Delft University of Technology
Software Engineering Research Group
Technical Report Series

The Spoofax Language Workbench

Lennart C. L. Kats, Eelco Visser

Report TUD-SERG-2010-029

TUD-SERG-2010-029

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/

This paper is a pre-print of:

Lennart C. L. Kats, Eelco Visser. The Spoofax Language Workbench. In *Companion to the Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH 2010)*. ACM, 2010.

```
@inproceedings{KatsVisser2010companion,
  title = {The {Spoofax} Language Workbench},
  author = {Lennart C. L. Kats and Eelco Visser},
  year = {2010},
  booktitle = {Companion to the Conference on Systems,
               Programming, Languages, and Applications:
               Software for Humanity (SPLASH 2010)},
  publisher = {ACM},
}
```

The Spoofax Language Workbench

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. It provides a comprehensive environment that integrates syntax definition, program transformation, code generation, and declarative specification of IDE components.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Programming Environments

General Terms Languages

1. Introduction

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain. They provide linguistic abstractions and specialized syntax specifically designed for a domain, allowing developers to avoid boilerplate code and low-level implementation details.

The development of new DSLs comprises many tasks, ranging from syntax definition to code generation to the construction of an integrated development environment (IDE). Language engineering tools are essential for productivity in each of these tasks.

The Spoofax language workbench [2] is a platform for the development of textual domain-specific languages with state-of-the-art IDE support. Spoofax provides a comprehensive environment that integrates syntax definition, program transformation, code generation, and declarative specification of IDE components. The environment supports agile development of languages by allowing incremental, iterative development of languages and showing editors for the language under development alongside its definition (Figure 1). These editors can be used to view the abstract syntax of a program or to directly apply transformations on a selection of text.

Spoofax is based on Eclipse, an extensible programming environment that offers many language-generic development facilities such as plugins for version control, build management, and issue tracking. Spoofax language definitions take the form of Eclipse plugin projects, and can be distributed to "end developers" using the Eclipse update site mechanism.

2. Syntax Definition

The grammar forms the heart of the definition of any textual language. It specifies the concrete syntax (keywords etc.) and the abstract syntax (data structure for analysis and transformations) of a language. In Spoofax, the syntax is also used to derive customizable editor services, such as a default syntax highlighting service and an outline view service.

We use the modular syntax definition formalism SDF2 [3] for the specification of grammars. SDF grammars are highly modular, combine lexical and context-free syntax into one formalism, and can define concrete and abstract syntax together in production rules. Grammar productions in SDF take the form $p_1 \ldots p_n$ `->` s and specify that a sequence of strings matching symbols p_1 to p_n matches the symbol s. Productions can be annotated with a constructor name n to uniquely identify them in the abstract syntax using the `{cons(`n`)}` annotation. Other annotations include `{left}` and `{right}` to specify the associativity of operators, and `{deprecated}` to indicate deprecated syntax.

Figure 1 (left) shows the SDF syntax for a datamodeling language (Figure 1, upper right). The first production rule defines `Start`, the start symbol of the grammar. It matches the `module` keyword, followed by an identifier, and a list of `Definitions`. Each `Definition` is a database entity with an identifier name and a list of `Property` symbols.

3. Editor Services

Modern IDEs increase developer productivity by incorporating many different kinds of *editor services* specific to the syntax and semantics of a language. They assist developers in understanding and navigating through the code, they direct developers to inconsistent or incomplete areas of code, and they even help with editing code by providing automatic indentation, bracket insertion, and content completion. As a consequence, developers that have grown accustomed to these services are growing less accepting of languages that do not have solid IDE support.

Editor services have a prominent role in Spoofax and can be specified using declarative editor descriptor languages. Spoofax generates default, customizable editor service descriptors based on the syntax of the language. Figure 2 illustrates an editor descriptor for the syntax highlighting of the entity language. It consists of two modules: `EntityLang-Colorer`, which specifies a custom color for types, and `EntityLang-Colorer.generated`, which contains generated default colors. Other editor services follow the same pattern, combining custom specifications with generated specifications that are based on static defaults and heuristic rules, each specified in its own file.

4. Code Generation

We use the Stratego program transformation language [1] to describe the semantics of a language. Stratego is based on rewrite rules for first-order terms, and strategies that control the application of these rules. Basic rewrite rules take the form

r `:` t_1 `->` t_2 `where` s

with r the name of the rule, t_1 and t_2 first-order terms, and s an optional *strategy expression*. A rule applies to a term when its left-

SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
ACM 978-1-4503-0240-1/10/10.

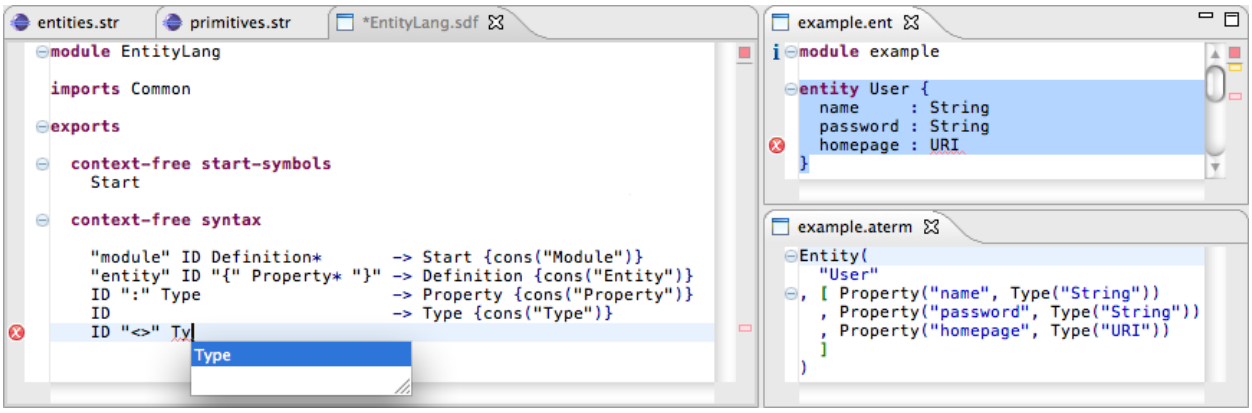

Figure 1. Multiple editors, side by side, in the same Eclipse IDE instance: the definition of an entity language (left), an editor for the entity language itself (upper right), and the abstract syntax of the selected entity (lower right).

hand side t_1 matches the term, and the condition s succeeds, resulting in the instantiation of the right-hand side pattern t_2. During development, the abstract syntax view can be used as a reference for the first-order term representation of a language's abstract syntax (lower-right of Figure 1).

Figure 3 shows rewrite rules that generate Java code for the entity language. These rules match against the abstract syntax of the language and generate string expressions for matching elements using the `$[...]` string interpolation syntax. String interpolation expressions construct a string of all literal characters between the quotes, except for escapes between `[...]`. Other transformation rules, not shown here, may rewrite to abstract syntax or may use syntax-checked concrete syntax expressions [4].

Code generation rules can be used to transform the DSL to a compilable form. They can be applied automatically as files are saved, or manually when triggered by the user. They can also be used to create *views* of the language, similar to the abstract syntax view of Figure 1. By default, views are automatically kept up-to-date and regenerated in the background as the source is changed.

5. Analysis and Transformation

Stratego rewrite rules are also used to specify semantic editor services, such as error markers, reference resolving, and content completion. Figure 4 shows two rules that check for semantic errors in the entity language. These rules use one or more conditions in their `where` clause to match elements of a program that contain errors.[1] At the right-hand side they include a tuple of the offending term – where the error marker would be placed in the editor – and the error message. As an example, the upper right editor of Figure 1 contains an error marker for the `"URI"` term in the User entity.

6. Conclusion

The Spoofax language workbench supports agile development of new programming languages by allowing selective, incremental development of editor services that can be dynamically loaded, evaluated, and tuned in the same environment. Using high-level languages to specify the syntax and semantics of a language, it provides a language development solution that greatly increases productivity of language engineers compared to using handwritten components or separate language engineering tools.

[1] For reasons of space, we do not include a full description and definition of these conditions here, but rather refer the reader to [2] for a comprehensive description of analyses and check rules.

```
module EntityLang-Colorer
imports EntityLang-Colorer.generated
colorer
  Type : blue

module EntityLang-Colorer.generated
colorer
  keyword    : magenta bold
  identifier : default
  string     : blue
  ...
```

Figure 2. Syntax highlighting rules for the entity language.

```
to-java:
  Entity(x, p*) ->
  $[ class [x] {
        [p2*]
     }
  ]
  where p2* := <to-java> p*

to-java:
  Property(x, Type(t)) -> $[
    private [t] [x];
    public [t] get_[x] { return [x]; }
    public void set_[x] ([t] [x]) { this.[x] = [x]; }
  ]
```

Figure 3. Code generation rules.

```
constraint-error:
  Property(x, Type(t)) -> (t, $[Unknown type [t]])
  where
    not(!t => "String");
    not(!t => "Int");
    not(<GetEntity> t)

constraint-warning:
  Entity(x, _) -> (x, $[Must start with a capital])
  where
    not(<string-starts-with-capital> x)
```

Figure 4. Semantic check rules for the entity language.

References

[1] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. of Comp. Programming*, 72(1-2):52–70, June 2008.

[2] L. C. L. Kats and E. Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In M. Rinard, editor, *OOPSLA 2010*. ACM, 2010.

[3] E. Visser. A family of syntax definition formalisms. Technical Report P9706, Progr. Research Group, University of Amsterdam, July 1997.

[4] E. Visser. Meta-programming with concrete object syntax. In *GPCE 2002*, volume 2487 of *LNCS*, pages 299–315. Springer-Verlag, 2002.

TUD-SERG-2010-029
ISSN 1872-5392