ELSEVIER

Domain-Specific Languages for Composable Editor Plugins

Lennart C. L. Kats[*,1,2]   Karl T. Kalleberg[+,3]   Eelco Visser[*,1,4]

[*] *Department of Software Technology*
Delft University of Technology
Delft, The Netherlands

[+] *Faculty of Medicine*
University of Bergen
Bergen, Norway

Abstract

Modern IDEs increase developer productivity by incorporating many different kinds of editor services. These can be purely syntactic, such as syntax highlighting, code folding, and an outline for navigation; or they can be based on the language semantics, such as in-line type error reporting and resolving identifier declarations. Building all these services from scratch requires both the extensive knowledge of the sometimes complicated and highly interdependent APIs and extension mechanisms of an IDE framework, and an in-depth understanding of the structure and semantics of the targeted language. This paper describes SPOOFAX/IMP, a meta-tooling suite that provides high-level domain-specific languages for describing editor services, relieving editor developers from much of the framework-specific programming. Editor services are defined as composable modules of rules coupled to a modular SDF grammar. The composability provided by the SGLR parser and the declaratively defined services allows embedded languages and language extensions to be easily formulated as additional rules extending an existing language definition. The service definitions are used to generate Eclipse editor plugins. We discuss two examples: an editor plugin for WebDSL, a domain-specific language for web applications, and the embedding of WebDSL in Stratego, used for expressing the (static) semantic rules of WebDSL.

Keywords: Domain specific language, integrated development environment, editor plugin

1 Introduction

Integrated development environments (IDEs) increase developer productivity by providing a rich user interface and tool support specialized for editing code in a particular software language. IDEs enhance readability through syntax highlighting and code folding, and navigability through cross-references and an outline view. Rather than providing an extensive programming environment for only one specific language, modern IDEs such as Eclipse provide an extensible platform for integrating

language processing tools for multiple languages using a plugin architecture. The flexibility of these platforms supports the integration of tools for specific languages such as editors and compilers, as well as language-independent tools, such as version control and build management systems.

Despite the extensibility of IDE platforms, implementing state-of-the-art support for a new language is a daunting undertaking, requiring extensive knowledge of the sometimes complicated and highly interdependent APIs and extension mechanisms of an IDE framework, and an in-depth understanding of the structure and semantics of the subject language. The popular Eclipse IDE provides a cross-platform, highly extensible software platform, offering support for various languages as well as language-independent utilities. Eclipse is primarily written in Java and offers top-notch Java development tools (JDT) for the Java language. Unfortunately, Eclipse offers little opportunity for reuse of the JDT components to support other languages. Furthermore, because of its scale and broad applicability, the standard extension interface and framework are rather complex.

Key to the efficient development of IDE components are *abstraction*, to eliminate the accidental complexity of IDE frameworks, *modularity*, to reuse definitions of editor components, and *extensibility*, to customize editor components. In the design of abstractions to increase expressiveness, care must be taken to avoid a prohibitive loss of coverage, i.e. the flexibility to customize aspects of a component. Reuse is particularly important when considering *language extension* and *embedding*, as occurs for example in scenarios such as domain-specific language embedding [3,1], and meta-programming with concrete object syntax [3]. Ideally, language combinations are defined modularly, and likewise their IDE support should be composed from IDE support for the constituent languages.

Language development environments facilitate efficient development of language processing tools. While in recent years a considerable number of language development environments have been developed [5,7,12,14,15,17,18], they offer limited support for modular and reusable editor definitions, with the exception of Monti-Core [14,15] and the Meta-Environment [17,18].

MontiCore offers a certain degree of support for language extensions by merging productions of constituent grammars. However, since MontiCore's parser is based on the LL(k) formalism, extensions share the same scanner. Thus, adding extensions with a different lexical syntax is not possible. Other extensions may cause incompatibilities with existing sources (just as the introduction of the `enum` keyword in Java 1.5 excluded programs using `enum` as identifiers, which used to be valid Java). MontiCore also provides an alternative approach, dynamically switching to different scanners and parsers for blocks of code, but this is restricted to embedded languages. In contrast, the Meta-Environment uses Scannerless Generalized LR (SGLR) parsing, which is closed under composition and supports fine grained extensions and embeddings. SGLR has been applied to Java, C, PHP, as well as embeddings and extensions based on these languages [3]. The Meta-Environment has extensive tool support for creating, debugging and visualizing grammars parsed by SGLR. However, it only offers very limited support for the "standard" editor

services programmers expect, such as interactive syntax highlighting, reference resolving, and integration of utilities such as version control.

In this paper, we describe SPOOFAX/IMP, a system that integrates a Java-based implementation of SGLR into the Eclipse environment. To adequately cope with the complexity involved in IDE development *and* evolution, SPOOFAX/IMP uses compositional *editor service descriptors*, defined in domain-specific languages (DSLs), for the specification of IDE components. The DSLs provide a concise notation for specific types of syntactic and semantic editor services. Due to their declarative nature, descriptors can be composed, supporting a modular definition of both stand-alone and composite subject languages. From the descriptors, SPOOFAX/IMP generates an Eclipse plugin, which can be dynamically loaded into the same workspace as the descriptors are developed in — ensuring a shorter development cycle than the customary approach of starting a secondary Eclipse instance.

To further assist in efficient IDE development, SPOOFAX/IMP automatically derives syntactic editor service descriptors for a language by heuristically analyzing the grammar. Using the compositional nature of the descriptors, generated service descriptors can be composed with handwritten specifications, customizing the default behavior. This technique allows for rapid prototyping of editors, and helps in maintaining an editor as a language evolves. When a language definition evolves, the generated components of an editor can simply be re-generated. We have applied these techniques to WebDSL, a mature domain-specific language [22,9] for web application development, which serves as running example in this paper.

The implementation of SPOOFAX/IMP is based on the IMP framework [5], which is designed to be interoperable with different parsers and other tooling, making it usable for our system. Furthermore, SPOOFAX/IMP uses the Stratego language [2] for describing the (static) semantic rules of a language, used for editor services such as cross-referencing and error reporting. For interoperability with the IMP framework, we applied a variation of the program object model adapter (POM) technique, previously used to integrate Stratego and an open compiler [10]. To support IDE feedback in combination with tree transformations, we apply a form of origin tracking [21] by adapting Stratego's generic tree traversal operators to track origins during rewrites.

This paper is organized as follows. First, we describe the definition of editor services, discussing services that handle the presentation of a language in Section 2 and those based on the language semantics in Section 3. In Section 4 we discuss composition of editor services. Section 5 provides an overview of the Eclipse and IMP architecture, and how our implementation augments it. Finally, in Section 6 we discuss related work, and offer concluding remarks and directions for future work in Section 7.

2 Syntactic Editor Services

An editor dedicated to a particular language offers a presentation of source code specifically tailored to that language. For the most part, the presentation is directed
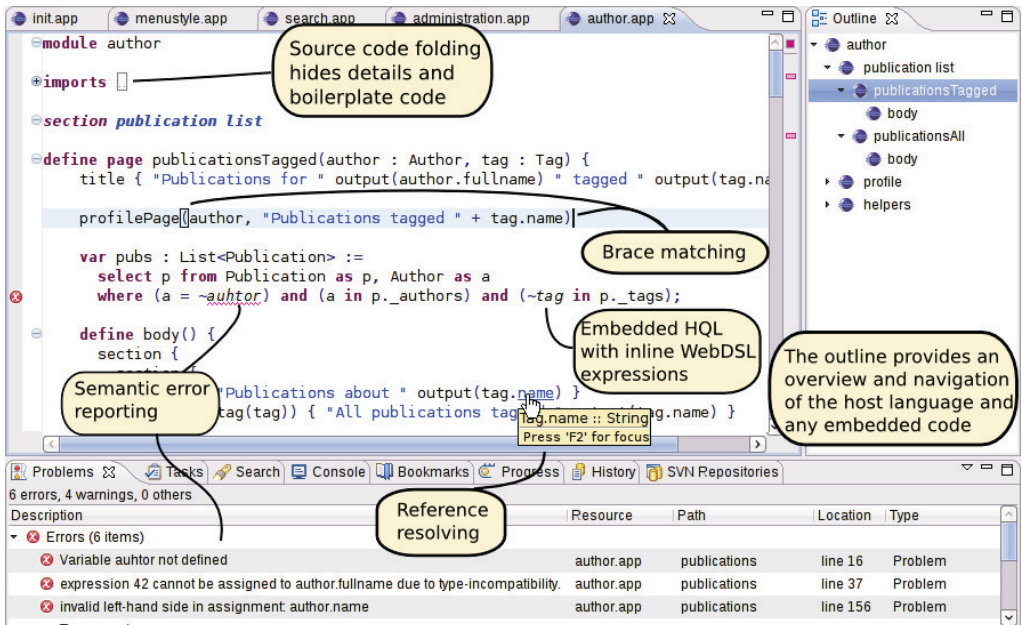
Fig. 1. Editor services for the WebDSL language.

by *syntactic editor services*. These include syntax highlighting, code folding, and the outline view. Figure 1 illustrates these and other services for WebDSL, a domain-specific language for web applications [22,9].

The grammar of a language forms the basis for the specification of syntactic editor services. For example, syntax highlighting can be specified as rules matching against grammar productions and keywords. At run-time, the grammar is used to parse the file and apply the editor service according to its grammatical structure. We employ an SGLR parser and use the modular syntax definition formalism SDF to specify the grammar of a language. Figure 2 shows part of the SDF definition for the WebDSL language. Basic SDF productions take the form

$$p_1 \ldots p_n \texttt{ -> } s$$

which specifies that a sequence of strings matching the symbols p_1 to p_n matches the symbol s. Productions can optionally be labeled with custom names using the `{cons(name)}` annotation. SDF supports lexical and context-free productions in a single specification. The specification may be organized into modules. Languages can be composed by means of imports. In our example, WebDSL imports a separate definition for access control and reuses the Hibernate Query Language (HQL) for embedded queries. To avoid any conflicts in symbol naming, the HQL language symbols (defined by others) are renamed in the context of this grammar using the suffix `[[HQL]]`. We use this in the last two productions in Figure 2 to combine the two languages: HQL expressions can be used directly as WebDSL expressions, while WebDSL expressions in HQL are prefixed with a tilde.

```
module WebDSL
imports MixHQL[HQL] AccessControl ...
exports
  context-free start-symbols
    Start
  lexical syntax
    [a-zA-Z][a-zA-Z0-9\_]* -> Id
    ...
  context-free syntax
    "module" Id Section*           -> Start      {cons("Module")}
    "section" SectionName Def*     -> Section    {cons("Section")}
    "define" Mod* Id "{" Element* "}" -> Def     {cons("SimpleDef")}
    ...
    Exp[[HQL]]                     -> Exp        {cons("ToHQL")}
    "~" Exp                        -> Exp[[HQL]] {cons("FromHQL")}
```

Fig. 2. SDF syntax rules for the WebDSL language.

```
module WebDSL
imports
  WebDSL-Syntax   WebDSL-Colorer   WebDSL-Folding      WebDSL-Outliner
  WebDSL-Analysis WebDSL-References WebDSL-Occurrences
language description and parsing
  name          : WebDSL
  id            : org.strategoxt.imp.generated.webdsl
  description   : "Spoofax/IMP-generated editor for the WebDSL language"
  extensions    : app
  table         : include/WebDSL.tbl
  start symbols : Start
  url           : http://www.webdsl.org/
```

Fig. 3. The default main editor service descriptor for WebDSL.

2.1 Editor Services Composition

The main editor service descriptor module imports all service descriptors for a given language. Figure 3 shows the main module for the WebDSL language. This module can be automatically generated by specifying a language name, the SDF grammar and start production, and the file extensions of the language. Once generated, it may be customized by the developer (in our example, we added the address of the WebDSL website). Similarly, default files for the other services are generated and imported into the main module.

Each type of service, e.g. outline, folding, coloring, is defined by its own declarative description language. All editor service descriptor languages share the notion of section headings for structuring the module, such as the `language` section in this example. These specify the kind of editor service described and can optionally contain human-readable text describing the part of the file.

Since the grammar is declaratively specified using SDF, we can programmatically analyze its structure. Using a set of heuristics, our system can derive default syntactic editor service descriptors for any given grammar. For example, based on the tree structure of the grammar of Figure 2, any `Module` node will be displayed in the outline, as it contains both a lexical string and a list of subnodes. Derived rules are placed in files clearly marked "generated" (e.g., `WebDSL-Colorer.generated`), so as to avoid confusion with handwritten code. Of course, deriving editor services based on a set of heuristic rules is never perfect, and may not suit a particular language or one's taste. Still, derived services form a good starting point for creating

```
module WebDSL-Colorer.generated
// ...documentation...
colorer default highlighting rules
  keyword : "Keywords" = magenta bold
  string  : "Strings"  = blue
  number  : "Numbers"  = darkgreen
  ...
colorer system colors
  darkred = 128 0 0
```

Fig. 4. The default WebDSL colorer.

```
module WebDSL-Folding
imports WebDSL-Folding.generated
folding additions
  Section._
  Def.SimpleDef
  Def.Imports (folded)
```

Fig. 5. Folding rules for WebDSL.

a new editor. Firstly, they can form the basis for a customized editor service. In particular, using editor service composition, generated editor services can be reused and customized as desired. Secondly, they quickly explain the syntax of the editor service descriptors, by means of generated, inline documentation, and by showing relevant examples that apply to the target language.

2.2 Syntax Highlighting

The most basic and perhaps one of the most essential editor services provided by an IDE is syntax highlighting. In SPOOFAX/IMP, syntax highlighting is based on the grammar of the language, rather than just a list of keywords. Thus, we correctly highlight tokens that are only considered keywords in parts of a composed grammar (such as `from`, which is a keyword in HQL but not in WebDSL). Using the grammar also enables us to report syntax errors inline.

SPOOFAX/IMP derives a default syntax highlighting service, based on the lexical patterns of a grammar. Figure 4 shows (fragments from) the default coloring specification for WebDSL. Each rule matches based on a lexical *token kind*, such as "keyword" for literals in the grammar, and "string" for lexical patterns with spaces. However, since SDF is scannerless, lexical tokens and productions are treated uniformly—they are just symbols. Therefore, rules can also match on productions or production types. Each rule specifies an (optional) human-readable description for display in the IDE's preference menus, together with the default color and formatting that should be applied to matching nodes.

For WebDSL, we were mostly satisfied with the default coloring rules, but wanted to emphasize sections and WebDSL expressions in HQL fragments. Therefore, we added the following rules to the `WebDSL-Colorer` module:

```
coloring customization of the default colorer
  Section._             : _ bold italic
  environment _.FromHQL : _ italic
```

The first rule specifies that terminal symbols for any production of type `Section` should be bold and italic, and use the default color (`_`) specified elsewhere in the descriptor. The second rule is an *environment rule*, and specifies that *all* nodes below a `FromHQL` node must be displayed in italics (such as ˜auhtor [*sic*] in Figure 1).

```
module WebDSL-Syntax.generated
// ...documentation...
language default syntax properties
  line comment  : "//"
  block comment : "/*" * "*/"
  fences        : [ ] ( ) { } |[ ]|
```

Fig. 6. WebDSL syntax properties.

```
[ Application -- V[ H [ "application"] _1] _2],
  Section     -- V is=2 [ H [ "section" _1]] _2],
  SimpleDef   -- V [ V is=2 [ H ["define" _1 _2
                      "{"] _3 ]
                      "}" ],
  ...
]
```

Fig. 7. Pretty printing rules for WebDSL.

2.3 Code Folding and Outline Display

The *outline view* provides a structural overview of a program and allows for quick navigation. Similarly, *code folding* uses the structure of a program to selectively hide code fragments for readability. Figure 5 shows an example folding definition. The specification imports the generated folding file, adding three new rules. The last rule specifies that all import declarations should be folded by default.

2.4 Braces, Comments, and Source Code Formatting

Brace matching and selection commenting is supported through the syntax properties service, shown in Figure 6. Following the IMP framework, these features are implemented at the character level, since they must operate regardless of malformed or unclosed comments and parentheses. Thus, the descriptor specifies which pairs of strings make up parentheses, and which form comments.

Code fragments can be formatted using the Box formatting language [20]. Box allows for very flexible formatting specifications based on nested layout boxes. It supports indentation and table-based formatting. We provide this as an editor service to format code in a subject language. Figure 7 illustrates some example formatting rules, including vertical (**V**) and horizontal (**H**) boxes. These rules format WebDSL code similar to the code in Figure 1. Using the generic pretty printer package [6], part of the Stratego/XT toolset [2], we can automatically derive pretty printers from grammars.

3 Semantic Editor Services

Semantic editor services include highlighting of semantic errors, providing type information for a selected expression, and looking up the declaration of an identifier. Since these services rely on semantic analysis, our approach aims at maximal reuse of any analysis components provided by a compiler for the targeted language.

WebDSL has been implemented using Stratego/XT [2], a program transformation language and toolset that uses SDF grammars for syntax definition. The Stratego transformation language combines term-based rewriting with strategies for specifying traversals on abstract syntax trees (ASTs). Trees in Stratego are represented as first-order terms. For example, represented as a term, the AST of the program in Figure 1 is `Module("author", [Imports(...), Section(...,...)])`.

```
module WebDSL-Analysis

analysis providers and observer
  provider webdsl-front.ctree
  provider WebDSL-pretty.pp.af
  observer: editor-analyze
```

Fig. 8. Semantic analysis bindings.

```
module WebDSL-References

references tree nodes with resolvable references
  reference Call: function-resolve function-info
  reference FieldAccess: field-resolve field-info
  ...
```

Fig. 9. The reference resolving descriptor.

3.1   Errors, Warnings, and Info Markers

We use Stratego specifications for expressing semantic editor services. Descriptors are used to bind the IDE to the Stratego specifications. The descriptor of Figure 8 describes the interface with the type checker. It specifies all external components used to provide the semantic services: Stratego modules compiled to the Stratego core language (`.ctree` files), and a WebDSL pretty printer (`.af` files). It also specifies the *observer* function, which is notified if there are any changes to WebDSL files. This function is implemented by a Stratego rule of the following form:

```
editor-analyze:
  (ast, path, fullpath) -> (errors, warnings, infos)
  where ...
```

This rewrite rule is given a tuple with the abstract syntax tree of the modified file, its project-relative path, and an absolute file system path; as a result it can produce lists of errors, warnings and info markers. For example, for our example of Figure 1, it returned a tuple of the form:

```
([(Var("auhtor"), "Variable auhtor not defined"), ...], [], [])
```

As the first element, the errors list contains a tuple with an error message for the misspelled "auhtor" variable, which is included in the form of a copy of the original node in the tree. This simple, but effective interface allows Stratego functions to return data and reference nodes of the abstract syntax tree. We discuss the implementation of this in Section 3.3.

3.2   Declarations and References

Reference resolving is a feature where an IDE can find the declaration site for an identifier, accessible in Eclipse as hyperlinks for identifiers when holding the control key (see Figure 1). Similarly, occurrence highlighting highlights all occurrences of an identifier when the cursor is over one of them. Figure 9 shows the reference resolving descriptor. For `Call` or `FieldAccess` nodes, it specifies the Stratego rules that retrieve their declaration site and a description for use in tooltips (see Figure 1). These rules can be implemented using a lookup table of identifiers to declarations (using dynamic rules [2]), similar to the typing rules in [9].

3.3   Implementation and Tool Integration

The main challenge of integrating IMP, Stratego and SGLR is having all tools working on the same set of data – AST nodes and tokens – at the same time. IMP defines standard interfaces for abstract syntax trees and lexical tokens. SGLR and Stratego represent trees as terms, using their own term interfaces. SGLR is

scannerless: it parses individual characters and uses no tokenizer. Tokens in the traditional sense are not present. Reconciling these differences is done in several ways. First, we use JSGLR, our own Java implementation of SGLR, and Stratego/J, a Stratego interpreter written in Java. The parse trees built by JSGLR are converted to new objects that implement the IMP interfaces. This conversion creates an IMP AST from the JSGLR parse tree, and maps literals and terminals in the parse tree to IMP tokens. The IMP AST nodes maintain links to tokens, which in turn know their exact source location. This is essential for editor services such as reference resolving or error markers that use the source locations of nodes in the tree. For integrating IMP ASTs with Stratego, we employ a variant of the program object model (POM) adapter approach [10]; our implementation of the IMP AST interface also implements the term interface necessary for Stratego rewriting.

Program analyses are implemented as Stratego transformations, and some transformations require the ability to construct AST nodes as part of an analysis. This is provided by the POM adapter, which supports node construction through an AST node factory. When Stratego programs transform trees, care must be taken that no position information is lost. For example, in the WebDSL typechecker, identifiers are given new, unique names. This means that each identifier tree node is replaced by a new value, and its parent node is transformed to one with a new set of children. Normally, tree nodes replaced in this fashion lose their associated position information. To avoid this, we made some extensions to the POM factory design for dealing with origin tracking [21] throughout the rewriting process. The factory now has added methods for supporting the primitive term traversal operators `one`, `some`, and `all`. Origin information for a term visited by a primitive traversal operator is propagated to the result after rewriting, similar to what is described in [21]: when a rewrite rule is applied to all children of a node (e.g., using the `all` operator), the origins for the old children are propagated to the new children. For the WebDSL typechecker example, this origin tracking scheme ensures that although the typechecker replaces several nodes in the tree, the new nodes maintain references to the original locations, providing proper position information for the editor services. Together with annotation support, origin tracking has proven sufficient for correct propagation of origin information during term rewriting.

4 Composition of Editor Services

The need for composing editor services arises directly from the composition required for language embedding and language extension. Consider the WebDSL generator, which is implemented in Stratego and includes embedded fragments of WebDSL, Java, and XML used for pattern matching and code generation [9]. The bottom pane of Figure 10 shows part of this generator, with WebDSL code fragments enclosed by |[ ]| in the Stratego code. WebDSL identifiers are italicized, and the remaining WebDSL code has a dark background.

SDF offers a uniform mechanism for composing grammars from modules. This mechanism forms the basis for both language extension and embedding. When a

```
module Stratego-WebDSL
imports Stratego[Stratego] WebDSL[WebDSL] ...
colorer add a background color to quoted code fragments
  environment Term[[Stratego]].ToMetaExpr:     _         metabg
              Term[[Stratego]].ToMetaExpr:     _         white  bold
  environment Term[[Stratego]].FromMetaExpr: _         white
  environment var:                           darkcyan  white  italic
  metabg = 210 230 220
```

```
type-of: |[ x := e ]| -> SimpleSort("Void")

derive-input: |[ output(e ) ]| -> |[ outputString(e) ]| where <type-of> e => SimpleSort("String")

constraint-error: Var(x) -> Error(["Variable ", x, " not defined"]) where not(type-of) // x has no type
```

Fig. 10. A colorer specification (top) and composed editor instance (bottom) for the embedding of WebDSL in Stratego.

given SDF module imports another module, it essentially adds all grammar productions of the imported module to its own set of productions. Multiple productions for the same non-terminal are allowed; all are kept (this may give rise to ambiguities, discussed later). In the case of language embedding, suffixing (cf. Figure 2) is used to distinguish productions for the embedded language from the host language.

SPOOFAX/IMP mirrors the SDF composition mechanism – modules and imports with suffixing – but the mechanism for resolving "overlapping" definitions is different. Figure 10 defines the composition of WebDSL editor services with those of Stratego. Stratego grammar symbols are suffixed with [[Stratego]], the WebDSL symbols with [[WebDSL]]. The first colorer rule gives WebDSL code fragments the dark background color, i.e. the text corresponding to all nodes under a ToMetaExpr node should be colored. The second rule sets a white background only for nodes *directly* beneath the ToMetaExpr node. Because this rule is more specific (i.e., applies to fewer nodes), the |[ ]| parentheses in the screenshot of the bottom pane have a white background, while the enclosed code has a dark background. The last two rules apply a white background to Stratego inside WebDSL fragments.

The rules for coloring, folding and outlining are all purely declarative, and share the same composition mechanics: we give priority to rules with more specific patterns, and rules defined later in the specification. New definitions can use the _ operator to set a property to the default as specified in other rules. Finally, definitions can be disabled using a disable annotation.

Not all aspects of the editor services are equally composable at present. Some pretty-printer formalisms are closed under composition [6], including the one we employ. However, our present DSLs do not expose this. The most difficult services to compose are those relating to language semantics: type checking and reference resolution. A good composition formalism for language semantics is still an open research topic. The solutions offered by SPOOFAX/IMP include the use of Stratego for specifying type checking and name resolution rules in a relatively declarative style, and the use of Java for exploiting existing language infrastructure.

5   Integration with Eclipse and IMP

The Eclipse architecture is based around the OSGi component model, where each plugin is (usually) a JAR containing Java classes, a plugin manifest, optional descriptor files, and auxiliary resources, such as images. The descriptors specify which parts of the Eclipse framework a given plugin extends, and which parts of the plugin may be extended by other plugins.

IMP [5] extends Eclipse with a high-level framework and specialized tooling for developing IDE support for new languages. Developed by IBM, it is currently being used for IBM-built DSLs, such as the X10 concurrent programming language. IMP provides wizards for quickly generating initial implementations of services in terms of the IMP framework. The usual caveat applies: any user-customizations are overwritten if the wizard is run again.

SPOOFAX/IMP uses the IMP framework for implementing editor services. As an alternative to the IMP wizards, we offer DSLs which better handle language evolution and compositionality of services. By clearly separating generated code from user-written code, we avoid any risk of overwriting existing customizations as changes are made to an evolving language, or when new services are derived. While IMP makes use of uncomposable visitors for many editor services, SPOOFAX/IMP abstracts over these with DSLs. As language extensions and embeddings are composed with the subject language, the necessary visitors are automatically provided.

The OSGi model implies distributing plugins as static JARs. During development, changes made to a plugin are, in general, not reflected in the active Eclipse instance. To avoid launching a new Eclipse instance to use a new or updated editor, we use interpretation of editor services to allow dynamic loading of services into the current instance of Eclipse. For example, for the syntax highlighting specification, SPOOFAX/IMP maintains hash tables that specify which formatting style to apply to which kind of token. This approach leads to a shorter development cycle than would be possible using code generation alone. For deployment, all required files for plugin distribution are still generated.

The implementation of dynamic loading relies on the IMP notion of language inheritance. Languages may inherit services from another language. If a particular service is not available for a language, IMP tries to use the service for the parent language. We defined a new, top-level language *DynamicRoot*, from which all SPOOFAX/IMP languages inherit. Our root language implements the standard IMP extension points by providing proxy stubs. These proxies dynamically load or update the actual editor services as required. For instance, the syntax highlighting service is implemented by a class `DynamicColorer`, which initializes and updates the proper concrete `Colorer` class on demand.

Wizards and DSLs may act synergistically. Using language inheritance, editor services provided by SPOOFAX/IMP can be overridden by custom-built, Java-based services. E.g., using a handwritten outliner service only requires the use of the IMP outliner wizard to create a class overriding the standard `DynamicRoot` service.

6   Related Work and Discussion

A large body of research on editor construction and generation exists, a significant portion of which is dedicated to structured, or syntax-directed, editors. Notable early examples include the Program Synthesizer (PG), a structured program editor, and the Synthesizer Generator [16] (SG), a generator for program editors (program synthesizers). Both SG and Spoofax/IMP provide composable language specifications, and therefore support language extensions and language families. In SG, languages are defined as abstract syntax with attribute grammars describing the type rules. SG is agnostic about obtaining ASTs, and provides no solution for composing concrete language syntaxes. PG imposes a strictly (classical) syntax-directed editing style: programs are built from templates filled in by the user. This style ensures syntactically valid programs, but never gained widespread acceptance.

The Meta-Environment is a framework for language development, source code analysis, and source code transformation [17,18]. It includes SDF, the ASF term rewriting language, and provides an IDE framework written in Java. Basic syntax highlighting is derived from SDF grammars. Coloring may be customized similar to the `environment` construct in Figure 10. ASF tree-traversal may also be used to annotate the AST with coloring directives. ASF is also used to specify the language type rules, and may include custom error messages, presented in a window similar to Figure 1. The IDE framework provides outlining but no folding or crossreferences. The Meta-Environment is presently being integrated into Eclipse.

MontiCore [14] and openArchitectureWare [7] (oAW) are tools for generating Eclipse-based editor plugins for DSLs. Both provide EBNF-like grammar formalisms which may be composed using inheritance (MontiCore) or module imports (oAW). ANTLR parsers are generated from the grammars. In MontiCore, basic editor presentation are included as grammar properties. Syntax coloring is specified as lists of keywords to highlight. Pre-defined (Java-style) comments are supported. Folding is specified by a list of non-terminals. For semantic editor services, MontiCore grammars specify events, which may be specialized with user-defined Java classes. Embedded languages are supported in MontiCore through "external" symbols in the grammar. An inheriting grammar module can implement these external symbols. The composed grammar is parsed by dynamically switching to the respective parser and lexer of one of the constituent grammars, depending on the current state of the parser. In oAW, an EMF [4] meta-model is generated in addition to the parser. Language semantics is expressed as constraints upon this model, either using an OCL-like language, or, optionally, using Java. oAW does not support embedded languages. For Spoofax/IMP, we provide an interface to Stratego, based on the POM adapter approach of [10] and origin tracking [21] techniques to handle semantics. Spoofax/IMP supports embedded languages.

Although the abstract syntax formalisms in MontiCore and oAW are both modular, the concrete syntax is limited by ANTLR grammar composition ability — LL(k) in the case of MontiCore, LL(*) for oAW, owing to the different versions of ANTLR employed. GLR parsing is closed under composition and allows grammars to be composed freely. Multiple, possibly conflicting, definitions of the same

non-terminal are allowed. This leads to the common criticism of GLR parsing that the resulting parse tree may contain ambiguities. This criticism is warranted, as ambiguities in grammars are by their nature undesirable, and not always trivial to discover, especially in composed grammars. Using explicit, declarative restrictions and priority orderings on the grammar, these can be resolved [19]. Other methods, including PEGs [8], language inheritance in MontiCore, and the composite grammars of ANTLR, avoid this by forcing an ordering on the alternatives of a production – the first (or last) definition overrides the others. The implicit ordering may not be the intention of the grammar engineer, and can be hard to debug as discarded alternatives cannot be inspected. In contrast, the GLR approach readily displays these alternatives. Using parser unit tests, ambiguities following from design mistakes, omissions, or regressions, can be avoided. The few remaining – intended – ambiguities are then handled immediately after parse time, when additional context information is available. SPOOFAX/IMP deals with ambiguous parse trees using a fixed pruning strategy (pick leftmost alternative from any ambiguous subtrees), which allows it to work with grammars under development.

Until recently, a drawback of SGLR was its lack of error recovery: in addition to reporting parsing errors, the parser should also try to parse the remainder of a file to ensure (partial) functionality of editor services. We recently addressed this issue and made a prototype implementation of JSGLR with error recovery [11].

7   Conclusion and Future Work

Providing high-quality IDE support has become paramount to the success of a new programming language. The implementation effort required for a solid, custom-built IDE is often prohibitive, in particular for domain-specific languages, language extensions, and embedded languages, with relatively small userbases. We have presented a partial solution to this, SPOOFAX/IMP, a meta-tooling suite for rapidly implementing editors for Eclipse. We have shown several techniques that contribute towards efficient development of IDE support. First, we provide high-level, declarative DSLs for defining editor services that abstract over the IMP meta-tooling framework, and allow reuse of previously defined services through composition. Second, SPOOFAX/IMP uses heuristics to automatically derive common editor services from the grammar, services which may be adapted and co-evolved with the grammar without any trouble related to modifying generated code. Third, we use the Stratego programming language for a high-level specification of both semantic analysis and compilation. Finally, we support dynamic loading of services into the active Eclipse environment, leading to a shorter development cycle.

Some open problems remain. We intend to replace the Stratego interpreter with a compiler (targeting the JVM) to address performance concerns, while maintaining the flexible interface of primitive operations as defined by the extended term factory. Similarly, we want to optimize the JSGLR parser for use in an interactive environment. The current prototype performs poorly, and, ideally, it would support incremental parsing. However, as it runs in a background thread, this has relatively

little impact on the user experience.

We previously reported on a predecessor of Spoofax/IMP [12]. We significantly expanded its support for language composition since, and added support for semantic analysis and dynamic loading of editor services at runtime. In part, these changes were driven by experience from the WebDSL case studies briefly reported on in this paper. In other previous work, we have applied strategic programming in the field of attribute grammars, allowing high-level, declarative specifications of semantic analyses [13]. In the future, we want to use these as a basis for defining semantic editor services. In particular, we want to investigate the feasibility of using such specifications for achieving full compositionality of complete syntactic, semantic, and tooling descriptions of language extensions.

References

[1] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In J. Lawall, editor, *Generative Programming and Component Engineering (GPCE 2007)*, pages 3–12. ACM, October 2007.

[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[3] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *OOPSLA 2004*, pages 365–383. ACM Press, October 2004.

[4] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[5] P. Charles, R. M. Fuhrer, and S. M. Sutton, Jr. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE 2007*, pages 485–488. ACM, 2007.

[6] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *The International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, 2000.

[7] S. Efftinge et al. openArchitectureWare User Guide. Version 4.3. Available from http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/, April 2008.

[8] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *ICFP 2002*, volume 37 of *SIGPLAN Notices*, pages 36–47. ACM, October 2002.

[9] Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and A. Vallecillo, editors, *Theory and Practice of Model Transformations. First International Conference on Model Transformation (ICMT 2008)*, volume 5063 of *LNCS*, pages 183–198. Springer, July 2008.

[10] K. T. Kalleberg and E. Visser. Fusing a transformation language with an open compiler. In A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *ENTCS*, pages 21–36. Elsevier, April 2008.

[11] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In G. T. Leavens, editor, *OOPSLA 2009*, ACM SIGPLAN Notices, New York, NY, USA, October 2009. ACM Press.

[12] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Generating editors for embedded languages. integrating SGLR into IMP. In A. Johnstone and J. Vinju, editors, *Proceedings of the Eighth Workshop on Language Descriptions, Tools, and Applications (LDTA 2008)*, April 2008.

[13] L. C. L. Kats, A. M. Sloane, and E. Visser. Decorated attribute grammars. Attribute evaluation meets strategic programming. In *Compiler Construction (CC 2009)*, volume 5501 of *LNCS*, pages 142–157. Springer-Verlag, March 2009.

[14] H. Krahn, B. Rumpe, and S. Völkel. Efficient editor generation for compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, technical report TR-38, pages 218–228. University of Jyväskylä, 2007.

[15] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: Modular development of textual domain specific languages. In R. Paige and B. Meyer, editors, *TOOLS EUROPE 2008*, volume 11 of *LNCS*, pages 297–315. Springer-Verlag, June 2008.

[16] T. Reps and T. Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, 1984.

[17] M. G. J. van den Brand. Applications of the Asf+Sdf Meta-Environment. In R. Lämmel, J. Saraiva, and J. Visser, editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2006.

[18] M. G. J. van den Brand, M. Bruntink, G. R. Economopoulos, H. A. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. J. Vinju. Using the Meta-Environment for maintenance and renovation. In *The European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 331–332. IEEE Computer Society, 2007.

[19] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC 2002)*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, April 2002.

[20] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. on Softw. Eng. and Methodology*, 5(1):1–41, January 1996.

[21] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545, 1993.

[22] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *LNCS*, pages 291–373. Springer, October 2008.