper, we have focused on the integration of editor libraries into a syntactically extensible object language such as SugarJ. In this section, we point out a number of further application scenarios for editor libraries and discuss whether it is sensible at all to organize editor services as part of source files.

7.1 Language embedding

There are several approaches to embed domain-specific languages, even when the host language is not syntactically extensible. Typical examples are string-based embeddings and embedding a language with constructs of the host language, known as pure embedding [14]. The latter works even better if the host language has a flexible syntax, as in Scala. In Figure 7, we illustrate three typical embeddings: embedding regular expressions as plain strings in Java, embedding XML as API calls in C#, and embedding LINQ-style queries in Scala.

Even for DSL embeddings in a nonextensible language, we want to add domain-specific IDE support. Even if regular expressions are embedded as strings or XML is embedded as API calls, we want to provide domain-specific editor services such as syntax coloring and content completion. Using editor libraries, DSL implementers can accompany their DSL embeddings with editor services to support programmers.

In the case of string-based embedding, Sugarclipse attempts to parse the document in more detail than the host language. In the pure-embedding scenario, we provide editor-service declarations that reason about more complex syntactic structures, for example nesting of XAttribute instantiation inside XElement instantiation. The general library mechanism works equivalently for languages that are syntactically extensible or not.

```
s.matches("a\\.∗[0−9]")
```
(a) String-based embedding of regular expressions in Java.

```
new XElement("book",
    new XAttribute("title", new String(title)),
    new XElement("author", new XAttribute("name", "Mintz")))
```
(b) Pure embedding of XML in C#.

```
from(books)(b => where(b.isPublished) select(b.title))
```
(c) Pure embedding of SQL in Scala.

Figure 7. Typical DSL embeddings in Java, C# and Scala.

7.2 Library-based pluggable type systems

The notion of pluggable type systems was first proposed by Bracha and describes type systems that accept extensions (plugins) to enforce additional static analyses on demand [4]. Programmers can configure a pluggable type system by selecting a set of extensions to activate. Due to Sugarclipse's support for marking user-defined errors and warnings visually in the source file and problems view, Sugarclipse is especially well-suited for the application of library-based pluggable type systems. In a library-based pluggable type system, type system extensions are organized in libraries and activated through usual import statements.

Pluggable type systems enable the definition of specialized language subsets for various purposes: pedagogical language subsets prohibit the use of certain language constructs, convention-based language subsets enforce the compliance with code style or author guide lines, language subsets for a particular platform (e.g., Java targeting Google Web Toolkit) often support part of the standard library only. However, more sophisticated language restrictions are possible as well. For instance, we implemented XML validation as a library-based type system plugin.

7.3 Are editor libraries a good idea?

Sugarclipse raises the question whether it is a good idea to have editor definitions as part of the sources of a program. One could argue that such metadata should be kept separate, because they are not part of the program semantics and they potentially couple the sources to a specific IDE. Our answer to this objection is twofold: First, SugarJ and Sugarclipse are attempts to tear down stratifications into base and metalevel. This enables self-applicability and the use of the same mechanisms for abstraction, versioning, deployment, evolution and so forth at all metalevels. Second, we tried to reduce the conceptual coupling to a specific IDE by making the editor definitions as abstract as possible, such that Sugarclipse-like functionality can be adopted for many IDEs. While more experience is necessary for the final word on editor libraries, we believe that the positive evidence we collected makes further research worthwhile.

8. Related work

Our work follows in a line of previous work on extensible and customizable code editors, IDEs, and language workbenches. We provide an overview and compare these works to Sugarclipse.

Extensibility of code editors and IDEs. Notable early examples of extensible code editors are Emacs and Vim. They support extensibility by means of plugins, written in dynamic languages such as Lisp and Vim Script. Using APIs and hooks to coordinate actions in the editor, these plugins can introduce syntax highlighting and shortcuts or commands specific to a language. Plugins that intro-

duce more advanced features, such as inline error markers are rare for these editors.

Modern IDEs distinguish themselves from the traditional code editors and programming environments by combining a rich set of programmer utilities such as version management with a variety of sophisticated language-specific editor services [12]. These IDEs parse the source code as it is typed, rather than treating it as text with regular-expression-based syntax highlighting. The parsed abstract syntax tree is used for semantic editor services such as inline error marking and content completion. Examples of these IDEs are Eclipse, IntelliJ IDEA, and Visual Studio. Each provides extensibility by means of plugins, written in general-purpose languages such as Java or C#, for which APIs and hooks are provided to customize the IDE experience.

Extensible code editors and IDEs use a plugin model for the organization and distribution of editor components. In contrast to our library-based approach, plugins are not part of the object language but are externally implemented and integrate into an editor's architecture directly. This has a number of significant implications. First, editor libraries can be activated through object language imports on a *per-file basis*, whereas plugins require external activation instead, for example, on a per-editor mode or per-language basis. Second, independent editor libraries typically *compose* based on locality and implicit coordination, whereas plugins have to be designed for composition a priori. Third, editor libraries are *declarative* and describe how to perform editor services, rather than imperatively changing the editor execution. Finally, while IDEs such as Eclipse or Visual Studio require the environment to be restarted whenever the implementation of editor service changes, editor libraries ensure a transparent compilation model.

Customizability of code editors and IDEs. IDEs usually provide some adaptability through configurations such as custom coloring schemes or user-defined code templates. However, these facilities are often coarse-grained and hard to deploy or share. For instance, Eclipse's standard Java plugin JDT defines a fixed set of colorable entities (decimal and hexadecimal numbers must look the same), requires completion templates to apply either to Java statements or type members only, or to complete Java (no completion templates for expressions only) and does not support an import and export mechanism for all editor configurations. In contrast, editor libraries are deployable just like usual Java libraries and enable precise configuration of editor services based on the language's full syntactic structure. Furthermore, since editor libraries are part of the object language, it is possible to package them with conventional programming libraries. This enables library-specific editor services such as code completion templates for typical use cases of an API or warnings for depreciated uses.

Language workbenches. Language workbenches are tools that integrate traditional language engineering tools such as parser generators and transformation systems and tools to develop IDE support [11]. By combining these tools and by providing IDE support for these metaprogramming tasks, language workbenches enable developers to efficiently create new languages with IDE support.

Language workbenches based on free text editing and parsing include EMFText [13], MontiCore [19], Rascal [27], Spoofax [17], TCS [15] and Xtext [7]. These workbenches provide modern editor service facilities such as content completion, following in a line of work on extensible IDEs with metaprogramming facilities, such as the Meta-Environment [18, 26]. Similar to our work, these workbenches provide support for developing and using editor services. However, they strictly separate metaprogramming and programming. Languages and editor services are deployed together in such a way that they apply to a certain file extension. Any changes to the language or editor service can only be applied at language-

definition level. In contrast, in our work editor services can be freely imported and composed as editor libraries across any number of metalevels, which enables the self-application of editor services.

In addition to language workbenches designed to implement arbitrary textual languages, there are also tools that are based on a fixed host language. Examples include Helvetia [22], a Smalltalk-based environment, and DrRacket [10], aimed at the Racket programming language (formerly known as Scheme). Helvetia supports syntactic extensibility and custom syntax highlighting for extensions through a dynamic meta-object protocol, but has no support for more sophisticated editor services such as reference resolving, content completion, or static checks. DrRacket does not provide the same syntactic flexibility as Helvetia or Sugarclipse, but does provide autogenerated reference resolving editor services. In Helvetia, language definitions can be loaded for a Smalltalk image and activated in parts of the application. In DrRacket a language definition can only be selected at file level using the `#lang` directive. Both tools are highly tied to their respective host languages, using dedicated metaprogramming systems. For instance, reference resolving in DrRacket demands that new constructs for binding identifiers are defined in terms of predefined binding constructs of the Racket language. In contrast, our editor libraries approach is language-agnostic as our Java-independent case study for Latex shows.

MPS is a language workbench based on projectional editing rather than free text editing [30, 31], notable for its support for compositionality of languages. It allows language extensions to be activated in a specific parts of an application, but does not organize them as true libraries. MPS strictly separates metaprogramming and programming by providing fixed templates for syntactic and semantic customization of language components.

In summary, while current language workbenches integrate metaprogramming and programming into a single IDE environment, they still strictly separate language definitions and language uses into two different worlds. They do not support composition of extensions as libraries, although MPS allows project-level configuration of the set of languages that is supported. Unlike our editor libraries they cannot import and compose editor services. They also provide limited support for self-applicability and abstraction over editor service definitions. Our work, which is based on the Spoofax [17] language workbench, shares properties such as providing IDE support for language development, but does away with the restrictions of the two-world approach of current language workbenches.

9. Conclusion and future work

Our main idea in Sugarclipse is the application of libraries for organizing IDE extensions as reusable units. As our case studies show, editor libraries are particularly beneficial in combination with syntactically extensible programming languages such as SugarJ and represent an important step towards our ultimate goal of *language libraries*. Language libraries enable the implementation of all aspects of a language as a library. Currently, we support the library-based adaptation of parsing, desugaring, analyzing and editor presentation, but lack library-based extensibility for implementing the semantics of a language extension. In our future work, we would like to support the configuration of builder services that declare the semantics of embedded languages and integrate into Sugarclipse naturally. Builder services should replace traditional build scripts completely and specify the order as well as the tool used to build a set of source files.

In addition, we would like to further investigate the modularity and composability of editor libraries. In particular, we would like to explore scoping mechanisms for editor libraries that retain composability while providing clearer interfaces for explicitly co-

ordinating services with staged editor libraries. We also plan to conduct a large-scale case study to evaluate the composability of editor libraries more accurately, namely Java Server Pages. Java Server Pages brings together a number of languages such as HTML, Java, JavaScript and CSS. We plan to provide editor libraries for each of these language separately and to compose the resulting editor libraries to form an editor library for Server Pages. While conducting this case study, we would furthermore like to explore new declarative means for explicitly coordinating editor libraries.

Acknowledgments

References

[1] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 31–42. ACM, 2001.

[2] M. Blume and A. W. Appel. Hierarchical modularity. *Transactions on Programming Languages and Systems (TOPLAS)*, 21:813–847, 1999.

[3] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 31–40. ACM, 2002.

[4] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[5] M. Chapman. Extending JDT to support Java-like languages. Invited Talk at EclipseCon'06, 2006.

[6] S. Dmitriev. Language oriented programming: The next programming paradigm. Available at `http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf.`, 2004.

[7] S. Efftinge and M. Voelter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[8] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–18. ACM, 2007.

[9] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2011.

[10] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[11] M. Fowler. Language workbenches: The killer-app for domain specific languages? Available at `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[12] M. Fowler. PostIntelliJ. Available at `http://martinfowler.com/bliki/PostIntelliJ.html`, 2005.

[13] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Proceedings of European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, volume 5562 of *LNCS*, pages 114–129. Springer, 2009.

[14] P. Hudak. Modular domain specific languages and tools. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 134–142. IEEE, 1998.

[15] F. Jouault, J. Bézivin, and I. Kurtev. TCS: A DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 249–254. ACM, 2006.

[16] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: Tool framework for feature-oriented software development. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 611–614. IEEE, 2009.

[17] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463. ACM, 2010.

[18] P. Klint. A meta-environment for generating programming environments. *Transactions on Software Engineering Methodology (TOSEM)*, 2(2):176–201, 1993.

[19] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Technology of Object-oriented Languages and Systems (TOOLS)*, pages 297–315. Springer, 2008.

[20] S. McDirmid and M. Odersky. The Scala plugin for Eclipse. In *Proceedings of Workshop on Eclipse Technology eXchange (ETX)*, 2006. published online `http://atlanmod.emn.fr/www/papers/eTX2006/`.

[21] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of Conference on Compiler Construction (CC)*, pages 138–152. Springer, 2003.

[22] L. Renggli, T. Gîrba, and O. Nierstrasz. Embedding languages without breaking tools. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 380–404. Springer, 2010.

[23] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Proceedings of Workshop on Reflection and Software Engineering*, volume 1826 of *LNCS*, pages 117–133. Springer, 2000.

[24] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011.

[25] L. Tratt. Domain specific language implementation via compile-time meta-programming. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):1–40, 2008.

[26] M. Van den Brand, A. Van Deursen, J. Heering, H. De Jong, et al. The Asf+Sdf Meta-Environment: A component-based language development environment. In *Proceedings of Conference on Compiler Construction (CC)*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[27] T. van der Storm. The Rascal language workbench. Submitted to Language Workbench Competition 2011, available at `http://www.languageworkbenches.net/lwc11rascal.pdf.`, 2011.

[28] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 575–599. Springer, 2007.

[29] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings of Conference on Rewriting Techniques and Applications (RTA)*, LNCS, pages 357–362. Springer, 2001.

[30] M. Voelter. Embedded software development with projectional language workbenches. In *Proceedings of Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 6395 of *LNCS*, pages 32–46. Springer, 2010.

[31] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. `http://voelter.de/data/pub/VoelterSolomatov_SLE2010_LanguageModularizationAndCompositionLWBs.pdf`, 2010.

[32] M. P. Ward. Language-oriented programming. *Software – Concepts and Tools*, 15:147–161, 1995.