Delft University of Technology
Software Engineering Research Group
Technical Report Series

Generating Editors for Embedded Languages

Lennart Kats, Karl Trygve Kalleberg, Eelco Visser

Report TUD-SERG-2008-006

TUD-SERG-2008-006

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Generating Editors for Embedded Languages

Integrating SGLR into IMP

Lennart Kats[1], Karl Trygve Kalleberg[2] and Eelco Visser[3]

[1,3] *Delft University of Technology, The Netherlands* [1]L.C.L.Kats@tudelft.nl, [3]visser@acm.org
[2] *University of Bergen, Norway* [2]karltk@ii.uib.no

Abstract

Integrated Development Environments (IDEs) increase productivity by providing a rich user interface and rapid feedback for a specific language. Creating an editor for a specific language is not a trivial undertaking, and is a cumbersome task even when working with an extensible framework such as Eclipse. A new IBM-guided effort, the IMP framework, relieves the IDE developer from a significant portion of the required work by providing various abstractions for this. For embedded languages, such as embedded regular expressions, SQL queries, or code generation templates, its LALR parser generator falls short, however. Scannerless parsing with SGLR enables concise, modular definition of such languages. In this paper, we present an integration of SGLR into IMP, demonstrating that a scannerless parser can be successfully integrated into an IDE. Given an SDF syntax definition, the SDF2IMP tool automatically generates an editor plugin based on the IMP API, complete with syntax checking, syntax highlighting, outline view, and code folding. Using declarative domain-specific languages, these services can be customized, and using the IMP metatooling framework it can be extended with other features.

1 Language-Specific Editors

Integrated Development Environments (IDEs) increase developer productivity by providing a rich user interface and rapid feedback for a specific language. Editors increase code readability by applying code folding (hiding details and boilerplate code), or by use of syntax highlighting [1]. In-place reporting of errors and warnings, as well as different views of a program further help understandability. Many IDEs provide various ways of navigation through the structure of a program by providing cross-referencing support or an outline view. Modern IDEs are also used as front-ends for program refactorings and analyses.

Software projects are typically developed in multiple languages, ranging from mainstream languages, such as Java and C#, to custom languages, such as embedded and domain-specific languages (DSLs). As programmers grow accustomed to IDEs for mainstream languages, they come to expect the same level of IDE support for custom languages. Thus, providing a good IDE becomes a significant factor in the success of a language. Building a custom, state-of-the-art IDE, however, is a

time-consuming and difficult undertaking. Other infrastructure, such as compilers, usually takes precedence over editors. As DSLs evolve rapidly, accommodating new domain insights and adapting to circumstances, extra effort is required to keep the editor up-to-date. For these reasons, editor construction is often neglected.

Extensible development platforms, such as the Java-based Eclipse platform, are a solid base for editor developers. Eclipse offers a single IDE interface for editing multiple file types. It provides various facilities that form a basis for full-functional IDEs for custom languages, such as integrated build and version management systems. However, creating an Eclipse editor plugin is currently a rather cumbersome task, because the Eclipse API is relatively low-level and changes slightly between releases as Eclipse continues to evolve. The IMP project (formerly known as SAFARI [3]) aims at providing abstractions for creating Eclipse IDE editor plugins. Currently in an alpha stage, it provides basic, modular editor services (i.e., features) in the form of Java classes that can be generated using wizards.

Despite extensible development platforms, a particular class of custom languages that often lacks IDE support, is that of embedded languages. Examples include database query languages embedded into a general-purpose language, where the embedding increases the expressivity of the host language and enables static checking of queries. For code generation, embedded object language quotations are applied in meta languages such as Stratego [8]. IMP currently employs the LALR parser generator LPG, formerly known as JikesPG [3]. LALR is not closed under composition and requires building embedded languages using a specially crafted scanner for the embedding. In contrast, the scannerless generalized LR parsing algorithm (SGLR) does support the full class of context-free grammars, which is closed under composition [2]. SGLR has been used in language definitions and tools for Java, C, PHP, as well as embeddings based on these languages [8].

In this paper, we present SDF2IMP, a generator for Eclipse editor plugins that integrates SGLR into the IMP framework. Given an SDF syntax definition of a language, it automatically creates the complete source of an editor plugin. Using declarative languages to define *editor service descriptors*, the generation process may be further customized. At this point, the generated editor includes in-line reporting of parse errors, syntax highlighting, code folding, and an outline view. By building upon the IMP framework, the editor can be extended with IMP-based editor services that may be generated with the wizards provided by IMP.

Our implementation gives insight into using a custom grammar formalism within IMP. While IMP aims at eventually providing support for custom parsers, the current (alpha) implementation makes several assumptions about the parser implementation. It defines a number of low-level interfaces for the parser components, such as the parser itself, the lexer, lexer stream and the token stream. These do not always match with a given parser implementation, of which SGLR, not employing a lexer, is a good example. IMP also uses a number of concrete LPG classes for some of the vital components, further hindering the compatibility with other parsers. Ideally, it would instead define a more high-level interface that exposes minimal implementation details.

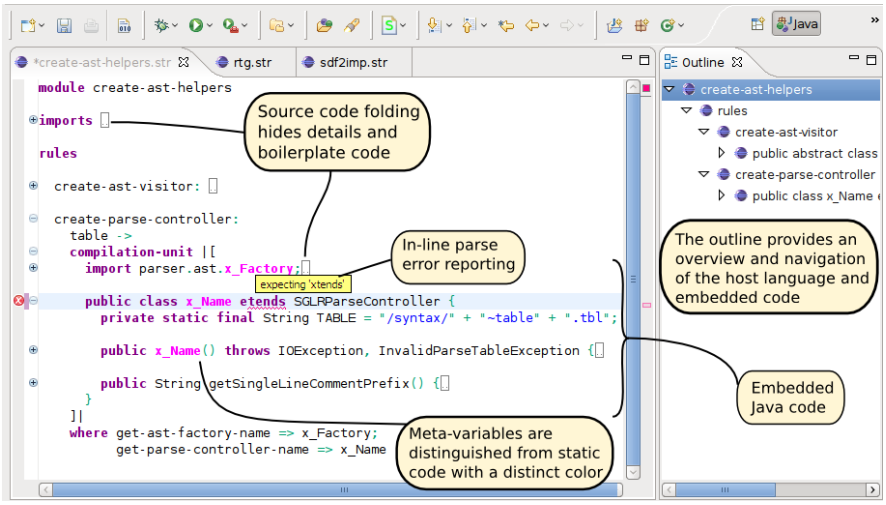

Fig. 1. Screenshot of an editor for Stratego with embedded (quoted) Java code

2 Generating Editors

Given a syntax definition, SDF2IMP generates strongly typed abstract syntax tree (AST) classes, as well as an associated visitor and factory implementation. These classes are used by IMP as a basis for all editor services. By defining standard interfaces for these common editor components, IMP lifts the editor implementation to a higher level of abstraction than possible with the regular Eclipse API. Using heuristics, SDF2IMP further generates a number of IMP-based editor services from the grammar. These can be customized using high-level editor service descriptor languages, but offers sane defaults for many languages.

In addition to the generator, SDF2IMP also includes a runtime component, which makes use of JSGLR, a Java implementation of SGLR. When parsing a file, SGLR produces a parse tree containing all input characters, as well as the applicable parsing productions required to produce an AST. As SGLR does not employ a separate scanner, tokens must be subsequently extracted from this tree to produce a token stream. This is handled by the SDF2IMP runtime compatibility layer, which traverses the parse tree to extract tokens and AST nodes. The strongly typed AST nodes are created using the generated AST factory, and include a reference to the tokens they are associated with.

We have applied SDF2IMP to a number of languages, including WebDSL (a domain-specific language that embeds HQL), AspectJ (which has a notoriously difficult lexical syntax), and a number of embedded languages based on PHP, Java, and Stratego [8]. Figure 1 shows the editor in action.

As JSGLR is based on Java, it integrates well into Eclipse and is platform independent. Any *parse errors* encountered are thrown as exceptions, and are directly reported in-line in the editor, providing the user with rapid feedback.

Syntax highlighting in IMP is based on mapping fonts and colors to tokens, depending on their *token kind*. The kind of a token is defined in the scanner definition. Since no such notion exists in Scannerless GLR, we have taken a somewhat different

approach. Based on the production rules of a grammar, we automatically assign *generic token kinds* to different tokens. For example, lexical production tokens may be categorized as Strings, numbers, or identifiers, based on the parsing pattern. A unique token kind is also reserved for *meta variables*, which exist as first class citizens in SDF2 syntax definitions. By default, syntax highlighting operates purely on the basis of generic token kinds, which is demonstrated in Figure 1. It may be customized using an editor service descriptor that allows different colors to be assigned to keywords (actually, production literals), sorts, and generic token kinds.

The *outline view* provides a structural view of a program and allows for quick navigation. Similarly, *code folding* operates on the structure of a program to selectively hide code fragments for readability. In particular, this also works in embedded code, such as a list of imports in a Java code fragment. Both services are based on the AST and internally use visitors to select applicable AST nodes. Using heuristics, we create default implementations for these services. For example, we include parse productions that have at least one subterm with an identifier generic sort and one list subterm. The editor service descriptors may indicate the production sorts to be included in the services. Names of outlined items can be indicated using pattern matching on the AST nodes.

Designed with modular, embedded language definitions in mind, the editor service descriptors provide a module system for reuse of descriptors. This would not be feasible without the abstraction from the visitor-based Java implementation.

3 Discussion

We have previously presented Spoofax [4], an editor for Stratego and SDF2, based directly on the Eclipse API. Its hand-written syntax analyzer makes use of context-sensitive, rule-based pattern matching rather than parsing. This ensures correct syntax highlighting for most cases of grammatically incorrect code. Two significant drawbacks, however, are the lack of interactive parser errors and incorrectly highlighted keywords of embedded languages. It is anticipated that the parser-based approach presented here will be integrated into Spoofax.

The Meta-Environment [7] is an extensible toolset for program analysis and transformation based around SDF that comes with its own extensible IDE framework written in Java. It provides a generic syntax highlighting facility based on SGLR. Our approach combines the advantages of SGLR with the robustness of Eclipse and the extensibility of IMP.

The MontiCore DSL development framework [5] is based on an extended grammar formalism which is used to specify both the concrete and abstract syntaxes of a language, as well as associations between AST nodes. Language extensibility is handled mainly through inheritance of grammar rules. The underlying parsing technology is predicated LL(k). The current prototype generates editor plugins for Eclipse that provide syntax highlighting, syntax (and some forms of semantic) error reporting and code outlining. In comparison, annotations in SDF grammars are used to automatically derive ASTs, and, thanks to IMP wizards, additional editor

services are easy to add with our approach.

JSGLR cannot currently be considered mature. In particular, it performs poorly, and, ideally, would support incremental parsing. However, as it runs in a background thread, this has very little impact on the user experience. A more problematic drawback is its lack of error recovery support: in addition to reporting parsing errors, the parser should also try to parse the remainder of the file to ensure at least partial functionality of any editor services. The IMP basis partly provides for this by ensuring that syntax highlighting and other services are maintained in unedited regions after an error occurs. This is an area that has not yet received much attention, but [6] shows promising results.

The use of strongly typed AST and visitor classes in IMP currently hampers run-time composition of embedded languages, e.g., enabling a Stratego editor for arbitrary object languages. It is possible to forgo generation of these classes and generate editor services that do not make use of them, but this would limit our compatibility with other IMP services. To maintain compatibility, architectural changes to IMP itself would be required.

In this paper, we presented language-specific editor features that can be derived from a grammar. Possible future extensions include grammar-aware brace matching and text selection, automatic commenting of regions, and extraction of metadata from comments. Code formatting tools that are already available for SDF could be integrated into the IDE. Another area of future work is that of integrated semantic analysis and refactoring, for which IMP already provides a basis. Specifically, it could benefit from the use of a specialized language such as Stratego for this.

References

[1] Baecker, R., *Enhancing program readability and comprehensibility with tools for program visualization*, in: *ICSE '88: Proceedings of the 10th international conference on Software engineering* (1988), pp. 356–366.

[2] Bravenboer, M. and E. Visser, *Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions*, in: D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)* (2004), pp. 365–383.

[3] Charles, P., J. Dolby, R. M. Fuhrer, J. Stanley M. Sutton and M. Vaziri, *Safari: a meta-tooling framework for generating language-specific IDE's*, in: *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2006), pp. 722–723.

[4] Kalleberg, K. T. and E. Visser, *Spoofax: An interactive development environment for program transformation with Stratego/XT*, in: A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, Braga, Portugal, 2007, pp. 47–50.

[5] Krahn, H., B. Rumpe and S. Völkel, *Efficient editor generation for compositional DSLs in Eclipse*, in: *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[6] Valkering, R., "Syntax Error Handling in Scannerless Generalized LR Parsers," Master's thesis, University of Amsterdam (2007).

[7] van den Brand, M. G. J., 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: *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering* (2007), pp. 331–332.

[8] Visser, E., *Meta-programming with concrete object syntax*, in: D. Batory, C. Consel and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, Lecture Notes in Computer Science **2487** (2002), pp. 299–315.

TUD-SERG-2008-006
ISSN 1872-5392