Projecting Textual Languages

Mauricio Verano Merino, Jur Bartels, Mark van den Brand, Tijs van der Storm, and Eugen Schindler

Abstract *Context:* Language Workbenches (LWBs) are tools used to develop Domain-Specific Languages (DSLs) and their tooling. They are proven to both speed up the development process of DSLs and improve the productivity of language engineers. Language workbenches can be either textual, projectional, or graphical – each of these with their own strengths and drawbacks.
Inquiry: However, it is not possible to compose or exchange languages developed using different LWBs. For instance, if a language engineer wants to compose a language which is implemented in a purely text-based language workbench with a language which is implemented in a projectional language workbench, he must rewrite one of the languages by hand. Previous research has addressed this problem by defining a grammar-based specification in MPS; yet this means language engineers have to manually rewrite the entire grammar or part of it into this formalism. To alleviate this problem, we seek to bridge grammar-based languages and projectional-based languages in a semi-automated fashion.
Approach: Use existing context-free grammars defined in Rascal to generate an equivalent language using JetBrains MPS. This based on identifying a mapping between the different syntactic building blocks of both textual-like and projectional languages. In this process, we identified that grammars encode information that can be used to derive language components other than the language's syntax. Thus, we

Mauricio Verano Merino
Eindhoven University of Technology, Eindhoven, The Netherlands,e-mail: m.verano.merino@tue.nl

Jur Bartels
Eindhoven University of Technology, Eindhoven, The Netherlands, e-mail: j.bartels@student.tue.nl

Mark van den Brand
Eindhoven University of Technology, Eindhoven, The Netherlands, e-mail: M.G.J.v.d.Brand@tue.nl

Tijs van der Storm
CWI, Amsterdam, The Netherlands,
University of Groningen, Groningen, The Netherlands, e-mail: storm@cwi.nl

Eugen Schindler
Canon CPP, Venlo, The Netherlands, e-mail: eugen.schindler@cpp.canon

used this grammar-specific information to derive a projectional editor.
Knowledge: We propose a mapping between context-free grammar productions and projectional-language concepts in a generic way. We analyze the grammar-based language definition, and then we derive the equivalent language concepts in MPS. During this analysis phase, we also extract some information from the grammar (e.g., literals). The extracted information is used together with some heuristics to generate an editor for the resulting projectional language.
Grounding: To evaluate the mapping and generation of a language, we use a tiny grammar for the addition of natural numbers and a grammar for ECMAScript 5. We also develop a mechanism to import textual programs and translate them into a projectional model. The transformed program must conform to the generated projectional language. As a case study, we use an ECMAScript grammar written in Rascal. To compare the resulting language we use as a baseline an ad-hoc MPS implementation of the same language.
Importance: There are many existing textual and projectional languages, and sometimes, they need to be composed. However, existing tools do not support this feature, so language engineers often have to re-implement existing languages. As a result of this research, we allow language engineers to use textual and projectional languages, without having to manually re-implement them. Generating a projectional language from a grammar-based specification supports this composition.

1 Introduction

Language Workbenches (LWBs [1]) are IDEs that support engineers in the design and development of software languages [2]. These tools are aimed to improve and increase the adoption of Language-Oriented Programming (LOP). LOP is a technique for solving domain problems by defining a single or multiple languages [3] (e.g., as in the Unix tradition of little languages [4]), also known as Domain-Specific Languages (DSLs). A DSL is often a small and simple language tailored to solve problems in a particular application domain [5]. There are two types of DSLs, internal and external [3] . The first one reuses the concrete syntax of the host language and its parser. In contrast, an external DSL does not rely on the host syntax nor its parser; instead, a language engineer must implement these two language components.

LWBs come in different flavors; in this chapter, we focus on two types, textual and projectional LWBs. The first type uses a sequence of characters to represent, view, and edit language elements. The concrete syntax of a textual language is often specified through a grammar. A textual language does not require special editing tools, a plain text editor is enough. Whereas, a projectional LWB uses projectional editors to implement the language's concrete syntax. A projectional editor is a representation and editor for the direct manipulation of Abstract Syntax Trees (ASTs), without the need for parsing text. An AST is a data structure often used by compilers to represent language elements.

Model-driven software development is becoming more popular because it allows the development of software at a higher level of abstraction by moving the technical complexity to code generators. The advantages of MDE is many-fold. First, it allows for a more efficient way of developing software. Second, it allows for checking (correct) properties using model checkers, and finally the specification is moved from software engineers to domain engineers.

Companies in the Eindhoven (The Netherlands) region (e.g., Canon PPP and ASML), have been using DSLs for a number of years [6]. Some of these companies are using textual LWBs, projectional LWBs, or both, such as Canon CPPP. When companies are using both types of LWBs, it is often desired to reuse existing textual languages within a projectional LWB and vice-versa. If such reuse facility exists, companies will avoid costs of reimplementing features and maintaining the same functionality in different platforms. Likewise, from the engineering point of view, developers can be more productive and invest more time in developing new features or improving existing ones. In addition, the reuse strategy could reduce time to market for new products.

In this chapter, we present a robust approach towards bridging the gap between textual and projectional LWBs. Rascal2MPS is a tool that implements the bridge between textual and projectional LWB, and it is based on Bartels [7] master thesis. This tool translates a textual language written in Rascal LWB [8] to an equivalent projectional language in JetBrains MPS.

The contributions of this chapter can be summarized as follows:

- A generic bridge between textual and projectional LWBs. By means of this bridge, developers can obtain a projectional language in JetBrains MPS from a context-free grammar written in Rascal.
- A mechanism to generate projectional editors from a context-free grammar. This mechanism uses a set of pretty-printing heuristics that takes into account the production rules' structure.
- A tool to import existing programs written in a textual language as projectional models of the generated language.

The structure of this chapter is as follows: Section 2 presents background information about software language engineering. Then, in Section 3, we describe the motivation that supports this work and the problem statement. In Section 4 we present our solution and its architecture, respectively. Section 6 discusses limitations of the current approach. In Section 5, we evaluate the current approach by comparing an ad-hoc implementation of JavaScript against a generated version. We conclude this chapter with related work and future research directions (Sections 7 and 8).

2 Background

In this section, we present some of the basic concepts used in this chapter. The concepts described below are mostly about Software Language Engineering (SLE).

Mainly, we focus on discussing the language's syntax and its definition in both textual and projectional LWBs.

2.1 Software Language Engineering

Software Languages A software language is a mean of communication between programmers or end-users and machines to develop software. Languages are often divided into three main components namely, syntax, semantics, and pragmatics [2, 9]. The syntax of a language is a set of rules that define valid language constructs – for example, defining a group of rules that captures expressions or statements. The language's syntax can be expressed in a concrete and abstract way. Concrete syntax is designed as the user interface for end-users to read and write programs; whereas abstract syntax is the interface to language implementation. The semantics of a language represents a mapping between language constructs defined in the syntax and their meaning depending on the domain. Such mapping can be defined in different manners such as operational semantics or model-to-model transformations [2]. Language pragmatics describes the purpose of the language constructs, and it is defined informally often in natural language through narrative and examples.

Language-Oriented Programming (LOP) is an approach to software development where the main activity in development consists of defining and applying multiple DSLs. Programmers define custom languages to capture aspects of a software system in a structured way. The idea is that each language captures the essential knowledge of aspects of a problem domain, so that productivity is increased, and domain knowledge is decoupled from implementation concerns. In other words, DSLs captures the "*what*" of the domain, whereas compilers, code generators and interpreters defined for those DSLs define the "*how*".

Language Workbench To help language engineers in the development of software languages they can rely on metaprogramming tools called LWBs. These tools simplify and decrease the development cost of software languages and their tooling [3]. A LWB offers two main features: a specialized set of metalanguages for defining the syntax and semantics of DSLs, and affordances to define various IDE services such as syntax highlighting, error marking, and auto-completion. In this chapter, we are going to focus on the former. There are two types of LWBs, namely, textual (also called syntax-directed), and projectional (also called structural) [1, 2, 10]. The main difference between these types is how languages are described and how programs are edited. A textual LWB employs plain text and parsing to map concrete syntax to the internal structures of the LWB. For instance, Rascal uses context-free grammars as formalism [11] for defining the language's syntax. A projectional LWB allows a program's AST to be edited directly [12]. For instance, MPS uses a node concept hierarchy [10] to define the language's structure, and MPS implements a projectional editor for manipulating programs. A *projectional editor* is a user interface (UI) for creating, editing, and manipulating ASTs.

2.2 Syntax of Textual and Projectional Languages

As mentioned before, the syntax of a software language is the notation. Usually, it is divided into two, namely, concrete syntax and abstract syntax. In this subsection, we describe how different LWBs represent both types of syntaxes.

In textual LWBs, the concrete syntax of a language is usually specified using Context-Free Grammars (CFGs); while in projectional LWBs the concrete syntax is expressed as AST projections. Below we explain both approaches and highlight their main differences. To clarify the differences between textual and projectional LWBs we will use Rascal and MPS. Table 1 shows a comparison of the notations used by these two platforms to define language's syntax.

Language	Rascal	MPS
Concrete Syntax	Context-Free Grammar	Projectional Editor
Abstract Syntax	Algebraic Data Type	AST Concept Hierarchy

Table 1: Comparison between notations used for describing languages in textual and projectional LWBs.

Context-free grammars

A CFG is a formalism for describing languages using recursive definitions of string categories. A CFG C is a quadruple

$$C \rightarrow (S, NT, T, P)$$

In which S is the start symbol ($S \in NT$), NT is a set of syntactic categories also known as nonterminals, T is a set of terminal symbols, and P are production rules that transform expressions of the form $V \rightarrow w$. V is a nonterminal ($V \in NT$), and w could be zero or more nonterminal or terminal symbols ($w \in (T \cup NT)$).

For example, a CFG that describes the addition of natural numbers $\mathbb{N}$ is shown below:

$$G = (Exp, \{Exp, Number\}, \{+\} \cup \mathbb{N}, P)$$

The production rules P are defined as follows:

$$start \rightarrow Exp \tag{1}$$

$$Exp \rightarrow Number \tag{2}$$

$$Exp \rightarrow Exp + Exp \tag{3}$$

$$Number \rightarrow i (i \in \mathbb{N}) \tag{4}$$

Listing 1: Concrete syntax of addition and numbers in Rascal.

```
start syntax Exp = number: Nat nat | addition: Exp lhs "+" Exp rhs;

lexical Nat = digits: Natural;
```

Listing 2: Lexical library.

```
lexical BasicString = [a-z]*[a-z];
lexical Natural = [0-9]+;
lexical String = "\"" ![\"]* "\"";
```

Listing 3: Abstract syntax of addition and numbers in Rascal.

```
data Exp = adddition(Exp lhs, Exp rhs) | number(int n);
```

By applying the previous production rules we can write the arithmetic expression $a + b$ (where $a, b \in \mathbb{N}$) as:

$$\begin{aligned} start &\rightarrow Exp \\ Exp &\rightarrow Exp + Exp \\ Exp + Exp &\rightarrow a + Exp \\ a + Exp &\rightarrow a + b \\ a + b & \end{aligned}$$

Once there are no more nonterminals (NT) we cannot rewrite the expression $a + b$ because there are no production rules that can be applied. We say that a program is syntactically valid if there is a derivation tree from the start symbol to the string that represents the program.

For instance, the concrete and abstract syntax of the language described above can be implemented in Rascal as shown in Listings 1 and 3, respectively. The first one defines two nonterminals, namely, *Exp* and *Nat*. The *Exp* rule contains two productions, for literal numbers and addition. The *Nat* nonterminal defines natural numbers. The AST Listing 3 defines an Algebraic Data Type (ADT) that captures the structure of the language with two constructors : *nat(...)* and *add(...)*. The terminals of the expression grammar (i.e., `Nat`) are represented using built-in primitive types of Rascal (i.e. **`int`**).

```
concept Addition extends    BaseConcept
                 implements Expression

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  left  : Expression[1]
  right : Expression[1]

  references:
  << ... >>
```

```
concept Number extends    BaseConcept
               implements Expression

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  value : integer

  children:
  << ... >>

  references:
  << ... >>
```

Fig. 1: Concept definition of addition (left) and numbers (right)

```
addition {

  left :
    number {
      value : 1

    }
  right :
    number {
      value : 6

    }
}
```

Fig. 2: Reflective editor for the operation a + b, where $a = 1$ and $b = 6$.

Syntax in Projectional LWBs

In a projectional LWB, the syntax is also divided into its concrete and abstract representation. The concrete syntax corresponds to an editor definition, whereas the abstract syntax is defined in a concept hierarchy.

Projectional editors do not share a common formalism for defining abstract syntax; therefore, each platform is provides its own formalism. MPS uses a node concept hierarchy [10]. For instance, the AST that represents a language for describing addition of natural numbers is shown in Figure 1. The MPS implementation uses an *Expression* interface and two concepts, namely *Addition* and *Number*. To represent numbers we use the built-in *integer* data type.

How the users will edit expressions of this kind is defined by an editor definition. However, MPS also offers a generic *reflective editor*, so that every concept in MPS comes with a default editor. A reflective editor is a projectional representation of an AST that developers can use out-of-the-box. An example of an arithmetic expression program using the reflective editor is shown in Figure 2.

3 Motivation

A DSL offers a programming abstractions that are closer to domain requirements than general programming languages [13]. Likewise, DSLs offer syntax closer to the domain expert's knowledge. DSLs have been around for a couple of decades, but they have not been widely adopted in the industry yet [14, 15]. The limited adoption of DSLs in the industry is partly due to the lack of mature tools for their adoption, as described in [16, 17].

Nowadays, language engineers have different tools and metalanguages to choose from when they require to implement a new language. The right selection of such tools is essential for the language's success. Likewise, this means that companies end-up with diverse ecosystems of languages and tools. These tools are continuously changing to support diverse business requirements, depending on what they want to achieve or the organization's needs. Communication between tools and languages is often required to share functionalities among different components. When there is no communication between platforms, developers could reimplement these features. However, reimplementing these functionalities is a cumbersome activity, and it does not fix the problem in the long term because, at some point, it might be required to reimplement those features again.

For instance, there are several textual languages at Canon PPP that they have been developing and maintaining for some years. However, they have more recent languages that were developed using a projectional LWB. Recently, they have found that they require to interoperate languages from different LWBs to address their new business needs and reduce the time to market. Therefore, they seek for a bridge that supports the reuse and translation of existing languages across heterogeneous LWBs.

4 Approach: Projecting Textual Languages

This section presents a mechanism for enabling the usage of textual languages through a projectional editor. In other words, the current approach translates existing textual languages into equivalent projectional languages, including both structure and editor aspects. Then the translation of existing textual programs into equivalent models of a generated projectional language is discussed. We first show a general overview of the approach. Then, we explain a generic mapping between CFGs and the structure of a projectional language. Afterwards, we describe the derivation of a projectional editor from a grammar; we show how to derive the editor aspect for each generated concept in the structure of the language. Finally, we explain the translation of textual programs to projectional models that conform to a generated projectional language. Although the current approach is implemented using Rascal and MPS, its principles can be adopted in the context of other LWBs.

Listing 4: Definition of the Exp interface in MPS.

```
interface concept Exp extends <none>

  properties:
  << ... >>

  children:
  << ... >>

  references:
  << ... >>
```

4.1 Mapping Grammars to Concept Hierarchies

This section contains the description of the mapping between a grammar and the structure of a projectional editor. The current approach analyzes elements of a CFG, namely, production rules, nonterminal, terminal, and lexical symbols. To illustrate each of the concepts of the mapping, we use the grammar for the *Addition language* shown in Listing 1.

Nonterminal symbols. The counterpart of a nonterminal symbol in MPS is an interface. An interface is a programming concept that may define the public, shared structure of a set of objects (typically described by classes). In the same way that interfaces may have multiple implementations (the classes), a nonterminal is "realized" by one or more productions. For instance, in Listing 1, there are two nonterminals, namely, `Exp` and `Number`. Thus, these two nonterminals map to two interfaces with the same name in the generated projectional language. The definition of the `Exp` interface in MPS is shown in Listing 4.

Furthermore, there is one additional nonterminal that we have not mentioned: the start symbol of the grammar. Structure concepts in MPS have a property named *instance can be root*. This property indicates whether the instances of a concept can be used to create ASTs. In our mapping, we take the start symbol of the grammar and create a concept in MPS. This concept will have the *instance can be root*-property set to `true`. For instance, in Listing 5, we show an example using the expression language, assuming we have a start symbol `Program` with a single production, `prog`.

Productions. A nonterminal rule has one or more productions. As we mentioned before, a nonterminal in a CFG is mapped to an `interface` concept in MPS. Therefore, to keep the relationship between a nonterminal and their productions, we map each production as an MPS *concept*. Each *concept* must implement the interface of the nonterminal. Moreover, the AST symbols in the production rule are mapped to either the *children* or *properties* field. When the symbol is a nonterminal, it is defined in the *children* field, and when the symbol is terminal or a lexical, it is mapped in the *properties* field. Note that symbols only relevant to concrete syntax, such as

Listing 5: Mapping a CFG start symbol into a MPS concept.

```
concept prog extends <default> implements Program

  instance can be root: true
  alias: <no alias>
  short description: Exp

  properties:
  << ... >>

  children:
  expression :Exp[1]

  references:
  << ... >>
```

Listing 6: Result of mapping a production rule to a concept in MPS.

```
concept addition extends <default> implements Exp

  instance can be root: false
  alias: +
  short description: Exp + Exp

  properties:
  << ... >>

  children:
  lhs :Exp[1]
  rhs :Exp[1]

  references:
  << ... >>
```

keywords and operator symbols are not mapped here, since they are not part of the abstract syntax; they will however be used to define editor aspects (see below).

For instance, `addition` (Listing 1) is a production rule of the nonterminal `Exp`. This production rule is mapped into an MPS concept that implements the `Exp` interface. The resulting concept in MPS is shown in Listing 6.

Lexicals Lexical define the terminals of a language, and are typically defined by some form of regular expressions. Rascal allows full context-free lexicals, but here we assume that all lexicals fall in the category of regular languages that can be defined by regular expressions.

To ease the mapping between Rascal lexicals and MPS concepts, we define a Rascal module that contains a set of default lexicals. These lexicals define the syntax of identifiers, string literals and integer numbers. Developers can use these lexicals

Listing 7: Lexical mapping.

```
concept digits extends <default> implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  nat: Natural

  children:
  << ... >>

  references:
  << ... >>
```

Listing 8: Result of mapping a Rascal lexical to an MPS constrained data type.

```
constrained string datatype: Natural

    matching regexp: [0-9]+
```

in their own Rascal grammars, but it is also possible for users to include their own lexicals. In this case, developers have to manually describe the mapping to MPS.

Each lexical is mapped both to a concept, like any other nonterminal, and to a *constrained data type*. To illustrate this, Listing 1 contains the definition of `Nat`, which consists of a single production, called `digits`. This production rule has a reference to `Natural`, which is one of the predefined lexicals (Listing 2). As a result, the lexical `Nat` is translated into a concept, called `digits` (Listing 7), and a *constrained data type*, called `Natural` (Listing 8). The `digits` concept has a single property of type `Natural`, which is a *constrained data type* capable of capturing natural numbers using the regular expressions engine of MPS.

List of symbols. In CFG it is possible to define a group of symbols of the same type, often expressed using Kleene's star (*) and plus (+). Kleene's operators (star and plus) are unary operators for concatenating number of symbols of the same type. The first one denotes zero or more elements, and the second one denotes one or more elements in the list. The current approach detects both operators (Kleene's star and plus) in productions. The operators are represented in MPS as children of a concept with cardinality zero-to-many (`0..*`) and one-to-many (`1..*`), respectively. For instance, let's add to the language shown in Listing 1 the following production:

```
start syntax Exp = ... | groupExp: Exp* exps;
```

This production defines zero or more expressions (`Exp`). The resulting mapping of the production `groupExp` is shown in Listing 9.

Listing 9: Concept mapping for a list of symbols.

```
concept groupExp extends <default> implements Exp

  instance can be root: true
  alias: <no alias>
  short description: Exp

  properties:
  << ... >>

  children:
  exps :Exp[0..n]

  references:
  << ... >>
```

4.2 Mapping Grammars to Editor Aspects

This section presents the mapping between a grammar and the editor aspect in MPS. For creating the editor aspect of the language we use the language's layout symbols, namely, literal and reference symbols. In this context, a reference symbol is a pointer to a nonterminal symbol (which can be lexical or context-free).

Literals. Literal symbols may be part of productions to improve readability of code or to disambiguate, they form an essential aspect of concrete syntax, and can be leveraged to obtain projectional editors.

To create an editor, we first take each production rule; we look at each symbol and keep track of the symbol's order. It is essential to keep track of the order because it affects how the editor displays the elements. In this process, we consider two types of symbols, namely, *literals* and *references*. If the symbol is a literal, it is added to the *node cell layout* as a placeholder text; for nonterminal symbols, we create a *reference*.

For example, the production rule that defines the addition between natural numbers has three symbols, namely, *lhs*, +, and *rhs*. Following the approach, we first take the *lhs* symbol and create a reference to its type `Exp`; then, we take the literal, +, and copy it to the editor, and finally, we create a reference to the *rhs* symbol, which is also of type `Exp`. Listing 10 shows the generated editor for addition. This editor has two references, namely, *lhs* and *rhs*. Editors use references to access concept properties. For instance, the reference `lhs` in the editor is a link to the `lhs` children in the addition concept. Moreover, the editor for *addition* has a literal (+) in between the two references. The literal is shown as a placeholder text for users, so that they can write expressions like `5 + 6`.

List of symbols. The editor aspect for a list of symbols (zero-to-many and one-to-many) is based on creating a collection of cells. More concretely, each list of

Listing 10: Generated editor for addition.

```
<default> editor for concept addition
  node cell layout:
    [- % lhs /empty cell: % + % rhs /empty cell: % -]

  inspected cell layout:
    <choose cell model>
```

Listing 11: Editor mapping for a list of symbols.

```
<default> editor for concept groupExp
  node cell layout:
    [-
      (- % exps % /empty cell: -)
    -]

  inspected cell layout:
    <choose cell model>
```

symbols is translated into an *indent cell* collection. Listing 11 shows the generated editor aspect for the `groupExp` production.

4.3 Editor improvement – AST Pruning

Having defined a mapping from grammars in CFG to the editor aspect in projectional languages, now we will move into improving the generated projectional editor. The editor can be improved by pruning the grammar to enhance IDE services (e.g., auto-completion). To prune the grammar, we removed chain rules (also known as unary rules) from production rules. The pruning process is as follows: first, we collect the production rules that have a single parent and are referenced only once in the grammar. Then, we merge the single reference with its parent.

To illustrate this, we use the following production rule:

$$A \rightarrow A|b|c|d$$

Long production rules are often split into smaller production rules for readability. Therefore, the previous production rule can also be written as:

$$A \rightarrow A|B$$

$$B \rightarrow b|c|d$$

The second alternative has an impact on the language's structure because it introduces a new nonterminal symbol *B*. This new nonterminal is translated in the AST as an extra node. Figure 3 shows a tree view of both ASTs. This new AST node, *B*, is only referenced once in the whole language definition. If we prune this type of definition, we avoid having an extra node in the projectional editor. For example, for defining a new instance `A`, users can use auto-completion to create node B, and then they can define leaf nodes (b, c, or d). If we prune the production rule, we remove node B (it is referenced only once), then we do not have to define an instance of B before creating leaf nodes. As a result, we enhance the tab-completion menu of the projectional editor.

Fig. 3: Tree-based view comparison.

4.4 Translating Textual Programs into Projectional Models

We extend the approach to translating existing textual programs into projectional models. The motivation for this extension is that we want to offer a mechanism for importing existing textual programs into the generated projectional language. We did not consider a manual translation because it is a cumbersome activity, and tools can automate it.

To this aim we applied the same approach proposed for generating languages. However, instead of only using a grammar as input, it takes both the program and the grammar. We use the grammar for creating a parser; then, the parser creates a parse tree of the program. Both Rascal and MPS offers support to write and read XML files, so we define an XML schema to serialize and deserialize parse trees as XML files. The former acts as an intermediate representation that supports the communication between platforms. The current approach is implemented in Rascal and MPS. However, it is possible to support other platforms by implementing the XML schema (Listing 12). In the textual world, the schema serializes the parse tree; while on the projectional world, the projectional LWB deserializes the XML and uses it to create the projectional model.

The current approach uses the XML file as the input of an MPS plugin. The plugin traverses the XML tree and creates a model that conforms with the generated

Listing 12: Simplified XML schema for exchanging information between LWBs.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="root">
    <xs:element name="nonterminal">
      <xs:element type="xs:string" name="name"/>
      <xs:element name="production" maxOccurs="unbounded" minOccurs="0">
        <xs:element type="xs:string" name="name"/>
        <xs:element name="arg" maxOccurs="unbounded" minOccurs="0">
          <xs:element type="xs:string" name="name"/>
          <xs:element type="xs:string" name="type"/>
          <xs:element type="xs:string" name="cardinality"/>
        </xs:element>
        <xs:element name="layout">
          <xs:choice maxOccurs="unbounded" minOccurs="0">
            <xs:element name="ref">
              <xs:element type="xs:string" name="name"/>
              <xs:element type="xs:string" name="type"/>
            </xs:element>
            <xs:element name="lit">
              <xs:complexType mixed="true">
                <xs:element type="xs:string" name="name" minOccurs="0"/>
                <xs:element type="xs:string" name="type" minOccurs="0"/>
              </xs:complexType>
            </xs:element>
          </xs:choice>
        </xs:element>
      </xs:element>
    </xs:element>
    <xs:element name="keywords">
      <xs:element type="xs:byte" name="keyword"/>
    </xs:element>
    <xs:element name="lexical">
      <xs:element type="xs:string" name="name"/>
      <xs:element name="arg">
        <xs:element type="xs:string" name="name"/>
        <xs:element type="xs:string" name="type"/>
      </xs:element>
    </xs:element>
    <xs:element type="xs:string" name="startSymbol"/>
  </xs:element>
</xs:schema>
```

language. If the translation is correct, the generated model should be a valid instance of the generated projectional language.

4.5 Architecture

We divide the bridge between textual and projectional LWB into five components, namely Rascal2XML, XML2MPS, *XMLImporter*, *ImportLanguage*, and *ImportProgram*. The solution has been implemented using Rascal MPL and Jetbrains MPS. In the following paragraphs we describe each one of these components and how to they interact with each other. All the code is available on a GitHub repository[1].

Rascal2XML. This module is written in Rascal, and it is responsible for generating an XML representation of Rascal grammars and existing textual programs. This modules produces an XML file that is used as input for the module XML2MPS.

XML2MPS. This MPS project holds the logic for generating MPS language definitions and model instances. It is responsible for creating MPS concepts and interfaces from an XML file. This library is used by both, `ImportLanguage` and `ImportProgram`.

ImportLanguage. is an MPS plugin that enables the import of languages. It creates the user interface of the plugin. In addition, it takes as input the grammar in XML format, calls the *XMLImporter*, and produces a projectional language.

ImportProgram. is an MPS plugin that enables the import of programs. This plugin takes as input an XML file that contains a program, and it produces a projectional model. To create the projectional model, this plugin relies on the XML importer to read the XMLFile and in XML2MPS to create the MPS nodes.

XMLImporter is a Java library for traversing the tree-like content of the XML files. This is used in both, the mapping of textual languages to projectional languages and for translating textual programs as projectional models.

5 Case Study

In this section, we present a case study to evaluate our approach. The language we have chosen for this purpose is JavaScript (ECMAScript 5). First, we explain the definition of the language. Then, we show how we create a mapping between the textual language and the generated projectional language. Afterwards, we generate a projectional editor based on the language's concrete syntax. Finally, we import existing textual programs as valid MPS models that conform to the generated projectional language. We conclude this section with a brief discussion about our case study results.

[1] https://github.com/cwi-swat/rascal-mps

5.1 Language Description

So far we have presented a way of applying the approach to a toy language of expressions, now we are going to apply it to a well-known and widely used language. To show the applicability of the approach to a real world language, we reused the existing grammar definition for JavaScript, included in the standard library of Rascal. The this grammar can be found on GitHub[2]. The objective of this evaluation is to use a Rascal implementation of the JavaScript grammar, and to obtain the equivalent language in MPS.

To do so, we first have to sanitize the existing grammar to meet the constraints of our solution Section 6. It is important to mention that this sanitization process is entirely manual. In this grammar, the sanitization process consists of adding labels to all the production rules and variable names to all symbols; and changing lexicals to use either one of our predefined lexical types or a user-defined construct. The resulting sanitized grammar can be found on GitHub[3].

We then used this grammar as input to generate the XML that encodes the grammar definition into the intermediate format. This XML representation is also available on GitHub [4]. The XML file can then be imported into MPS. In MPS, we use the plugin that we built, and we use the XML file as an input to successfully generate the projectional version of JavaScript.

To evaluate our generated version of JavaScript, we decided to compare it against an ad-hoc MPS implementation of such language called *EcmaScript4MPS* [5]. *EcmaScript4MPS* is a fine-tuned implementation of JavaScript based on domain knowledge. In other words, the implementation considers how developers use JavaScript editors. For comparing both implementations, we show several examples of language elements and programs of both implementations. For the rest of this section, we will refer to the generated version as JsFromRascal and the MPS ad-hoc implementation as JsManual.

5.2 Editor Aspect

To compare the editor aspect of both languages we will present how a program is displayed in both editors. The program for the JsFromRascal version was created using the importing of textual programs feature. This feature takes a program, parses it, and produces an XML file that contains the resulting parse tree. We did not make any custom change to JsFromRascal, we used the default generated version. The resulting projectional representation of the textual program using the JsFromRascal editor is shown in Figure 4. The program for JsManual was written by hand, and

[2] https://github.com/usethesource/rascal/blob/master/src/org/rascalmpl/library/lang/javascript/saner/Syntax.rsc

[3] https://github.com/cwi-swat/rascal-mps/blob/master/Rascal2XML/src/Grammars/JS/JSGrammar2.rsc

[4] https://github.com/cwi-swat/rascal-mps/blob/master/Examples/JS_Grammar.xml

[5] https://github.com/mar9000/ecmascript4mps

```
function
  substrings
(
  str1
)
{ var
    array1 = []
  ;
  for
  ( var
    x = 0
    y = 1
  ;
    x < str1 . length
  ;
    x ++
    y ++
  )
    { array1 [ x ]
      = str1 . substring ( x
        y
      )
      ;
    }
  var
    combi = []
  ;
  var
    temp = ""
  ;
  var
    slent = Math . pow ( 2
      array1 . length
    )
  ;
  for
  ( var
    i = 0
  ;
    i < slent
  ;
    i ++
  )
    { temp = "" ;
      for
      ( var
        j = 0
      ;
        j < array1 . length
      ;
        j ++
      )
        { if
          ( ( i & Math . pow(2,j)
            )
          )
            { temp += array1 [ j ]
              ;
            }
        }
      if
      ( temp !== ""
      )
        { combi . push ( temp
          )
          ;
        }
    }
  console . log ( combi . join("\n")
  )
  ;
}
```

Fig. 4: The substring JavaScript program displayed using the JsFromRascal editor.

it reflects the exact same program we used for JsFromRascal. Figure 5 shows the resulting program using JsManual.

As can be seen from Figures 4 and 5 the program in the JsFromRascal editor takes up more lines of code than its counterpart in JsManual. The JsManual editor makes the program look more readable due to the application of heuristics to capture break lines and whitespaces. This ends up splitting statements and expressions into several lines. The JsManual does not break this language constructs into several lines. However, it forces users to define variables outside *for* statements due to the way the language deals with variable identifiers.

Another difference between the editors is the usage of the dot (.) operator. This operator is often used in programming languages to get access to fields or methods

```
program substring
-----------------------------------------------
function substring(str1) {
  var array1 = [],
      x = 0,
      y = 1;
  for (; x < str1.length; x++) {
    array1[x] = str1.substring(x, y);
  }
  var combi = [];
  var temp = '';
  var slent = 'Math.pow(2, array1.length)';
  var i = 0;
  for (; i < slent; i++) {
    var temp = '',
        j = 0;
    for (; j < array1.length; j++) {
      if (i & 'Math.pow(2,j)')
        {
          temp += array1[j];
        }
    }
    if (temp !== '')
      {
        combi.push(temp);
      }
  }
  'console.log(combi.join("\n"))';

}
```

Fig. 5: The substring JavaScript program displayed using the JsManual editor.

in an object. JsFromRascal identifies it as a normal binary operator (e.g., '+', '-') and therefore the editor introduces a whitespace before and after the dot. This is an example of the limitations introduced by the heuristics, they are rigid. To make such heuristics more flexible, a language's domain knowledge is required to be able to treat such special edge cases in a different manner.

In sum, the JsManual editor is more appealing, and visually, it looks more like a textual program written using a plain text editor than the one generated using JsFromRascal. This kind of difference was expected because the domain knowledge was applied directly to the JsManual implementation, while the JsFromRascal tries

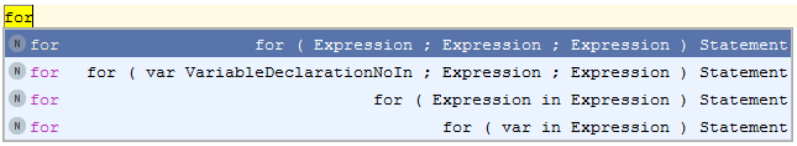

Fig. 6: JsFromRascal editor tab-completion menu of a *for* loop

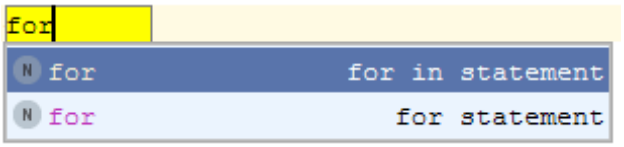

Fig. 7: JsManual editor tab-completion menu of a *for* loop

to provide a generic solution for any language. The knowledge of the JsFromRascal depends entirely on the information contained in the grammar and the set of heuristics applied to such grammar. On the one hand, the creation of ad-hoc editors from scratch, such as the one made for JsManual, is a cumbersome activity. On the other hand, a generated editor might speed up the development process of projectional editors because there are some structures that can be generalised, and then a developer can focus solely on the edge cases based on the domain knowledge of the language and their coding styles.

Program's Usability

Now we are going to discuss the usability aspects of both editors. Here we only focus on the ease of creating and editing programs with the aforementioned editors. First, we look into the tab-completion menu, which is one of the key aspects of a projectional editor since it allows users to navigate through the language's structure (AST). In Figure 6 we present a code completion menu for a *for* statement in JsFromRascal, and in Figure 7 we present the equivalent using JsManual. Both editors show similar information: concept's name and a brief description. However, the JsFromRascal editor displays also the structure of the child nodes of such concept, which might be useful for developers to understand how to use concepts or just to remember the concept's syntax.

5.3 Discussion

Projecting Grammars as Language Structures.

The first goal and building block for this project was to be able to recreate the structure of a language in two different LWBs. This goal was previously achieved and explained by Ingrid [18]. We wanted to try a different solution in which we do not directly integrate both platforms, but instead we define an intermediate format to make the solution more general. In Section 4.1 we describe the process for mapping a textual language definition into a projectional language definition. As shown in Section 5.1 the current approach works, yet there are some considerations that must be taken into account to generate a proper language. We understand that the way we treat lexicals might be cumbersome since the mapping of complex structure has to be manually defined. We also think this could be solved by defining some pre-processing strategies to capture lexicals and generate them into the second platform.

Editor Aspect – Language Usability

The editor aspect of a language is very important because it is the user interface to the language. Nevertheless, implementing a good editor is cumbersome. As shown in Section 5.2, usability is one of the main differences between ad-hoc and generated implementations. In the generated version we applied heuristics from the literature (e.g., well-known formatting and pretty printing approaches) to try to identify production rule patterns in a generic way. However, these heuristics have limited power and of course they might not fit every language, especially if we compare them against custom implementations. Nonetheless, with the current approach we show that it is possible to apply existing heuristics to create projectional editors based solely on the language's concrete syntax. In addition, the current approach considers the language's structure to generate a projectional editor that in some cases might be more appealing than the reflective MPS editor.

To improve the current approach we could have implemented more heuristics or define a mechanism for customizing them. We might also require additional information other than the information contained in the grammar. In addition, domain knowledge and user feedback is very important to generate a better editor. In other words, we need more information to be able to apply the heuristics in a less rigid fashion, and therefore improve the editor generation.

6 Limitations

In this section, we discuss limitations of the approach, the rationale behind them, and possible solutions to overcome them. These limitations are based on assumptions

and constraints in the grammar. In addition, there is also a technical limitation related to how the mapping is implemented.

Summary - Grammar Preconditions

- Non-terminal symbols name and production rules labels within a grammar must be unique.
- Symbol labels within a production rule must be unique.
- Lexicals can be either one of the MPS predefined data types or the lexical must be defined by hand using the lexical library.
- Each production rule and each symbol within a production rule must be labeled.

1. The names of the non-terminal symbols in a grammar must be unique. In other words, the current approach does not support the definition of two concepts with the same name. The rationale behind this is that the name of a non-terminal symbol is used to define an interface concept in the generated MPS language and the production rules labels are used to create concepts. One way to avoid this constraint could be to define a renaming scheme that is able to detect and fix name conflicts. However, this solution might introduce a side effect on the usability of the language; projectional editors use these names for IDE services such as tab-completion, so they must be descriptive enough for end-users. In addition, other language components must be refactored according to the renaming mechanism. This is why we did not implement an automatic renaming scheme, and we preferred to include it as a limitation of the current approach.
2. In the mapping between a Rascal grammar and an MPS language, symbol labels are used as variable names, either for `children` or `references` in MPS concepts. These names should be unique within the same concept, yet not for the whole language. For instance, if we define concept *A* and *B*, both can contain a reference of a child named *name*, however *A* cannot have more than one child or reference called *name*. In other words, symbol labels can be reused across concepts but not within the same concept.
3. Lexicals are a challenging concept to deal with because there is no standard way of defining them. However, it is possible to make some assumptions on regularity and define a set of constraints to translate lexical between platforms in an automatic way, but this requires considerable effort.As a result, we did not want to restrict the usage of regular expressions, so we included lexicals that represent MPS built-in types (e.g., string, int) to the lexical library. The current approach does not limit users from defining custom lexicals. However, users have to manually define a mapping between the custom lexical defined in Rascal and the right translation for MPS. Section 4.1 describes the details on how to support custom-defined lexicals.
4. It is required to label all the production rules and symbols within a production rule because the approach uses the labels for naming concepts or children reference

fields. A solution could be to generate placeholder names yet this introduces other issues such as non-descriptive names and name matching issues when importing existing textual programs.
5. The current approach does not take advantage of name resolution, especially for code completion, which is a keystone for projectional LWBs. For instance, in MPS, concept hierarchies do not rely on the definition of trees; instead, they use graphs.
6. The current implementation supports the mapping of lists and separated lists of symbols into MPS language concepts (editor and structure aspects). However, the mapping for separated lists is partially implemented. The current approach treats separated lists just as a list. As a result, the separator symbol is ignored for the generation of the editor.

Support different versions – Versioning/Language evolution

The current approach does not support language nor program evolution. In other words, the current approach consider languages as a standalone units. It does not consider that changes might happen to the language. For example, if a developer uses a textual language A and generates a projectional language $A*$ inside MPS, the current approach only accepts valid programs according to A. If there are changes to the original language A, those changes cannot be patched in the generated versions. This forces either to re generate the whole language from scratch or make changes by hand. There are some changes that do not break the importing of programs, namely,

- Addition of language constructs to the grammar, and then using them in a program. This means that the plugin for importing programs, *ImportProgram* (Section 4.5), will not be able to find such elements. As a result, the plugin notifies the user.
- Modification of existing language constructs (e.g., adding or removing parameters). As expected, this type of change often ends-up in a failure.

In sum, language engineers and users in general should be aware of the language's version and the version used to define programs. We see this problem as an opportunity for future extensions of the current approach in order to support the evolution of languages and programs.

7 Related Work

Grammar to model

A major component of this thesis is the generation of models from textual grammars. In order to place ourselves within the current state of the art we identify the following related work.

The goal of the Ingrid [18] project quite similar ours: bridging the gap between the textual and projectional language workbenches. In their case, they chose the same projectional language workbench, Jetbrains MPS, but a different textual meta-language in ANTLRv4[19]. Ingrid has an implementation of their solution as a hybrid Java/MPS project.

Ingrid approaches the problem in three steps: first, the grammar must be parsed and relevant information about the structure and other required language elements is stored as linked Java objects. Secondly, the stored structure is traversed and equivalent MPS model nodes and interfaces are constructed. Finally, an editor is generated for each MPS Language Concept Node.

There are many high-level similarities between Ingrid and Rascal2MPS. The steps taken for parsing, gathering information about the language, generating an intermediate structure to represent the language, and finally generating a model from said intermediate structure are performed by both projects. The main differences are in the details, and the choices made in the implementations of both projects, which have various consequences for the use of the respective tools.

The main architectural difference is in the choice of intermediate structure. Whereas we chose for an external file-based format (see Section 4.5), Ingrid uses an internal representation of linked Java objects. This is enabled by the use of the an ANTLRv4 parser implemented in Java and the ability of MPS to directly call into Java executables. Thus, the Ingrid MPS plugin is able to call the parser and start the process of data extraction internally. This is in contrast to Rascal2MPS, where the projectional language workbench and textual language workbench are kept completely separate, only communicating through an external intermediate format. There are several advantages to the Ingrid's integrated approach:

- The solution becomes a one-step process, making it more efficient for the language engineer.
- All implementation is done on one side of the gap (The projectional language workbench). This simplifies development.
- The language engineer does not need to maintain both the textual and projectional language workbench.

However, there is also a major downside to this approach which is why we ultimately decided against this it. Since the projectional language workbench has to call the grammar parser directly, there is a strong binding between the projectional language workbench and the specific grammar parser. In the case of Ingrid, the MPS plugin calls into the Java ANTLR parser. However, ANTLR is far from the only parsing tool. If we now wished to extend Ingrid for a new textual language workbench, such as Rascal, we would need to replace the calls to the ANTLR parser with calls to a Rascal parser.

This can lead to several problems:(i) The architecture has to allow this replacement. This can be partially solved with smart use of interfaces and abstraction over the parser, but the problem of potentially different APIs remains. In the worst case, a complete mapping from ANTLR parser function calls to Rascal parser function calls would have to be made. (ii) The parser needs to be implemented in Java. ANTLRv4

already has a Java-based parser available, and thus is a prime candidate for integration with the also Java-based MPS. However, this is not necessarily true for any given textual LWB. If one is not available, the language engineer would either have to create one in Java, or find some way to expose the parser of specific LWB to a Java environment.

Part of the original problem statement for Rascal2MPS was to create a more general approach to the problem of bridging the gap between the textual and projectional worlds. This also extends to the implemented tool. Neither side of the solution is aware of the other; only of the intermediate file-based format. This format serves as a contract between the different parts of the solution. Thus whether the intermediary file was generated by from an ANTLR-, Rascal- or Xtext-based grammar is irrelevant to the implementation on the side of the projectional language workbench.

Another difference between Ingrid and Rascal2MPS lies in the Editor generation within MPS. While Ingrid does identify the problem of usability of the reflective editor and discusses several solutions, such as heuristics or prompts during the import process, they have not been implemented. Ingrid only generates an editor containing the structural elements of the node, i.e. the literals and references to other nodes. It is then left up to the language engineer to manually apply whitespace to the editor. Rascal2MPS goes further and applies heuristics to automatically apply whitespace during the import process. While this does not fully eliminate the need to manually edit the editor definitions (See Section 5), it can save time given the right set of heuristics.

Finally, Ingrid does not address the problem of language artifacts, i.e. programs created within the textual world. Thus even after a language has been imported, programs written using said language in the textual language workbench need to be manually recreated as MPS models of the imported language. Rascal2MPS does implement the ability to construct MPS program models using textual source code.

Wimmer et al. [20] describe a generic semi-automatic approach for bridging the technical space between Extended Backus-Naur form (EBNF), a popular grammar formalism, and Meta-Object Facility (MOF), a standard for model driven-engineering. In this approach an attributed grammar, which describes EBNF and the mapping between EBNF and MOF, is used to generate a Grammar Parser (GP). This GP can then be used to generate MOF meta-models from grammars. This is not far from the approach implemented by this thesis. However, this approach fixates on MOF as the target meta-model directly. In the case of going between language workbenches in separate worlds, we do not want to be specific in the target. Instead, out approach focuses on the contract in the form of the intermediate format, and makes the source and target formalism up to implementation.

Another downside of the given approach is that it requires grammar annotations and additional manual improvements of the generated model in order to refine the generated model. We seek to limit the actions of the language engineer, especially concerning the source grammar. The Gra2Mol [21] is another project which seeks to bridge the gap between the textual grammar and model worlds. The authors define a domain-specific model transformation language which can be applied to some source program which conforms to a grammar, and generates a model which conforms to

a target meta-model. This language can be used to write a transformation definition consisting of transformation rules. In this way, the presented approach abstracts over the generated meta-model, which would be quite useful in our use-case, as we would be able to simply give the meta-model of the target language workbench as input with the transformation definition. In practice however, this runs into problems when the desired target model is specific rather than generic. For example, the standard storage format for Jetbrains MPS is a custom XML format. The models contain much information tied specifically to MPS, such as the node ID's, layout structures etc. Generating these from outside of MPS would be quite tedious, and also would introduce a dependency on the MPS model format, which may change in the future. It is thus best to interact with MPS model from within MPS itself, where MPS can do the heavy lifting of generating the models.

Editor generation

Another aspect of the project is the generation of the Editor Aspect, or the visual representation of language model elements. This is closely related to the pretty printing problem.

Van de Vanter et al [22] identifies part of the core problem between the textual and model-based approach. From a system's perspective, a model-based editor allows for easier tool integration and additional functionality. However, language users are often more familiar and comfortable with text-based editing. In this paper the authors propose a compromise based on lexical tokens and fuzzy parsing. This is not unlike what is offered by MPS. MPS Editors are highly customizable and can be made to closely resemble the text-based editing experience.

The BOX language for formatting text as introduced by Van den Brand et al [23] is closely related, as the heuristics for generation white space between language elements is reused in this project. The BOX language is further used in other work on pretty printing of generic programming languages, such as GPP (Generic Pretty Printer) [24], which constructs tree structures of the layout of a language element, which can then be consumed by an arbitrary consumer.

Syntax-directed pretty printing [25] also identifies several structures for creating language-independent pretty printers. In this approach a grammar extended with special pretty printer commands is used as input to generate a pretty printer of for that specific language. The generated pretty printer can then be reused for any program written in the language the pretty printer was generated for. The annotated grammar approach does limit the form the final pretty printer can have due to the lack of options. Also, annotating an entire grammar can be tedious work. In our approach, we attempt to limit the required user interaction with the source grammar, although we did not eliminate it entirely.

Following this line of research, Terrence et al [26] propose *Codebuff*, which is a tool for the automatic derivation of code formatters. *Codebuff* is a generic formatter that uses machine learning algorithms to extract formatting rules from a corpus. This is a neat approach because, as we mentioned before, source code formatting

is subjective, it depends on the style of each programmers, and it changes across languages. For example, in Section 5.2, we showed that applying the same heuristics for any language does not always produce a good editor. Therefore, we consider tools like *Codebuff* as inspiration for future work. We could benefit from their techniques and knowledge to generate editors in a flexible and highly configurable way, and perhaps to learn from existing source code examples.

8 Conclusions and Future Work

In this chapter we presented an approach to bridge the gap between textual and projectional LWB. In concrete, we defined a mapping between textual grammars and projectional meta models; this mapping (Section 4) produces the structure and editor aspects of a projectional language. Moreover, our approach allows users to reuse textual programs by means of translating them to equivalent MPS models(Section 4.4). To validate our solution, we used as a case study a Rascal grammar of JavaScript (Section 5). Based on the grammar definition we generated a projectional version of JavaScript. To verify the correct mapping of the generated language, we successfully imported existing valid textual JavaScript programs into MPS. In Section 6, we discuss some of the limitations of the current approach.

Language evolution is a key aspect to look at in the future. Since the current approach assumes that the generation is done only once. However, we ignore the fact that the textual language and the projectional generated version might change as well. Then we consider that a way of keeping track of these changes and being able to transfer/apply this changes to the other is important. If there are changes in the grammar after the projectional language generation, developers have to regenerate the whole language, and this may lead to losing information (if changes were made on the generated language).

Similarly, this applies to programs written in such languages. We consider that a mechanism for maintaining both versions is worth to investigate as future work to be able to keep a bidirectional mapping. Language engineers can switch from one platform to another without losing information. Our approach offers support for a unidirectional mapping from textual to projectional. We believe that a bidirectional communication is required. Because depending on the language one may benefit more from having a textual or a projectional version of the language. Therefore, to support changes on both sides we require a bridge to create a textual language from a projectional language. Moreover, to complete the circle, a way of keeping track and propagating changes in both worlds will be required. To avoid losing or reimplementing existing features.

As we described in Section 5.3, the usability of generated editors is one of the key aspects that should be addressed in future research. We found that we can generate editors with limited capabilities (that do not consider domain knowledge or existing formatters). Therefore, we consider as future work, to explore artificial intelligence techniques (e.g., machine learning or programming by example) to

improve the existing editor (in the style of [26]), maybe by identifying patterns in existing programs or commonalities in the grammar's structure to guide or to customize the generation of the editor aspect.

References

1. Sebastian Erdweg, Tijs van der Storm, Markus Volter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriel Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24 – 47, 2015.
2. Ralf Lämmel. *The Notion of a Software Language*, pages 1–49. Springer International Publishing, Cham, 2018.
3. Martin Fowler. Language workbenches: The killer-app for domain specific languages?, 2015.
4. Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.
5. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 12 2005.
6. J.G.M. Mengerink, B. van der Sanden, B.C.M. Cappers, A. Serebrenik, R.R.H. Schiffelers, and M.G.J. van den Brand. Exploring dsl evolutionary patterns in practice: a study of dsl evolution in a large-scale industrial dsl repository. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, pages 446–453. SCITEPRESS-Science and Technology Publications, Lda., 2018.
7. Jur Bartels. Bridging the worlds oftextual and projectionallanguage workbenches. Master's thesis, Eindhoven University of Technology, 1 2020.
8. Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society.
9. Maurizio Gabbrielli and Simone Martini. *How to Describe a Programming Language*, pages 27–55. Springer London, London, 2010.
10. Fabien Campagne and Fabien Campagne. *The MPS Language Workbench, Vol. 1*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 1st edition, 2014.
11. CWI-SWAT. Syntax definition, 2020.
12. Véronique Donzeau-Gouge, Gérard Huet, Bernard Lang, and Gilles Kahn. Programming environments based on structured editors : the mentor experience. *Interact Program Environ*, 01 1984.
13. A. J. Mooij, J. Hooman, and R. Albers. Gaining industrial confidence for the introduction of domain-specific languages. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, pages 662–667, 2013.
14. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.
15. Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.
16. Istvan Nagy, Loek Cleophas, Mark van den Brand, Luc Engelen, Liviu Raulea, and Ernest Xavier Lobo Mithun. Vpdsl: A dsl for software in the loop simulations covering material flow. In *Proceedings of the 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, ICECCS '12, page 318–327, USA, 2012. IEEE Computer Society.
17. Jacques Verriet, Hsuan Lorraine Liang, Roelof Hamberg, and Bruno van Wijngaarden. Model-driven development of logistic systems using domain-specific tooling. In Marc Aiguier, Yves Caseau, Daniel Krob, and Antoine Rauzy, editors, *Complex Systems Design & Management*, pages 165–176, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

18. Premysl Vysokỳ, Pavel Parízek, and Václav Pech. Ingrid: Creating languages in mps from antlr grammars. 2018.
19. ANTLR. https://www.antlr.org/.
20. Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In *International Conference on Model Driven Engineering Languages and Systems*, pages 159–168. Springer, 2005.
21. Javier Luis Cánovas Izquierdo, Jesús Sánchez Cuadrado, and Jesús Garcıa Molina. Gra2mol: A domain specific transformation language for bridging grammarware to modelware in software modernization. In *Workshop on Model-Driven Software Evolution*, pages 1–8, 2008.
22. Michael L. Van de Vanter, Marat Boshernitsan, and San Antonio Avenue. Displaying and editing source code in software engineering environments. 2000.
23. Mark Van Den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):1–41, 1996.
24. Merijn De Jonge. Pretty-printing for software reengineering. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 550–559. IEEE, 2002.
25. Lisa F. Rubin. Syntax-directed pretty printing—a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, (2):119–127, 1983.
26. Terence Parr and Jurgen Vinju. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, page 137–151, New York, NY, USA, 2016. Association for Computing Machinery.