Declarative Specification of Template-Based Textual Editors

Tobi Vollebregt
tobivollebregt@gmail.com

Lennart C. L. Kats
l.c.l.kats@tudelft.nl

Eelco Visser
visser@acm.org

Software Engineering Research Group
Delft University of Technology
The Netherlands

ABSTRACT

Syntax discoverability has been a crucial advantage of structure editors for new users of a language. Despite this advantage, structure editors have not been widely adopted. Based on immediate parsing and analyses, modern textual code editors are also increasingly syntax-aware: structure and textual editors are converging into a new editing paradigm that combines text and templates. Current text-based language workbenches require redundant specification of the ingredients for a template-based editor, which is detrimental to the quality of syntactic completion, as consistency and completeness of the definition cannot be guaranteed.

In this paper we describe the design and implementation of a specification language for syntax definition based on templates. It unifies the specification of parsers, unparsers and template-based editors. We evaluate the template language by application to two domain-specific languages used for tax benefits and mobile applications.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*pretty printers, program editors*; D.3.1 [**Programming Languages**]: Processors—*parsing, code generation*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.11 [**Software Architectures**]: Languages

1. INTRODUCTION

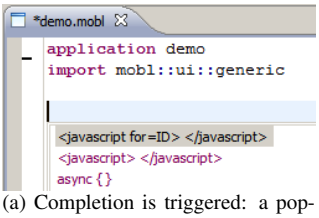
Language-aware structure editors provide a template-based paradigm for editing programs. They allow composing programs by selecting a template and filling in the placeholders, which can again be extended using templates. A crucial advantage of structure editors is syntax discoverability, helping new users to learn a language by presenting possible syntactic completions in a menu. Structure editors can be automatically generated from a syntax definition. Notable projects aiming at automatic generation of structure editors include MPS [23] and the Intentional Domain Workbench [19]. Structure editors can be used for general-purpose languages or for domain-specific languages (DSLs). A modern example of the former is the structure editor in MPS [23], for extensible languages based on Java. An example of the latter category is the DSL for modeling tax-benefit rules developed by IT services company Capgemini using the Cheetah system. Cheetah's facilities for discoverability and the use of templates are particularly effective to aid a small audience of domain expert programmers manage the verbose syntax based on legal texts.

Despite their good support for discoverability, structure editors have not been widely adopted. Pure structure editors tend to introduce an increased learning curve for basic editing operations. For example, they only support copy-pasting operations that maintain well-formedness of the tree and require small, yet non-trivial "refactoring" operations for editing existing code, e.g. when converting an `if` statement to an `if-else` statement. They also lack integration with other tools and expose the user to vendor lock-in. Transferring code across tools requires a shared representation that is generally not available. With software engineering tools such as issue trackers, forums, search, and version control being based on text, a textual representation is preferable, but requires the use of a parser and a parseable language syntax. This forces tools based on structure editors to find new solutions to problems long solved in the text domain.

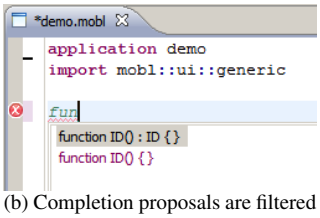
To alleviate the problems of structure editors, there has been a long history of hybrid structure editors that introduce textual editing features to structure editors [24, 18, 11]. Conversely, modern textual code editors such as those in Eclipse and Visual Studio are increasingly syntax-aware, based on parsers that run while a program is edited. Over time, they have acquired features ranging from code folding to syntactic completions allowing programmers to fill in textual templates. Indeed, structure and textual editors are converging into a new editing paradigm that combines text and templates.

In order to provide the advantages of text editing to the tax-benefit DSL of Capgemini, we converted the language from the Cheetah system, which uses a structure editor, to the parser-based Spoofax language workbench [10]. We quickly realized that it would be impossible for a user to write new models in such a verbose language in the textual editor of Spoofax, without accurate and complete syntax discovery. In syntax-aware text editors this discovery is provided in the form of syntactic completion. Accurate and complete syntactic completion depends critically on two features of a language workbench: first, the syntactic completion proposals presented to the user must be relevant and complete, and second, it must be feasible to create and maintain the specification necessary to make the editor aware of these completion proposals.

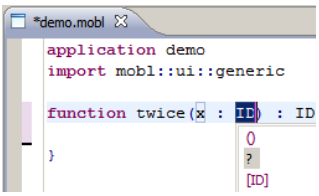
LDTA 2012 Tallinn, Estonia



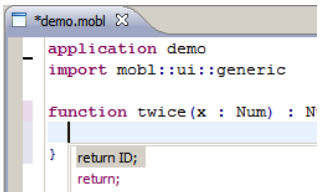
(a) Completion is triggered: a pop-up menu shows the available completion proposals.



(b) Completion proposals are filtered by typing the *fun* prefix into the editor.



(c) The type of function argument is selected.



(d) Back to the start: completion is triggered for the body of the function.

Figure 1: Template-oriented editing in a textual editor. The templates are editable as text and mark placeholders with rectangles, providing context menus to assign their values.

Text-based language workbenches currently require redundant specification of the ingredients for a template-based editor, i.e. concrete syntax, abstract syntax, completion templates, and pretty-print rules, which is detrimental to the quality of syntactic completion in syntax-aware editors. Evolution of the language requires maintenance of all ingredients in order to maintain completeness and consistency. It is tedious and therefore easy to make mistakes while adding or adapting a completion template for each new or modified language construct.

In this paper, we present the design of a template-based syntax definition language[1] that unifies the specification of parsers, unparsers, and template-based editors in order to support the efficient construction of template-based editing facilities in textual editors. In order to improve the runtime support for these template-based editing facilities, we describe an approach to compute the set of applicable templates at the location of the cursor.

We have implemented the template language in an extension of the Spoofax language workbench [10] and validated the approach by applying the techniques in two mature DSLs.

We proceed as follows. In the next section, we provide background on language and editor implementation. In Section 3 we describe the design of a template-based specification language for syntax. We then describe how to generate template-based editors from such a language in Section 4 and how template-based editing can be supported in Section 5. In Section 6 we evaluate the approach. We discuss related work in Section 7 and present our conclusion in Section 8.

2. BACKGROUND

Modern integrated development environments (IDEs) contribute significantly to the productivity of software developers and the adoption of new languages. The development of a complete IDE from scratch is a significant undertaking. As an alternative, language workbenches [6, 10, 23] can generate a complete IDE plug-in and a compiler from high-level *language definitions*.

Key to the generation of full-featured editors is the use of an abstract representation of programs that is maintained as programs are

[1] http://strategoxt.org/Spoofax/TemplateLanguage

```
context-free syntax
  "module" ID Definition*       -> Start      {cons("Module")}
  "entity" ID "{" Property* "}" -> Definition {cons("Entity")}
  ID ":" Type                   -> Property   {cons("Property")}
  ID                            -> Type       {cons("Type")}
```

(a) Grammar of the language, in SDF

```
completion template : Start =      "module␣" <m> (blank)
completion template : Definition = "entity␣" <e> "␣{}" (blank)
completion template : Property =   <x> "␣:␣" <T> (blank)
```

(b) Completion templates, for syntactic completion, in the EditorService language

```
[  Module   -- V[H[KW["module"] _1] _2],
   Entity   -- V is=2 [H[KW["entity"] _1 KW["{"]] _2] KW["}"],
   Property -- H[_1 KW[":"] _2]
]
```

(c) Pretty printer specification, in the PP language

Figure 2: Redundant concrete syntax specification in Spoofax

edited. The abstract representation is used for *editor services*, i.e. facilities such as an outline of the program and reference resolving to navigate to the definition sites of identifiers. Some of these facilities can be implemented directly at the level of concrete syntax, using regular expressions or other forms of pattern matching, but a structured, abstract representation ensures a uniform interface for editor services.

A central part of the definition of a language is the mapping between the concrete syntax of a language and its abstract representation. In structure editors, the mapping is defined as a projection from abstract representation to concrete syntax. In textual editors, the mapping is defined through a parser that constructs the abstract representation from a textual concrete syntax. Syntax-aware textual editors also apply a reverse mapping in editor services such as content completion, pretty printing, code formatting, and refactoring. In particular, modern syntax-aware textual editors support generation of code snippets via textual templates, triggered in a context-sensitive fashion by means of a content completion user interface. For example, Figure 1 demonstrates how a simple function is created using content completion.

In order to support the various applications of syntax definition, the different aspects of parsing, unparsing, formatting, and completion templates are often specified separately. For instance, Figure 2 shows an example of concrete syntax specification in Spoofax. Figure 2a defines a grammar, specified in SDF [8, 22], which is used to generate a parse table. Figure 2b defines completion templates that include layout and placeholder text. Figure 2c defines pretty printing rules, used for formatting existing and refactored or otherwise transformed code. Without going into the details of the three examples, it is immediately obvious that there is redundancy in these specifications.

Redundancy in syntax definitions is a common issue in tools to create syntax-aware textual editors, as we discuss in Section 7. It poses a maintenance problem as the language evolves: completion templates and pretty printer specification may lag behind modifications to the grammar of the language, hampering their completeness. One solution for the redundancy is to generate a default pretty printer from the syntax definition as applied for Spoofax in [10], and possibly generate default completion templates as well. Unfortunately, such a generative approach means that manually customized templates and formatting rules have to be combined with generated rules, which means it does not address the maintenance problem of these separate specifications.

3. TEMPLATE-BASED SYNTAX DEFINITION

In this section we introduce a new syntax definition language based on templates as those found in template engines such as StringTemplate [14]. Additionally, we base the design of auxiliary features of the language, such as priority specification and lexical syntax, on that of SDF [8, 22]. The aim of the language is to eliminate the redundancy between different syntactic specifications, by combining concrete syntax, abstract syntax, formatting, whitespace, and placeholder names. We argue that through the use of templates, the language is rich in information yet elegant and simple.

Basic template-based syntax definitions consist of *template productions* that correspond to production rules in a grammar. They have the following form:

```
s.label = <
  template
>
```

where *s* is the name of the symbol being defined, *label* is its constructor label used for the abstract representation, and *template* is a template that may include concrete syntax, references to other symbols (*placeholders*), and layout. Both the template and its placeholders are enclosed by <...> brackets.

As a first example, the following template productions define basic arithmetic expressions:

```
templates
  Exp.Num = <<INT>>
  Exp.Plus = <<Exp> + <Exp>>
  Exp.Times = <<Exp> * <Exp>>
```

The first production defines a template for number literals, defining a template for the `Exp` symbol based on a reference to the `INT` symbol. The other productions specify templates for the `+` and `*` operators. The last two templates consist of five elements: an `<Exp>` placeholder, whitespace, the `+` or `*` sign, more whitespace, and another placeholder. Of these elements, the whitespace elements are not considered for parser generation. Instead they are used for formatting in a generated pretty printer and completion templates. Whitespace characters treated this way are spaces, tabs, and newlines.

Placeholders can use the common `*` and `+` operators for repetition, and `?` for optionals. For repeated symbols with a separator symbol *s*, `<symbol*; separator=s>` can be used. The following template productions illustrate these features, adding function calls and definitions to the expression language.

```
templates
  FunctionDef.Function = <
    function <ID>(<ID*; separator=", ">) = <Exp>
  >
  Exp.Call = <<ID>(<Exp*; separator=", ">)>
```

Disambiguation

Grammars can be extended with disambiguation rules and annotations to express language characteristics such as associativity and operator precedence. In our running example, multiplication has a higher precedence than addition. Whereas in SDF [13] priorities are specified declaratively by *copying* the relevant productions and ordering them, separated by >-symbols, we add the option of specifying priorities through *references* to the relevant productions, so as to eliminate redundancy. The difference is shown in Figure 3.

```
templates
  Exp.Plus =
        <<Exp> + <Exp>> {left}
  Exp.Times =
        <<Exp> * <Exp>> {left}
context-free priorities
  Exp.Times >
  Exp.Plus
```

(a) Associativity and priorities with templates and references

```
context-free syntax
  Exp "+" Exp -> Exp
        {left, cons("Plus")}
  Exp "*" Exp -> Exp
        {left, cons("Times")}
context-free priorities
  Exp "*" Exp -> Exp >
  Exp "+" Exp -> Exp
```

(b) Equivalent associativity and priorities in SDF

Figure 3: Expression grammar

```
lexical syntax
  ID      = [A-Z] [A-Za-z0-9]*
  INT     = [0-9]+
  LAYOUT = [\ \t\r\n]
```

(a) Lexical productions

```
lexical restrictions
  ID  -/- [A-Za-z0-9]
  INT -/- [0-9]
context-free restrictions
  LAYOUT? -/- [\ \t\r\n]
```

(b) Lexical and context-free restrictions

Figure 4: EBNF-ordered productions

Lexical Syntax

A part of syntax definitions we have not discussed so far is lexical syntax. Lexical syntax elements such as `INT` and `ID` in our expression language, are, unlike the context-free productions we discussed so far, generally specified using a form of regular expressions. In the abstract representation they are usually represented as simple strings, making unparsing trivial. For consistency, lexical productions can be specified in symbol-first order, as shown in in Figure 4a. The definition of the body of lexical productions is shared with SDF [8, 22]. Both lexical and context-free syntax can be disambiguated using restriction sections in SDF [13], a construct that we inherit in our syntax specification language (Figure 4b).

The syntax of the template language is summarized in Figure 5. We proceed with a description of the mapping from syntax templates to SDF, completion templates, and pretty printing rules.

4. GENERATING TEMPLATE-BASED EDITORS

Generating SDF

Syntax templates closely match context-free syntax in SDF. Specifically, to go from a syntax template to an SDF production, all layout is discarded: in SDF, there is implicit `LAYOUT?` between all symbols in a context-free production. The remaining elements (literals and placeholders) are converted in-order to their respective SDF equivalents. Layout is trimmed from the separator option of list placeholders. An example of this transformation is shown in Figure 6.

Generating Completion Templates

Completion templates in Spoofax consist of the following components:

- A symbol that indicates the context in which the template is applicable.
- A string, which is displayed in the completion pop-up, and is used to filter the list of proposals.
- A list of elements of the completion template. Each element is either a string, possibly including line breaks and indentation, a placeholder, or the special `(cursor)` directive. This last directive indicates the location of the cursor after the user

Productions	
`Sort = <e*>`	Template with elements *e*∗
`Sort.Cons = <e*>`	Template with elements *e*∗
Placeholders	
`<A>`	Placeholder (1)
`<A?>`	Optional placeholder (0..1)
`<A*>`	Repetition (0..*n*)
`<A+>`	Repetition (1..*n*)
`<A*; separator="\n">`	Repetition with separator
`<A; text="hi">`	Placeholder with replacement text for completion template
`<A; hide>`	Placeholder hidden from completion template
Priority specification	
`context-free priorities {left: Exp.Times Exp.Over} > {left: Exp.Plus Exp.Minus}`	References to template productions
Lexical syntax	
`lexical syntax ID = [A-Za-z]+`	Lexical productions in EBNF-order

Figure 5: Summary of the template language syntax

```
Exp.Plus = <<Exp> + <Exp>>
```

⇓ generate SDF ⇓

```
Exp "+" Exp -> Exp {cons("Plus")}
```

Figure 6: Generate an SDF production from a syntax template

has cycled through all placeholders. A placeholder consists of an initial replacement text, and an optional symbol, which is used to display a list of syntactic completions applicable at the position of that placeholder, as soon as the user switches to this placeholder.

- A set of annotations. The only relevant annotation in use is `(blank)`, which constrains the completion template to blank lines.

We do not perform a linear transformation from syntax templates to completion templates, as we did for the grammar. The reason is best illustrated with Figure 7. For certain language elements, we may want to factor out repeated constructs, such as the `<Statement*; separator="\n">` placeholder in the example. The user of the editor, however, should not be exposed to such implementation details. In particular, the user should not be forced to repeatedly apply completion to fill in required parts of a language construct: those required parts should be inserted into the program text as soon as the completion proposal for the language construct is applied.

Therefore, we substitute the referred template for each placeholder with a multiplicity of one and higher (`<A>` and `<A+>`), unless the placeholder refers to the containing template. In the step *expand template* of the example in Figure 7, the `<MetaAnnos>` and `<Statements>` placeholders are expanded.

The *simplify template* step removes placeholders with the `hide` option, and processes placeholders that can generate the empty string (`<A?>` and `<A*>`). Call those placeholders ε-placeholders. These placeholders are treated differently depending on their location in the template. The `(cursor)` directive is substituted for the first

```
FunctionDef.Function = <
  <MetaAnnos>
  function <QId>(<FArg*; separator=",␣">) : <Type> {
    <Statements>
  }
>
MetaAnnos  = <<MetaAnno*; separator="\n", hide>>
Statements = <<Statement*; separator="\n">>
```

⇓ expand template ⇓

```
FunctionDef.Function = <
  <MetaAnno*; separator="\n", hide>
  function <ID:QId>(<FArg*; separator=",␣">) : <ID:Type> {
    <Statement*; separator="\n">
  }
>
```

⇓ simplify template ⇓

```
FunctionDef.Function = <
  function <ID:QId>(<:FArg>) : <ID:Type> {
    (cursor)
  }
>
```

⇓ generate completion template ⇓

```
completion template FunctionDef: "function␣ID()␣:␣ID␣{␣}" =
  "function␣" <ID:QId> "(" <:FArg> ")␣:␣" <ID:Type>
    "␣{\n\t" (cursor) "\n}" (blank)
```

Figure 7: Generate completion templates from syntax templates

line that consists of a single ε-placeholder. Further instances of ε-placeholders on a single line are ignored: we expect the user to retrigger completion when they desire to insert a template at these positions. In Figure 7, `<MetaAnno*; separator="\n", hide>` is removed, and `<Statement*; separator="\n">` is replaced by the `(cursor)` directive. Remaining ε-placeholders are replaced by an empty placeholder in the completion template that can be expanded to a single occurrence of one of the referred templates. In Figure 7 this is demonstrated by the replacement of `<FArg*; separator=",␣">` by `<:FArg>`.

Generating Pretty Printing Rules

A simple set of recursive, bottom-up pretty printing rules can be generated from syntax templates. We generate these rules in the program transformation language Stratego. It can be seen in Figure 8 that a pretty printing rule consists of a number of components. The name of the rule, `pp-Statement`, is composed from the name of the symbol. The rule matches the constructor `IfThen` with two arguments. The number of arguments is equal to the number of placeholders in the syntax template.

When the rule matches, child nodes are pretty printed by (recursively) invoking (other) pretty printing rules. Then, the text for all elements of the template is concatenated, while the text for child nodes is indented by the amount the respective placeholder is indented in the syntax template.

5. RUNTIME SUPPORT FOR TEMPLATE-BASED EDITORS

For template-based editing to be effective, only contextually relevant templates should be shown. Pure structure editors achieve this based on the currently selected placeholder. To achieve the same in textual editors, the editor must be aware of the syntactic category of the text at the cursor location at any time. The pro-

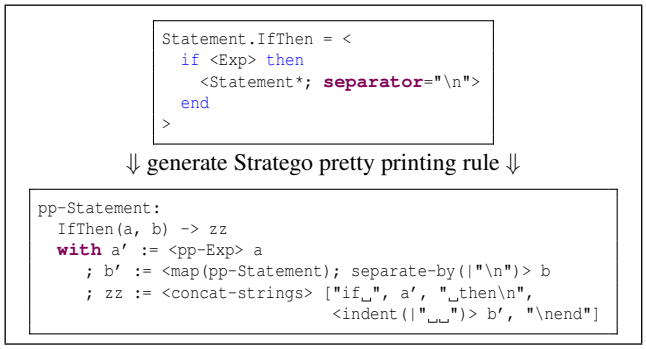


Figure 8: Generate Stratego pretty printer from syntax templates

vided completions must be accurate: all applicable templates must be included, and no inapplicable templates may be included. In this section we describe an approach for gathering an accurate list of templates in a parser-based editor.

Determining the syntactic category at the cursor location in a language-agnostic fashion is not trivial. If possible, changes to generated parsers to support this facility should be provided. In addition, syntax errors need to be taken into consideration; the editor must be able to determine what type of template should be inserted even when a program is edited and is in a syntactically incorrect state.

Original Implementation

The solution originally implemented in Spoofax first creates a modified program text that includes a marker at the cursor location. The marker matches the syntax for identifiers, and is unlikely to be present anywhere else in the program text. The modified program text is then parsed, after which the AST is searched for the marker. Spoofax infers the symbols that should have been allowed at the position of the cursor from the token stream, and meta data attached to AST nodes.

The interaction between the involved components is a problem with this implementation. When the modified program text is parsed, and it has parse errors, error recovery gets involved. Unless completion is invoked at a position where an identifier is allowed, there will be parse errors. The error recovery algorithm in the SGLR parser used in Spoofax attempts to get the parser back on track by performing a minimum number of token insertions and/or removals. As such, it may remove the marker, or insert punctuation that pushes the marker into another language construct.

One solution is to make the parser aware of the cursor location, and report the allowable syntactic categories at that character offset during parsing. This solution is specific to SGLR, and needs to be re-implemented in every other parser. We look for a more generic and less complex solution.

Our Solution

We apply a grammar generation technique that adds a production `CONTENTCOMPLETE -> X {cons("COMPLETION-X")}` for every symbol `X`, where `CONTENTCOMPLETE` is a symbol that recognizes the inserted marker text, including surrounding identifier characters. When the program text with marker is parsed, the marker can be parsed as every symbol allowed at its location. Because we encode the name of the symbol in the AST constructor, the editor runtime knows all symbols allowed at the position of the marker. The set of symbols is then used to select appropriate completion templates to display to the user.

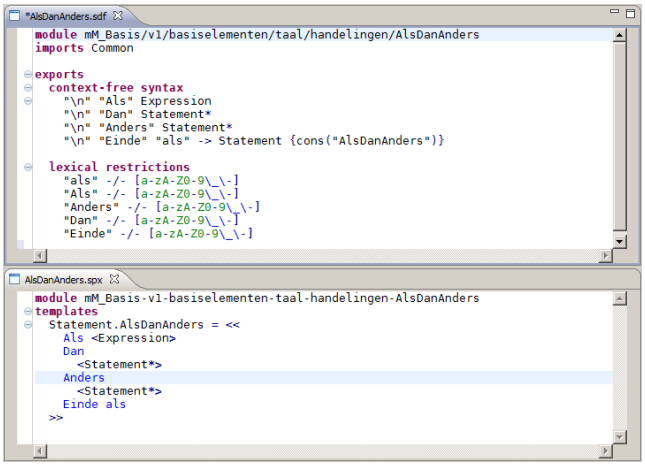


Figure 9: A Dutch `if-else` statement expressed in SDF (top), and expressed in the template language (bottom)

6. EVALUATION

We investigated whether the template language is sufficiently expressive to describe the syntax of existing DSLs. The requirements are that the syntax definition generated from the syntax templates must be equivalent to the original syntax definition. The completion templates must behave as was intended with the design of the template language, and the pretty printer must be able to output reasonably pretty code.

To perform the evaluations we converted the SDF grammar of the DSL into unformatted syntax templates. These unformatted syntax templates were then manually formatted to match the existing specification of the DSL. The syntax definition using formatted templates was then compiled into a Spoofax editor for the language, which we then used to try completion and to pretty print a number of DSL programs.

6.1 A Tax-Benefit Language

We converted the tax-benefit DSL from Cheetah to the template language. The DSL implements a *temporal database* [20] on top of .NET services and relational databases, while hiding the accidental complexity of those aspects from the user. Relevant to this evaluation is that the DSL is very verbose. It contains many specialized statements and expressions, many of which are Dutch sentences, with gaps where other statements or expressions can be inserted.

To acquire a syntax specification in the template language, we retargeted our initial conversion. The DSL models could be parsed using the template language syntax specification without modification. Additionally, we got completion templates and a working pretty printer "for free." The template languages reduced the syntax definition from 2101 (only SDF) to 997 lines. This large reduction can be attributed to the **`lexical restrictions`** the template language automatically generates for all keywords in each syntax template, in combination with the large number of keywords in the language, and the fact that, as a result of the conversion from Cheetah, each of the 129 language constructs is stored in a separate file. An example of a language construct defined in the template language, versus the same language construct defined in SDF, is shown in Figure 9.

6.2 The Mobl Web Programming Language

Mobl [9] is a DSL for the construction of mobile web applications. It features an extensive standard library, declarative speci-

fication of user interface, static type checking, and embedding of Javascript, CSS styling rules and HTML. We created a clone of the syntax of the Mobl language in our template language, using a converter in the Spoofax SDF editor. Within a few hours we formatted the 343 unformatted syntax templates, by inserting line breaks and indentation into 64 multi-line language constructs, and adapting layout throughout the language using search-replace. One production had to be manually refactored to three separate productions, because it employed the *alternative* operator, which is deprecated in SDF, and (by design) not present in the template language.

After some minor fixes the syntax templates resulted in a syntax definition that could be used to parse and pretty print all example code included with the Mobl project, although for nested `if-else` statements we hit limitations with regards to the placement of braces. Overall, the template language reduced the combined size of the syntax specifications from 1562 lines to 1162 lines, while delivering a complete pretty printer, and a complete set of completion templates.

The grammar of the Mobl language as specified by our template language is slightly more permissive than the original Mobl grammar, due to keywords that contain special characters, such as `@<javascript>` and `@doc`, which get tokenized by the SDF generator to `"@<" "javascript" ">"` and `"@" "doc"`, so that layout is allowed between these tokens. We will have to revisit this design decision, and consider, for example, removing the tokenization, and introducing a zero-length space character to insert `LAYOUT?`, to ensure it is possible to define such keywords in the template language.

Initially, syntactic completion in our evaluation was suboptimal due to the *annotations* present in Mobl at the start of many language constructs. Because the placeholder for these annotations is on a separate line, completion templates that produce a blank line before the language construct were generated. This behavior is likely not expected by the user, because these annotations are rarely used in Mobl. We corrected this by introducing the **`hide`** option to suppress the placeholders for annotations from the completion templates for many language constructs. The templates for annotations can be invoked separately, where desired.

Evaluation of the Runtime Support

We evaluated the runtime support for template-based editors on a sample program in the Mobl language by triggering completion on relevant positions in the sample program. Our approach was able to provide the editor runtime with an accurate and complete set of symbols on each sample position. The fact that our approach minimizes the interaction with error recovery likely explains these promising results.

7. RELATED WORK

Unified Syntax Specifications

There are a number of current syntax specification approaches that aim to unify the specification of parsing and unparsing.

SYN [3] aims to be one syntax definition language for the specification of ASTs, lexical analysis, parsing and pretty-printing. Its notation is similar to BNF, extended with a sublanguage for the specification of a lexical analyzer, and operators `h` (horizontal composition), `hv` (inconsistent line breaking), and `hov` (consistent line breaking) for the generation of a pretty printer. SYN has been implemented in Standard ML. Because the SYN compiler translates the syntax definition to input for the tools ML-Lex and ML-Yacc (ML implementations of the well known UNIX tools lex and yacc), a syntax definition in SYN faces the limitations of separate scanner and parser, and LALR(1) parsing.

Extended SDF [15] is an extension of SDF that embeds other specification languages. An important application of the work is the embedding of PP pretty printing rules (such as those in Figure 2c) in SDF attributes. Although this improves the locality of the different syntax definitions, it does not solve redundancy, as elements of the syntax are present both in the SDF, and in the attached pretty printing rule.

More recently, Rendel and Ostermann [16] propose partial isomorphisms for invertible computation, and use these to implement a combined parser/pretty printer library in Haskell. Productions are specified using invertible combinators used for parsing, unparsing, and abstract syntax (de)construction.

These approaches differ from our approach in their use of explicit operators that specify layout and formatting. By using templates, we provide a concise syntax that forgoes the use of operators and uses plain whitespace instead, while still being sufficiently expressive for our case study with two complete DSLs. In addition, they also do not consider completion templates.

Template-Based Editing

Many early language workbenches used a template-based editing paradigm in structure editors. Examples include Centaur [2] and the Synthesizer Generator [17]. While these systems faced the same problem of having to specify both abstract and concrete syntax, they did not have the problem of specifying both a parser and an unparser.

Hybrid textual/structure editors make it possible to switch to a text editing mode for a part of a program. Systems used to specify these editors do have the added dimension of parsing and unparsing, where they need a specification of formatted concrete syntax and a specification that specifies how to parse concrete syntax independent of the layout. While there have been different ways to address the issue, there has not been a solution that unifies the specification of all syntactic aspects. Examples of hybrid systems include the Programming System Generator (PSG) [1], PREGMATIC [21], and the ASF+SDF Meta-Environment [12].

PREGMATIC [21] specifies syntax using as part of attribute grammars. The grammar formalism does not include a formatting specification: instead, unformatted templates are generated from the grammar. The user can then edit the layout in those templates to format them as desired.

The Meta-Environment [12] uses SDF for syntax definition, and originally used the Generic Syntax-directed Editor (GSE) [4] as a hybrid editor. Contrary to many earlier structured editors it does not use pretty printing to convert abstract syntax into concrete syntax. Instead it maintains a two-way mapping between the text the user entered and the AST, so that a pretty printer is not needed during editing, and the user has full control over the layout of the program.

Template-based textual editors have text editing as their principal mode of operation, but can provide textual templates for editing. Examples of tools to create these editors include MontiCore [7], Xtext [5], and our own Spoofax [10]. Each of these systems has so far used a separate specification of syntax for parsing, pretty printing, and completion templates. As part of our work we implemented a template language for Spoofax, showing these aspects can be combined.

8. CONCLUSION

Syntax discoverability has been a crucial advantage of structure editors for new users of a language. Despite their excellent support for syntax discoverability, structure editors have not been widely adopted. Based on immediate parsing and analyses, modern textual code editors are also increasingly syntax-aware: structure and tex-

tual editors are converging into a new editing paradigm that combines text and templates. In these syntax-aware text editors, syntax discovery is provided in the form of syntactic completion, which depends critically on two features of a language workbench: first, it must be feasible to define and maintain sensible completion proposals for editors, and second, the syntactic completion proposals presented to the user must be relevant and complete. Current text-based languages require redundant specification of the different aspects that make up a template-based editor. Evolution of the language means that maintenance is required of all aspects in order to maintain completeness and consistency.

This paper addresses the issue of effective specification and implementation of syntax-aware textual editors. First, through unification of the specification of syntax, thus addressing the higher risk of incomplete/incorrect syntax due to redundant specifications. To accomplish this, we presented the design and implementation of a specification language that incorporates enough information so that syntax definition, pretty printer and completion templates can be generated. Doing so ensures completion templates are complete and up-to-date, thus improving the discoverability of syntax in the editor. Second, we describe techniques to accurately determine the syntactic categories at the cursor location, in order to present a relevant and complete set of completion proposals. We showed that by applying grammar generation techniques, it is possible to accomplish this goal without parser-specific modifications.

9. REFERENCES

[1] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[2] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 14–24. ACM, 1988.

[3] R. Boulton. *Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing*. Number 390. University of Cambridge, Computer Laboratory, 1996.

[4] M. H. H. van Dijk and J. W. C. Koorn. GSE, a generic syntax-directed editor. Technical Report CS-R9045, Centrum voor Wiskunde en Informatica (CWI), 1990.

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

[6] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.

[7] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Monticore: a framework for the development of textual domain specific languages. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Companion Volume*, pages 925–926. ACM, 2008.

[8] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[9] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In K. Fisher and C. V. Lopes, editors, *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA 2011*, pages 695–712. ACM, 2011.

[10] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463. ACM, 2010.

[11] A. A. Khwaja and J. E. Urban. Syntax-directed editing environments: Issues and features. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 230–237, 1993.

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

[13] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, October 1994.

[14] T. J. Parr. Enforcing strict model-view separation in template engines. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 224–233. ACM, 2004.

[15] N. Pouillard. Extending SDF. Technical Report 0407, EPITA, jul 2004.

[16] T. Rendel and K. Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 1–12. ACM, 2010.

[17] T. W. Reps and T. Teitelbaum. The synthesizer generator. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 42–48. ACM, 1984.

[18] U. Shani. Should program editors not abandon text oriented commands? *SIGPLAN Notices*, 18(1):35–41, 1983.

[19] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 451–464. ACM, 2006.

[20] R. Snodgrass. Temporal databases. *IEEE Computer*, 19:22–64, 1992.

[21] M. G. J. van den Brand. *PREGMATIC - a generator for incremental programming environments*. PhD thesis, University Nijmegen, 1992.

[22] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[23] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In M. van den Brand, B. Malloy, and S. Staab, editors, *Software Language Engineering, Third International Conference, SLE 2010*, Lecture Notes in Computer Science. Springer, 2010.

[24] R. C. Waters. Program editors should not abandon text oriented commands. *SIGPLAN Notices*, 17(7):39–46, 1982.