ESDF: A Proposal for a More Flexible SDF Handling

Akim Demaille Thomas Largillier Nicolas Pouillard

Epita Research and Development Laboratory (LRDE)
`http://transformers.lrde.epita.fr <transformers@lrde.epita.fr>`

Abstract

By the means on its annotations, Syntax Definition Formalism (SDF) *seems* to be extensible: the user is tempted to tailor its grammar syntax by adding new annotation kinds. Unfortunately the standard SDF crunching tools from Stratego/XT do not support the extension of SDF, and the user has to develop the whole set of tools for her home grown extension(s). We present the Extended SDF (ESDF) tool set that provides "weak" genericity with respect to the grammar grammar: support for arbitrary SDF annotations. We would like to contribute it to Stratego/XT since its components subsume their stock peers. Finally, we present a set of four extensions we find useful.

1 Introduction

SDF (Visser, 1995) is modular, and extensible thanks to its support for annotations. The combination of these two features makes SDF a unique place where additional grammar features can be added, immediately taking advantage of all the other SDF features. Examples of embeddable data include pretty-print directives, extended Abstract Syntax Tree (AST) generation directives, attribute rules, disambiguation tags, in addition to the "official" support for simple disambiguation filters, AST constructors, etc. With such self-contained ESDF files, mixing grammar modules is straightforward, and the user can focus on its extensions without having to bother with different concepts of modules.

Nevertheless aggregating several unrelated aspects of a grammar within a single file violates the principle of separation of concerns: a given facet of a grammar is surrounded with unrelated, distracting, material.

This contradiction is actually very comparable to a well-known object oriented design issue: given a family of related classes and a group of functions to implement on them, what is the best design? Classical Object Oriented Programming (OOP) recommends to implement the functions as methods. This eases the addition of new objects, but makes the addition of functions tedious: each class (read class implementation file) must be edited. The Design Pattern (DP) approach advocates the introduction of Visitors (objects that implement the functions) and the extension of the classes to cooperate with any kind of Visitor. The implementation of new functions is straightforward — implement new Visitors —, but the addition of new classes requires the edition of all the existing visitors. In fact there is not one right solution, both axis to read the matrix "classes $\times$ features" has its advantages: if the classes are stable and in fixed number, use Visitors (DP); conversely, if the functions are stable and in fixed number, sticks to methods[1] (OOP).

The same tension exists when working with grammars. If your grammar modules (classes) are versatile and numerous, then encapsulation (OOP) is more productive than separation of concerns: the grammar-centric vision is best suited. Conversely, stable grammars with well established features call for separation of concerns (DP): one feature should be isolated from unrelated issues (visitors), and the feature-centric approach is more suitable.

Because we work with several unstable grammar modules, because simple composition of modules eases our tasks, we wrote the ESDF set of tools to convert from the all-in-one paradigm to the separation-of-concern approach. ESDF should be enriched with reverse conversions,

[1]Actually this tension is at the origin of the inception of Aspect Oriented Programming (AOP), but it is unclear what the parallel would be in the context of SDF.

eventually providing the user with an easy means to zip and unzip grammar and grammar annotations.

We would like to contribute ESDF to Stratego/XT, since there is quite some code duplication between ESDF filters and their SDF peers, which they subsume. In the following, the components of ESDF are presented, and then a set of local SDF extensions we depend upon.

2 ESDF: An SDF Chain Robust to New Annotations

ESDF is a set of simple tools that provide generic support to SDF annotations.

2.1 Packing Modules: `pack-esdf`

The regular `pack-sdf` tool takes a grammar module as argument, gathers all its dependencies and produces a single big self-contained grammar file. If modularity was considered as syntactic sugar, then `pack-sdf` is its desugaring pass: none of the tools downstream need to support modularity. Unfortunately `pack-sdf` does not support annotation plug-ins: this is what `pack-esdf` addresses. It supports an additional option to be given the actual SDF grammar to use.

2.2 Filtering Annotations Out: `sdf-strip`

This simple tool strips (or preserves) selected annotations from a grammar. Of course, as `pack-esdf`, it needs to be given the SDF grammar used (unless it is the stock grammar).

2.3 Parsing Extended Grammars: `parse-esdf`

One would like to handle ESDF grammars seamlessly, like regular SDF grammars. Therefore it is the SDF parser that needs to be extended, or rather, extensible. Even the two aforementioned tools (`pack-esdf` and `sdf-strip`) need to parse ESDF grammars, and therefore demand a separated parser to avoid code duplication.

This tool is `parse-esdf`, an extension of `parse-sdf`. Based on the same ideas used to implement (foreign) concrete syntax within Stratego, `parse-esdf` looks for a `foo.meta` file for each `foo.sdf` file. This meta file describes the actual SDF grammar used.

This one tool factored several of the tools we had, since they all addressed the particular extension we were working on (BoxedSDF, DetGen etc.).

3 The lrde-syntax bundle

In addition to the general framework to extend SDF, we propose a set of specific extensions designed to support the grammar-centric vision.

3.1 Pretty-Printing: `boxedsdf`

Embedding the GPP pretty-printing tables in the grammar eases the maintenance, and provides a more comfortable environment to edit these tables: one can use names instead of numbers etc. See Fig. 1.

3.2 Disambiguation Tags: `sdf-detgen`

It is convenient, in particular to write disambiguation test cases or to check by human the result of a disambiguation pass, to enrich an ambiguous grammar with special comments to specify the correct alternative. For instance, in C++ `namespace A {}` is ambiguous: its actual nature depends whether the namespace name `A` was met for the first time (is "original") or not. Therefore parsing the following:

```
namespace A {}
namespace A {}
```

results in the following disambiguated text, printed with disambiguation comments (`org` stands for original, and `ns` for namespace):

```
namespace /*<org>*/A/*</org>*/ {}
namespace /*<ns>*/A/*</ns>*/ {}
```

The generation of such comments *and* the rules that recognize them is straightforward. The most adequate place to specify these comments is the grammar, as additional `dettag` annotations (Fig. 2).

```
%% 7.3.1 [namespace.def]
"namespace" Identifier "{" NamespaceBody "}" → OriginalNamespace
Definition
    {pp (V[H[KW["namespace"] Identifier]
           V is=2[KW["{"]
                  NamespaceBody]
           KW["}"]])}
```

In BoxedSDF, one can use symbol names to denote nonterminals (e.g., `Identifier` and `NamespaceBody` in the first rule) or labels, instead of `_1` as with GPP.

Figure 1: BoxedSDF sample

The following piece of C++ grammar extensions introduces special comments that can be used to disambiguate explicitly the "first occurrence of namespace name" issue.

```
"namespace" "/*<org>*/" Identifier "/*</org>*/" "{" NamespaceBody "}"
                                               → OriginalNamespaceDefinition
"/*<ns>*/" Identifier "/*</ns>*/"              → OriginalNamespaceName

"/*<org>*/" | "/*</org>*/"  → LAYOUT {reject}
"/*<ns>*/"  | "/*</ns>*/"   → LAYOUT {reject}
```

These rules were generated by `detgen` from the following annotated SDF rules. The first rules disambiguate the type names, and the last reject the parsing of the disambiguating tags as comments.

```
%% 7.3.1 [namespace.def]
Identifier → OriginalNamespaceName            {dettag("ns")}
"namespace" Identifier "{" NamespaceBody "}"
           → OriginalNamespaceDefinition      {dettag("org", 1)}
"namespace" OriginalNamespaceName "{" NamespaceBody "}"
           → ExtensionNamespaceDefinition
```

Figure 2: Disambiguation annotations.

3.3 Attribute Grammar (AG): sdf-attribute

A more ambitious extension of SDF consists in supporting AGs. Two companion papers present this topic in depth: Borghi et al. (2005) detail the evaluation mechanisms, and David et al. (2005) demonstrate how AGs can be used to disambiguate C and C++. A simple sample follows.

```
e1:Exp "+" e2:Exp → Exp
 {attributes(eval:
   root.value := <add> (e1.value,
                        e2.value))}
```

3.4 Flexible AST generation: sdf-astgen

The cons annotations relieves the SDF user from having to implement an abstract syntax grammar: it is extracted from the concrete grammar. As long as the concrete syntax is "natural", the resulting ASTs are lightweight and pleasant to process. But if the grammar is entangled with Yacc idiosyncrasies, or disambiguates "by hand" instead of relying on precedence and associativity directives, then the ASTs look like Parse Trees (PTs)... Some advocate the de-Yaccification of the grammar, but when this is not possible or desirable, one would like a more powerful cons annotation.

`sdf-astgen` allows the user to specify her abstract syntax, using an extended cons an-

```
e:Expr "+" t:Term -> Expr  { ast(BinOp(Plus, e, t)) }
t:Term            -> Expr  { ast(t) }
t:Term "*" f:Fact -> Term  { ast(BinOp(Mult, t, f)) }
f:Fact            -> Term  { ast(f) }
n:NUM             -> Fact  { ast(Int(n)) }
"(" e:Expr ")"    -> Fact  { ast(e) }
```

Constructors such as `Infix`, `Mult` . . . are freely choosen by the user.

Figure 3: More flexible AST generation: the `ast` annotation

notation: `ast` (Fig. 3). This enables the exchange of ASTs between closely related, but different, grammars. For instance we plan to use `sdf-astgen` to bridge the gap between our (standard) C++ ASTs, and CodeBoost's tailored ASTs.

4 Conclusion

We emphasized that self-contained grammar modules can be the most productive paradigm depending on the actual constraints of the project at hand. Because regular Stratego/XT tools do not support SDF annotation variations, we propose to replace them with the ESDF set of tools that support genericity with respect to annotation kinds. We also submitted four such extensions that are useful in our framework. Interesting extensions to this work include the support of more ambitious changes in the grammar of the grammars, and the exploration of means to provide easy composition of several different aspects of grammar modules while keeping concerns separated.

References

Borghi, A., David, V., Demaille, A., and Gournet, O. (2005). Implementing attributes in SDF.

David, V., Demaille, A., Durlin, R., and Gournet, O. (2005). C/C++ Disambiguation Using Attribute Grammars.

Visser, E. (1995). A family of syntax definition formalisms. In van den Brand, M. G. J. et al., editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam.