Projecting Textual Languages

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

Abstract One of the strengths of the Jetbrains MPS projectional language workbench is that it supports mixing different kinds of notations (graphical, tabular, textual, etc.). Many existing languages, however, are fully textual and are defined using grammar technology. To allow such languages to be used from within MPS, language engineers have to manually recreate the syntax of a language using MPS concepts. In this chapter, we present an approach to automatically convert grammar-based languages to MPS languages, by mapping context-free grammars to MPS concept hierarchies. In addition, parse trees of programs in those languages are mapped to MPS models. As a result, MPS users can import textual languages and their programs into MPS without having to write tedious boilerplate code. We have implemented the approach in a tool, Rascal2MPS, which converts grammars in the built-in grammar formalism of Rascal to MPS. Although the tool is specific for the Rascal context, the underlying approach is generic and can be instantiated for other grammar formalisms. We have evaluated Rascal2MPS by generating an importer for a realistic programming language (ECMAScript 5). The results show that useable MPS editors for such languages can obtained but that further research is needed to improve their layout.

M. V. Merino (✉) · J. Bartels · M. van den Brand
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: m.verano.merino@tue.nl; j.bartels@student.tue.nl; M.G.J.v.d.Brand@tue.nl

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

E. Schindler
Canon Production Printing, Venlo, The Netherlands
e-mail: eugen.schindler@cpp.canon

A. Bucchiarone et al. (eds.), *Domain-Specific Languages in Practice*,
https://doi.org/10.1007/978-3-030-73758-0_7

1 Introduction

Language workbenches [1] (LWBs) 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 software engineering problems through the use of multiple domain-specific languages (DSLs) [3]. DSLs are small and simple languages tailored to solve problems in a particular application domain [4]. There are two types of DSLs, internal and external [3]. The first one reuses the concrete syntax of the host language and its parser, much like a stylized library. An external DSL, however, typically requires the implementation of a parser and compiler.

Jetbrains MPS is a projectional language workbench that obviates the need for parsing and, as a result, allows the engineer to define DSLs with a multiplicity of notations, varying from textual, and tabular, to diagrammatic, or prose-like. MPS provides editor support that allows users to directly edit the abstract syntax structures of a language rather than reconstructing such structure from the linear sequences of characters entered in text editors.

Nevertheless, many existing languages are defined purely textually. For instance, all mainstream programming languages are textual (e.g., Java, C#, Javascript etc.). But many DSLs, like GNU Make, Graphviz, SQL, etc., are strictly textual languages too. To make such existing languages available for (re)use from within MPS, language engineers have to redefine the syntax of such languages using the concepts and editor features of MPS, which is a tedious and error-prone endeavor.

In this chapter, we detail an approach to take an existing context-free grammar (e.g., from a parser generator tool) of a textual language and convert it automatically to MPS concept definitions. As a result, such languages can be imported into MPS without having to write abstract syntax definitions by hand. Furthermore, the approach supports loading parse trees of existing programs into automatically generated MPS editors, so that they become available for reuse immediately.

Companies in the Eindhoven (The Netherlands) region (e.g., Canon Production Printing and ASML) have been using DSLs for several years [5]. Some of these companies use textual LWBs, projectional LWBs, or both, such as Canon Production Printing. 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 a reuse facility exists, companies will avoid the costs of reimplementing features and maintaining the same functionality in different platforms. Likewise, developers can be more productive from the engineering point of view and invest more time in developing new features or improving existing ones. Finally, the reuse strategy could reduce time to market for new products.

In this chapter, we present an approach toward bridging the gap between textual and projectional LWBs, which has been implemented in the context of the Rascal (textual) and MPS (projectional) language workbenches. Our Rascal2MPS [6] takes a Rascal grammar and converts it to equivalent concept hierarchies and editor definitions in MPS.

The contributions of this chapter can be summarized as follows:

- A generic bridge between textual and projectional LWBs. Employing 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: in Sect. 2, we describe the motivation that supports this work and the problem statement. Then, in Sect. 3, background information about software language engineering is presented. In Sect. 4, we present our solution and its architecture. Then, we evaluate the current approach by comparing an ad hoc implementation of JavaScript against a generated version (Sect. 5). In Sect. 6, we discuss the limitations of the current approach. We conclude this chapter with related work and future research directions (Sects. 7 and 8).

2 Motivation

A DSL offers programming abstractions that are closer to domain requirements than general programming languages [7]. Likewise, DSLs offer syntax closer to the domain expert's knowledge. DSLs have been around for a few decades, but they have not been widely adopted in the industry yet [8, 9]. The limited adoption of DSLs in the industry is partly due to the lack of mature tools [10, 11].

Nowadays, language engineers have different tools and metalanguages to choose from when implementing 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 Production Printing that they have been developing and maintaining over the years. However, they have more recent languages that were developed using a projectional LWB. They have recently found that they require to interoperate languages, which means reusing language concepts across LWBs. This interoperation allows them to address new business needs and reduce the time to market. Therefore, they demand a bridge that supports the reuse and translation of existing languages across heterogeneous LWBs.

3 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.

3.1 Software Language Engineering

Software Languages A software language is a means 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, 12]. A language's syntax is a set of rules that define valid language constructs, such as defining a group of rules that captures expressions or statements. The language's syntax can be expressed in a concrete and abstract way. The concrete syntax is designed as the user interface for end users to read and write programs, whereas the abstract syntax is the interface to the language implementation. The semantics of a language is a mapping between syntactic elements and their meaning. 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) LOP is an approach to software development where the main activity in development consists of defining and applying multiple DSLs [3, 13]. 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 or aspects of a domain problem so that the productivity increases and domain knowledge is decoupled from implementation concerns. In other words, a DSL captures the "*what*" of the domain, whereas compilers, code generators, and interpreters define the "*how*."

Language Workbench To help language engineers develop software languages, they 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, 14]. 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 [15] for defining the language's syntax. A projectional LWB allows a program's AST to be edited directly [16]. For instance, MPS uses an AST Concept Hierarchy [14] 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.

3.2 Syntax of Textual and Projectional Languages

As mentioned before, a software language's syntax is a set of rules that describe valid programs [2]. 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, a language's concrete syntax 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.

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 N is shown below:

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

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

Language	Rascal	MPS
Concrete syntax	Context-free grammar	Projectional editor definition
Abstract syntax	Algebraic data rype	AST concept hierarchy

The production rules P are defined as follows:

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

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

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

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

By applying the previous production rules, we can write the arithmetic expression $a + b$ (where $a, b \in \mathrm{N}$) 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 the 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. 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`**).

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 standard formalism for defining abstract syntax; therefore, each platform provides its own formalism. MPS uses a node

Listing 1 Concrete syntax of addition and numbers in Rascal

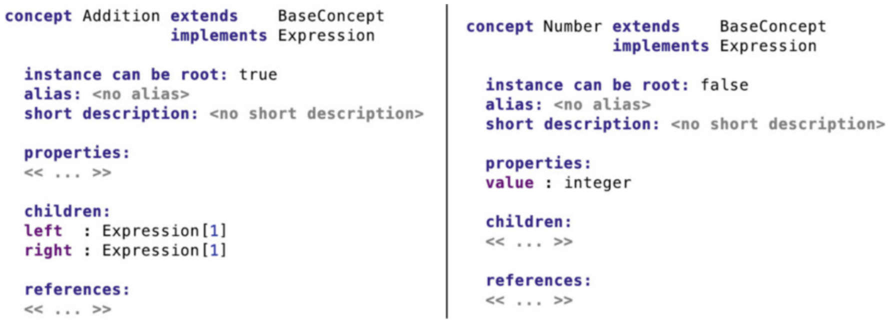
```
start syntax Exp = number: Nat nat | addition: Exp lhs "+" Exp rhs;

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);
```



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

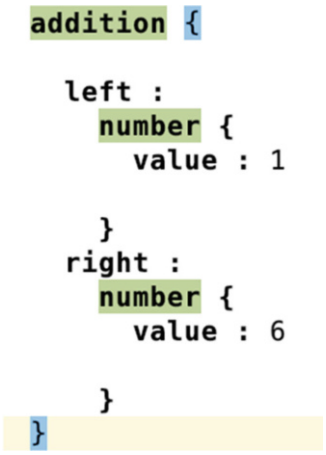
Fig. 1 Concept definition of addition (left) and numbers (right)

concept hierarchy [14]. For instance, the AST representing a language for describing the addition of natural numbers is shown in Fig. 1. The MPS implementation uses an *Expression* interface and two concepts, namely, *Addition* and *Number*. To represent integer 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 Fig. 2.

4 Approach: Projecting Textual Languages

This section presents a mechanism for enabling textual languages usage in a projectional editor by generating a projectional language from a grammar. 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,



```
addition {

  left :
    number {
      value : 1

    }
  right :
    number {
      value : 6

    }
}
```

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

we explain a generic mapping between CFGs and the structure of a projectional language. Afterward, 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 language structure. 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.

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 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 MPS, interfaces are represented as concepts, and their instances are called nodes. 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.

Listing 4 Definition of the Exp interface in MPS

```
interface concept Exp extends <none>

  properties:
  << ... >>

  children:
  << ... >>

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

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

  children:
  expression :Exp[1]
```

Furthermore, one additional nonterminal that we have not mentioned is the start symbol. Structure concepts in MPS have a property named *instance can be root*. This attribute indicates whether the concept can be used to create an AST root node [14]. In our mapping, we take the start symbol of the grammar, and create a concept in MPS. This concept will have the property *instance can be root* 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 the *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 that are only relevant to concrete syntax, such as keywords and operator symbols, are not mapped here, since they are not part of the abstract syntax; they will be used to define the 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.

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

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

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:
  << ... >>
```

Lexicals Lexicals define the terminals of a language and are typically defined by 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 in their Rascal grammars, but it is also possible for users to include their lexicals. In this case, developers must describe the mapping to MPS manually.

Each lexical is mapped to a concept, like any other nonterminal, and a *constrained data type*. To illustrate this, Listing 1 contains `Nat`'s definition, which consists of a single production, called `digits`. This production rule references `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`, a *constrained data type* capable of capturing natural numbers using the regular expressions engine of MPS.

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

```
constrained string datatype: Natural

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

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:
  << ... >>
```

List of Symbols In CFG, it is possible to define a group of symbols of the same type, often expressed using Kleene's star ($\star$) and plus (+). Kleene's operators (star and plus) are unary operators for concatenating several 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 us 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.

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). come with the language.

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>
```

Literals Literal symbols may be part of productions to improve the readability of code or disambiguate. They form an essential aspect of the 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 its 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. Moreover, this is used to define the syntax highlighting of the resulting editor. The literals are displayed with a different color to show the users that they are reserved words of the language. As a result, the current approach offers a binary coloring scheme: keywords are blue and the remaining symbols in black. Instead, if it is a nonterminal symbol, we create a *reference*.

For example, the production rule that defines the addition between natural numbers has three symbols: *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, in the editor, the reference `lhs` creates 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 to 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 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 CFGs to the editor aspect in projectional languages, we will improve the generated projectional editor. The editor can be improved by pruning the grammar to enhance IDE services (e.g., auto-completion). To prune the

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>
```

grammar, we eliminate chain rules (also known as unary rules) from the productions. To eliminate the chain rules, we first collect all the productions with a single parent and are referenced once in the grammar. Then, we merge the single reference with its parent.

To illustrate this process, let's consider the following production:

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

Long production rules are often split into smaller production rules for readability. For example, a language engineer can also write the previous production as:

$$A \rightarrow A|B$$

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

The second alternative impacts the language's structure because it introduces a new nonterminal B. This new nonterminal is translated in the AST as an extra node. To illustrate the difference between both versions, Fig. 3 shows a tree view of the ASTs. From the right-most AST in Fig. 3, we observe that node B is referenced once in the language. Thus, production $A \rightarrow B$ represents a chain rule. This chain rule is translated to the end users as an extra keystroke to access the leaf nodes b, c, d via `B`. If we remove the chain rule, we avoid creating an extra node (B) before accessing the terminals (b, c, d) in the projectional editor.

For example, if users want to create a node b, they can call auto-complete, and they will obtain two options, A or B. Based on the AST shown in Fig. 3, they select to create a node `B`. However, they have not reached b yet. Thus, they must press tab completion again, and then they get all the options of B: b, c, and d. In contrast, if we prune the chain rule, meaning we remove concept B, we can omit the second tab completion because all the options will be visible from the first tab completion. Removing chain rules from a grammar impacts both the structure and the editor of a projectional language since removing a concept means the editor of such concept is no longer needed. As a result, we enhance the user's interaction with the projectional editor by removing the chain rules.

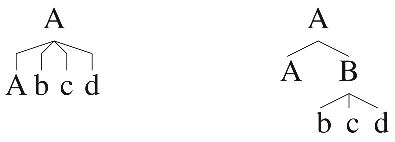


Fig. 3 Tree-based view comparison

4.4 Translating Textual Programs into Projectional Models

We extend the approach to translating existing textual programs into projectional models. This extension's motivation 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 cumbersome, 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 offer 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 in 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 plug-in. The plug-in traverses the XML tree and creates a model that conforms with the generated language. If the translation is correct, the generated model should be a valid instance of the generated projectional language.

4.5 Architecture

The approach to bridge textual and projectional LWBs contains five components: Rascal2XML, XML2MPS, *XMLImporter*, *ImportLanguage*, and *ImportProgram*. The solution has been implemented using Rascal MPL and Jetbrains MPS. We consider two different architectures for the implementation of the current approach. The first one was based on integrating Rascal directly into MPS, including Rascal as a Java library in MPS. This architecture allows us to call Rascal parsers directly from MPS. However, this approach does not allow reusability, and this integration should be repeated for any textual LWB. Instead, the second architecture uses an intermediate format to communicate between a textual LWB and MPS. In the

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>
```

following paragraphs, we describe each of the components of this architecture and how 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 module 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. Both `ImportLanguage` and `ImportProgram` use this library.

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

ImportLanguage This is an MPS plug-in that enables the import of languages. It creates the user interface (GUI) for importing a textual language. The GUI displays a pop-up that takes the grammar (in XML format) as input, calls the *XMLImporter*, and produces a projectional language.

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

XMLImporter This is a Java library for traversing the tree-like content of the XML files. This is used to map textual languages to projectional languages and translate 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) because there is an existing implementation of it for MPS, and it allows a proper validation of Rascal2MPS. 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. Afterward, 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. This section concludes with a brief discussion based on results.

5.1 Language Description

So far, we have presented a way of applying the approach to a toy language of expressions. Now we will 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 Rascal's standard library. This grammar can be found in GitHub.[2] This evaluation aims to use a Rascal implementation of the JavaScript grammar and obtain the equivalent language in MPS.

First, we must sanitize the existing grammar to meet our solution's constraints, as described in (Sect. 6). It is essential to mention that this sanitization process is entirely manual. In this grammar, the sanitization process consists of adding labels

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

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 plug-in 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 a language called *EcmaScript4MPS*.[5] *EcmaScript4MPS* is a fine-tuned implementation of JavaScript for MPS. In other words, the implementation considers how developers use JavaScript editors and the features offered for JavaScript in IDEs. To compare 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 of both languages, we present how a program looks like in both editors. The JsFromRascal program was created using the approach described in Sect. 4.4. This approach takes a textual program as input, and the tool parses it and produces an XML file with the resulting parse tree. It is important to mention that we did not tweak the resulting program; we used the generated version as is. Figure 4 shows the resulting program using the JsFromRascal editor.

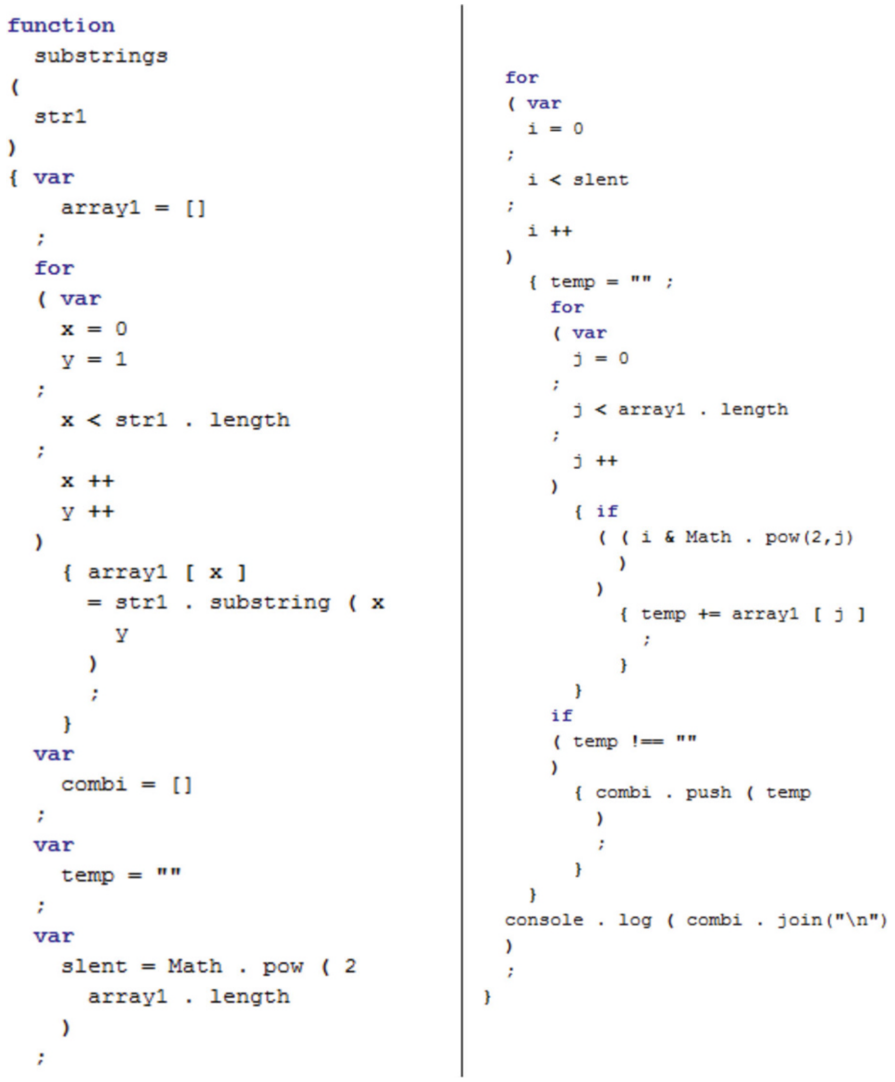
In contrast, the program for JsManual was written by hand because we did not have a mechanism, like the one described before, for arbitrary textual programs. However, the handwritten program is the same as the one used for JsFromRascal. The resulting program in the JsManual editor is shown in Fig. 5.

As can be seen from Figs. 4 and 5, the program in the JsFromRascal editor takes up more lines of code than its counterpart in JsManual. According to the JavaScript standards, the JsManual editor makes the program look more readable due to the ad hoc implementation of the editor, which places break lines and whitespaces in the right place. The JsFromRascal editor splits up statements and expressions into several lines based on the implemented heuristics. Instead, the JsManual editor does not break these language constructs into several lines. However, it forces users

[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.



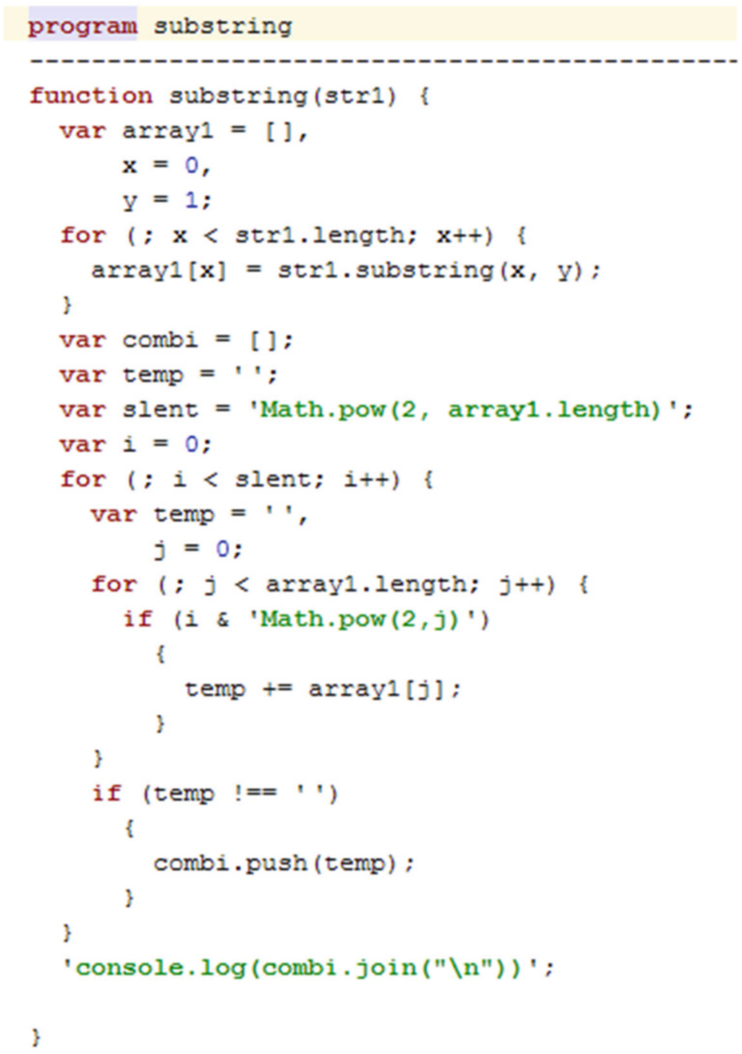
```
function
  substrings
(
  str1
)
{ var
    array1 = []
  ;
  for
  ( var
    x = 0
    y = 1
  ;
    x < str1 . length
  ;
    x ++
    y ++
  )
    { array1 [ x ]
      = str1 . substring ( x
        y
      )
      ;
    }
  var
    combi = []
  ;
  var
    temp = ""
  ;
  var
    slent = Math . 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

to define variables outside *for* statements due to the language's name resolution implementation.

Another difference between the editors is the usage of the dot operator (.). This operator is often used in programming languages to access fields or methods. For instance, JsFromRascal identifies it as a binary operator (e.g., "+," "−"), and therefore the editor introduces whitespaces before and after the dot operator. This is an example of the limitations introduced by the heuristics; they are rigid. A



```
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

customization mechanism might be needed to make such heuristics more flexible; thus, they can be adapted to different languages and scenarios.

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 JsManual editor is implemented in an ad hoc way to offer the best experience for this language, while the JsFromRascal editor is obtained through a generic tool that works for various languages. However, the JsFromRascal editor can be manually fine-tuned to achieve the desired editing experience. The knowledge of the JsFromRascal editor depends entirely on two core elements, 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 speeds up editors' development process because they use generic abstractions that can be applied to several languages, so that developers can focus on fine-tuning the generated editors on edge cases based on platform-specific features and the language's coding styles.

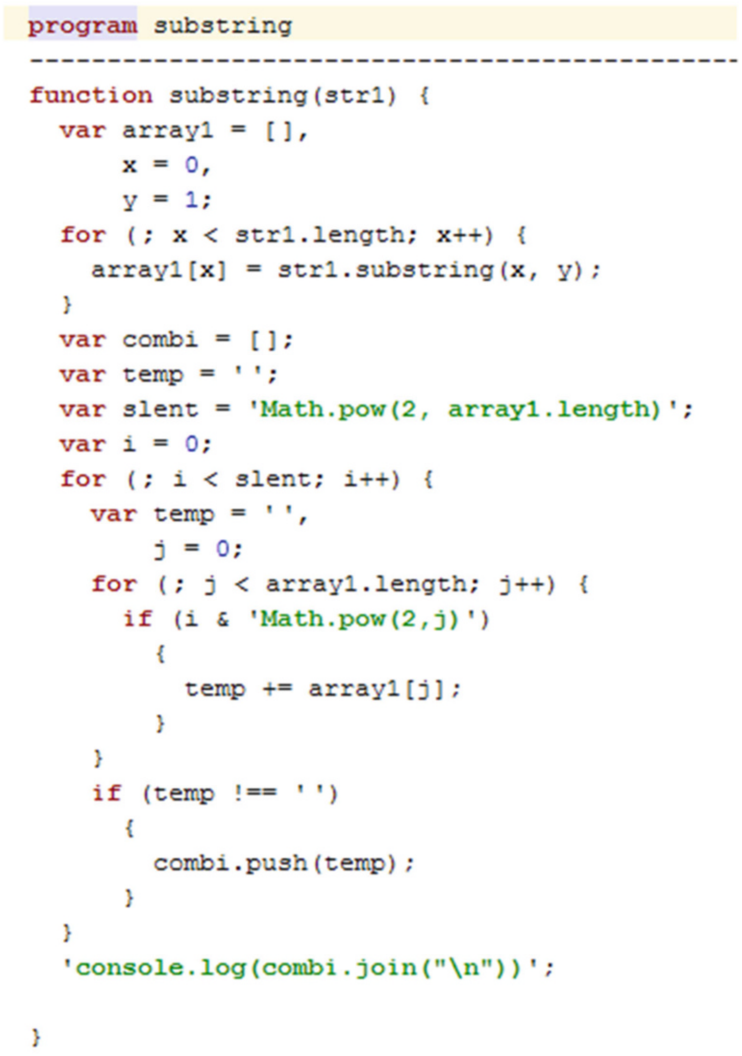
5.3 Program's Usability

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 editors mentioned above. First, we investigate the tab-completion menu, which is one of the critical aspects of a projectional editor since it allows users to navigate through the language's structure (AST). In Fig. 6, we present a code completion menu for a *for* statement in JsFromRascal, and in Fig. 7, we present the equivalent using JsManual. Both editors show similar information: the concept's name and a brief description. However, the JsFromRascal editor also displays the structure of the child nodes of such a concept, which might help developers understand how to use concepts or remember the concept's syntax.

5.4 Discussion

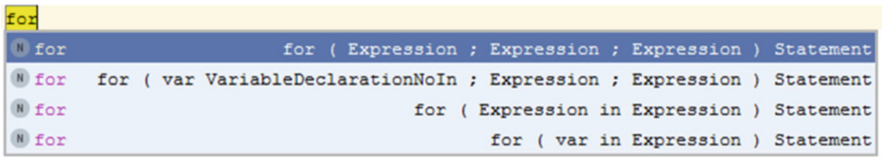
Projecting Grammars as Language Structures The first goal and building block for this project is to recreate the structure of a language in two different LWBs. This goal was previously achieved and explained by Ingrid [17]. We wanted to try a different solution in which we do not directly integrate both platforms, but instead, we define an intermediate format to make the solution more general. Section 4.1 describes the process for mapping a textual language definition into a projectional language definition. As shown in Sect. 5.1, the current approach works, yet some considerations must be taken into account to generate a proper language. We understand that the way we treat lexicals might be cumbersome since the complex structure's mapping must 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 essential because it is the user interface to the language. Nevertheless, implementing a good editor is cumbersome. As shown in Sect. 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-



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



```
for
for                  for ( Expression ; Expression ; Expression ) Statement
for  for ( var VariableDeclarationNoIn ; Expression ; Expression ) Statement
for                              for ( Expression in Expression ) Statement
for                                     for ( var in Expression ) Statement
```

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

printing approaches) to try to identify production rule patterns generically. 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 grammar. Besides, 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. Also, languages' coding style and user feedback are fundamental to improve the quality of generated editors. In other words, we need more information to implement the heuristics in a less rigid fashion and therefore improve the editor generation.

6 Limitations

This section discusses the 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. Besides, there is also a technical limitation related to how the mapping is implemented.

Summary: Grammar Preconditions

- Nonterminal symbol name and production rule 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 nonterminal 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 nonterminal symbol is used to define an interface concept in the generated MPS language, and the production labels are used to create concepts. One way to avoid this constraint could be defining a renaming scheme that can detect and fix name conflicts. However, this solution might introduce a side effect on the language's usability; projectional editors use these names for IDE services such as tab completion, so they must be descriptive enough for end users. Also, other language components

must be refactored according to the renaming mechanism. Therefore, 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 concepts *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 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 must 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 nondescriptive 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 trees' definition; 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.

The current approach does not support language nor program evolution. In other words, the current approach considers languages as stand-alone 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 to regenerate the whole language from scratch or make changes by hand. Some changes do not break the importing of programs:

- Addition of language constructs to the grammar and then using them in a program. This means that the plug-in for importing programs, *ImportProgram* (Sect. 4.5), will not find such elements. As a result, the plug-in 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 to supporting languages and programs' evolution.

7 Related Work

Projectional LWBs allow users to manipulate the programs' AST directly; therefore, parsing technology is no longer needed. In contrast, textual LWB parsing is essential. This section presents the state of the art in grammar to model transformation and editor generation.

7.1 Grammar to Model

The generation of models from grammars is essential for the current approach. Thus, we identified the following related work in this direction.

Ingrid [17] is a project that attempts to bridge the gap between textual and projectional LWBs. Their approach uses ANTLRv4 [18] as textual LWB and JetBrains MPS as projectional editor. Ingrid is implemented as a hybrid solution in Java/MPS project. Ingrid bridges textual and projectional LWBs in three steps: firstly, 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 some high-level similarities between Ingrid and Rascal2MPS. Both projects perform the steps taken for parsing, gathering information about the language, generating an intermediate structure to represent the language, and finally generating a model from the said intermediate structure. The main differences are in the architecture, design, and implementation choices of both projects, which have various consequences for using the respective tools. The main architectural difference is in the choice of the intermediate structure. Whereas we chose an external file-based format (see Sect. 4.5), Ingrid uses an internal representation of linked Java objects. This decision enables them to use the ANTLRv4 parser implemented in Java and the ability of MPS to call into Java executables directly. Thus, the Ingrid MPS plug-in can call the parser and start the data extraction process internally. In contrast, Rascal2MPS keeps both LWBs separate; they can communicate only through an external intermediate format. Some of the advantages of not using an intermediate format are:

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

However, this approach has a significant downside: the projectional LWB must call the grammar parser directly. Thus, there is a strong coupling between the projectional LWB and the specific grammar parser. In the case of Ingrid, the MPS plug-in calls into the Java ANTLR parser. However, the ANTLR parser is not the only one. If we wished to extend Ingrid to support Rascal, we would need to replace ANTLR parser calls with Rascal parser calls. This can lead to several problems: (i) The architecture must allow this replacement. This can be partially solved using interfaces and abstractions over the parser, but the problem of potentially different APIs remains. A complete mapping from ANTLR parser function calls to Rascal parser function calls would have to be made in the worst case. (ii) The parser needs to be implemented in Java. ANTLRv4 already has a Java-based parser and is a prime candidate for integration with the 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 implement the parser in Java or find some way to expose the parsing features to a Java environment.

Rascal2MPS addresses the problem of bridging the gap between the textual and projectional worlds in a generic fashion. In other words, neither side of the solution is aware of the other; they communicate only through the intermediate file-based format. This format serves as a contract between the different parts of the solution. If the intermediary file is generated from an ANTLR-, Rascal- or Xtext-based grammar is irrelevant to the implementation on the side of the projectional LWB.

Another difference between Ingrid and Rascal2MPS lies in the editor generation. 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 node's structural elements, i.e., the literals and references to other nodes. It is then left up to the language engineer to apply whitespace to the editor manually. Rascal2MPS goes further and applies heuristics to apply whitespace during the import process automatically. While this does not eliminate the need to edit the editor definitions manually (Sect. 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 are written using the said language in the textual LWB that needs 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. [19] describe a generic semiautomatic approach for bridging the technological space between the extended Backus-Naur form (EBNF), a popular grammar formalism, and Meta-Object Facility (MOF), a standard for model-driven engineering. In this approach, an attribute grammar describes the EBNF structure and the mapping between EBNF and MOF. Then, it is used to generate a Grammar Parser (GP). This GP can then be used to generate MOF meta-models from grammars. However, this approach fixates on MOF as the target meta-model directly. In the case of going between LWBs in separate worlds, we do not want to be specific in the target. Instead, Rascal2MPS uses an intermediate format and makes the source and target formalism up to the implementation. Another downside of the given approach is that it requires grammar annotations and additional manual improvements of the generated model to refine the generated model. We seek to limit the actions of the language engineer, especially concerning the source grammar. The Gra2Mol [20] 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 that can be applied to a program that conforms to a grammar and generates a model that 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 give the meta-model of the target LWB 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 node IDs and layout structures. Generating these from outside of MPS would be quite tedious and would introduce a dependency on the MPS model format, which may change. Thus, it is best to interact with the MPS model from within MPS itself, where MPS can do the heavy lifting of generating the models.

7.2 Editor Generation

Editor generation is an essential step in bridging the gap between textual and projectional LWBs. It is closely related to the well-known pretty printing problem in the grammar world. Grammar cells [21] is an extension of MPS that offers a declarative specification for defining textual notations and interactions in a projectional editor. Implementing editors with this extension makes it easier to offer a text-like editing experience; thus, it is widely adopted by the MPS community. Our current implementation does not use grammar cells because we restricted our approach on a plain MPS installation. However, this extension's adoption is part of the roadmap for the next iteration of the current implementation.

Van de Vanter et al. [22] identify 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 resemble the text-based editing experience closely.

As introduced by van den Brand et al. [23], the BOX language for formatting text 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 generic programming languages, such as GPP (Generic Pretty Printer) [24], which constructs tree structures of a language element's layout that can be used 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 for such a 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. We attempt to limit the required user interaction with the source grammar in our approach, although we did not eliminate it.

Following this research line, 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 each programmer's style, and it changes across languages. For example, in Sect. 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 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. We defined a mapping between textual grammars and projectional meta models; this mapping (Sect. 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 (Sect. 4.4). To validate our solution, we used as a case study a Rascal grammar of JavaScript (Sect. 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 Sect. 6, we discussed some of the limitations of the current approach.

Language evolution is a crucial aspect to look at in the future. Since the current approach assumes that the generation is done only once, we ignore the fact that the textual language and the projectional generated version might change. Then we consider that keeping track of these changes and transferring/applying these changes to the other is essential. If there are changes in the grammar after the projectional language generation, developers must regenerate the whole language, which 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 investigating as future work 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 both sides' changes, 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 Sect. 5.4, the usability of generated editors is one of the critical 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 exploring 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. Erdweg, S., van der Storm, T., Volter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015)
2. Lämmel, R.: The Notion of a Software Language, pp. 1–49. Springer International Publishing, Cham (2018)
3. Fowler, M.: Language workbenches: The killer-app for domain specific languages? (2015)
4. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. (CSUR) **37**(4), 316–344 (2005)
5. Mengerink, J.G.M., van der Sanden, B., Cappers, B.C.M., Serebrenik, A., Schiffelers, R.R.H., van den Brand, M.G.J.: 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 - Volume 1: MODELSWARD, pp. 446–453. INSTICC, SciTePress (2018)
6. Bartels, J.: Bridging the worlds of textual and projectional language workbenches. Master's thesis, Eindhoven University of Technology, 1 2020
7. Mooij, A.J., Hooman, J., Albers, R.: Gaining industrial confidence for the introduction of domain-specific languages. In: 2013 IEEE 37th Annual Computer Software and Applications Conference Workshops, pp. 662–667 (2013)
8. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Not. **35**(6), 26–36 (2000)
9. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131–183 (1992)
10. Nagy, I., Cleophas, L., van den Brand, M., Engelen, L., Raulea, L., Xavier Lobo Mithun, E.: 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, pp. 318–327. IEEE Computer Society, USA (2012)
11. Verriet, J., Liang, H.L., Hamberg, R., van Wijngaarden, B.: Model-driven development of logistic systems using domain-specific tooling. In: Aiguier, M., Caseau, Y., Krob, D., Rauzy, A. (eds.), Complex Systems Design & Management, pp. 165–176. Springer, Berlin, Heidelberg (2013)
12. Gabbrielli, M., Martini, S.: How to Describe a Programming Language, pp. 27–55. Springer London, London (2010)
13. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. Technical report, Jetbrains, 2004
14. Campagne, F., Campagne, F.: The MPS Language Workbench, Vol. 1, 1st edn. CreateSpace Independent Publishing Platform, North Charleston, SC, USA (2014)
15. CWI-SWAT.: Syntax definition (2020)
16. Donzeau-Gouge, V., Huet, G., Lang, B., Kahn, G.: Programming environments based on structured editors: the mentor experience. Interact Program Environ (1984)
17. Vysokỳ, P., Parízek, P., Pech, V.: Ingrid: Creating languages in MPS from ANTLR grammars (2018)
18. ANTLR.: https://www.antlr.org/
19. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: International Conference on Model Driven Engineering Languages and Systems, pp. 159–168. Springer (2005)
20. Luis Cánovas Izquierdo, J., Cuadrado, J.S., Molina, J.G.: Gra2mol: A domain specific transformation language for bridging grammarware to modelware in software modernization. In: Workshop on Model-Driven Software Evolution, pp. 1–8 (2008)
21. Voelter, M., Szabó, T., Lisson, S., Kolb, B., Erdweg, S., Berger, T.: Efficient development of consistent projectional editors using grammar cells. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, pp. 28–40. Association for Computing Machinery, New York, NY, USA (2016)
22. Van de Vanter, M.L., Boshernitsan, M., Avenue, S.A.: Displaying and editing source code in software engineering environments (2000)
23. van den Brand, M., Visser, E.: Generation of formatters for context-free languages. ACM Trans. Software Eng. Methodol. (TOSEM) **5**(1), 1–41 (1996)
24. De Jonge, M.: Pretty-printing for software reengineering. In: International Conference on Software Maintenance, 2002. Proceedings, pp. 550–559. IEEE (2002)
25. Rubin, L.F.: Syntax-directed pretty printing—a first step towards a syntax-directed editor. IEEE Trans. Software Eng. (2), 119–127 (1983)
26. Parr, T., Vinju, J.: Towards a universal code formatter through machine learning. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, pp. 137–151. Association for Computing Machinery, New York, NY, USA (2016)