Department of Distributed and Dependable Systems
Technical report no. D3S-TR-2018-01
May 7, 2018

Department of Distributed and Dependable Systems D3S

INGRID: Creating Languages in MPS from ANTLR Grammars

Přemysl Vysoký, Pavel Parízek, Václav Pech

Abstract: JetBrains MPS is a language workbench, an IDE that allows developers both to write code and create language definitions. It leverages the concept of projectional editing, where the developer directly manipulates the AST representation of program code. While MPS focuses on domain-specific languages (DSL), it needs to support also general-purpose programming languages (GPPLs) — for example, to compile and run programs written in some DSL.

We present INGRID, a method for construction of a language definition in MPS based on its ANTLR grammar. The structure of a language is generated automatically, but its projectional editor and other aspects has to be adjusted manually. During the development of INGRID, we encountered several challenges related to the mapping between grammars and language definitions in MPS that are based on object-oriented principles. Another difficulty is that grammars do not hold any information about code layout. We implemented INGRID as a plugin for MPS and evaluated it on several mainstream GPPLs, such as JavaScript and C#. Results show that our approach is practical, and it saves the user from many hours of tedious and error-prone work. Necessary manual adjustments take only a short time.

This work was partially supported by JetBrains.

1 Introduction

The prevailing approach to define valid syntaxes for programming languages is through grammars, which are typically written in notations based on the Extended Backus-Naur Form (EBNF). Many tools exist for automated generation of parsers from the grammar definitions — for example, ANTLR [5, 13] and Bison/Flex [3]. The source code of programs is written in a text editor, usually embedded into an IDE. Parsers are used to create an abstract syntax tree (AST) representation of the code, e.g. inside the compiler.

An alternative approach is projectional (or structural) editing [8, 7, 2], where a developer directly manipulates the AST representation of the source code instead of plain text. This idea emerged as early as in 1970s, but it failed to get adopted widely, mostly due to inconvenient and unnatural way of manipulating code. A recent revival of projectional editing has been observed in the area of language workbenches — IDE-like tools that enable the developers to manipulate the actual language definition. Popular examples are JetBrains MPS [12, 1] and Spoofax [4]. In particular, JetBrains MPS (Meta Programming System) is an open-source language workbench that focuses on domain specific languages (DSL) and leverages the concept of projectional editing. MPS provides the whole IDE infrastructure that enables developers to design custom languages, use them to write program code, and compile the code into executables. In the rest of the paper, we will often use examples from MPS to illustrate the key concepts of our approach and related issues.

Projectional editing, keeping the code in the AST form, and the absence of parsers, brings along several benefits.

- A projectional editor does not allow the developers to enter syntactically invalid code, because it controls the interaction between the user and the program code.
- Programming languages can be defined in a modular way, and multiple languages can be easily combined together in a single program or one can extend another.
- The languages may support diverse contextual or non-parseable notations with complex layout, such as tables, diagrams, and mathematical expressions.
- Since the projection is detached from the physical representation of code (AST), authors of languages can define multiple notations and allow the developers to switch between them on the screen.

All of this is useful especially for DSLs, which are frequently used by domain experts who may not be professional software developers. The mbeddr project [6] illustrates the abilities of language workbenches in general and of JetBrains MPS in particular. Voelter et al. [7] discuss the benefits and limitations of projectional editors in more detail.

Nevertheless, the usage of projectional editors introduces some new problems. Before a language can be used inside an editor, it has to be defined through the specific infrastructure, so that the IDE can understand the language and work with it. More precisely, an author of a language has to create:

1. the abstract syntax (structure of the language), which defines the types of allowed AST nodes,
2. the concrete syntax for editing, i.e. projection of the AST on the screen and interactions with the user, and
3. text generation scripts to enable creation of plain text representation used as input for compilers.

IDE tools based on projectional editors are typically used for syntactically rich DSLs. In the specific case of MPS, before a program written in a DSL can be executed, it is transformed on the AST level by a series of model-to-model transformations to an AST model that represents the desired semantics in a general-purpose programming language (GPPL), such as C or Java. This model is then converted to a textual representation of that language and compiled by the standard means of the target platform. Therefore, the target GPPL must also be defined in MPS, using the respective infrastructure, in order to allow the DSLs to have their models transformed into a model in the GPPL.

However, very few mainstream GPPLs are now supported by MPS, because it requires substantial effort to manually create a full definition of a language in MPS. Only Java and C have been implemented to date. The overall goal of our project is to automatize the process of language definition in MPS as much as possible. We believe that such an automated process would encourage and speed-up the migration of more GPPLs into MPS, thus giving the authors of DSLs more options regarding target

general languages. Another possible application is the combination of general-purpose languages with DSLs natively created in MPS.

In this paper we present INGRID, a method and a tool for semi-automated construction of a language definition in MPS that uses an ANTLR grammar as a description of its syntax. The process of construction is semi-automated for two reasons: (1) ANTLR grammars have to be adjusted before INGRID can process them, and (2) some aspects of the language definitions in MPS typically must be tweaked or created manually in order to improve languages' usability.

We also discuss the main challenges that we encountered during the development of INGRID and our solution to them. They are related to principal differences between the two approaches to the definition of a programming language — (i) one that uses grammars in an EBNF-like notation, with rules and tokens written in plain text, and parser generators, and (ii) the structured object-oriented approach used in MPS and other language workbenches. Specifically, the description of a language in the form of a grammar does not hold any information about the code layout, and the grammar typically contains many rules that do not directly correspond to AST nodes and programming language constructs.

INGRID can also make the construction of DSLs more efficient. Based on our experience, we believe that, especially for simple languages, it is less time consuming to write their grammar by hand and then create the language definition in MPS with the help of INGRID, than to create everything manually using the MPS GUI.

Although we define the INGRID method in the context of MPS, and illustrate everything using examples from MPS and ANTLR grammars, it can be easily adapted to other language workbenches (i) that use projectional editing and (ii) where the language definition is based on a structural object-oriented approach. Most of the challenges that we discuss and key principles behind our method are general and therefore apply also to other language workbenches.

We make the following key contributions:

- The INGRID method for construction of MPS language definitions from ANTLR grammars, which can be used (1) for the import of general-purpose languages into MPS and (2) for efficient construction of DSLs. INGRID can be adapted also to other language workbenches based on similar principles as MPS.
- Identification and discussion of general challenges related to the differences between (1) language definitions that use grammars and (2) the structured object-oriented approach used in projectional editors.
- Discussion of the practical usability of generated MPS languages and its dependence on (1) the specific form of ANTLR grammars and (2) the manual adjustments of input grammars and specific aspects of MPS languages (e.g., with respect to code layout). Details are provided in Section 3.4 and Section 3.6.

The rest of the paper has the following structure. In the next section, we provide (i) all information about MPS that is necessary to understand the examples presented in the paper and (ii) a brief overview of related work, including projects with similar goals as INGRID. Then, in Section 3, we describe in detail our approach to the construction of a language definition in MPS from its ANTLR grammar, discussing the major challenges together with our solution along the way. We also discuss our experience with application of INGRID on complex mainstream languages, such as JavaScript and C# (Section 4), and then we conclude.

2 Background and Related Work

In this section, we introduce basic features of the MPS platform, and then we discuss other existing projects that aim to add support for general purpose languages into MPS.

2.1 JetBrains MPS

JetBrains MPS is a language workbench — an IDE that allows developers to create their own languages and use them to write code. The code can then be transformed into a target language, typically a GPPL such as Java or C, and eventually compiled into executable programs.

When using the MPS projectional editor, the developer does not work with the textual representation of the source code, but rather directly with its AST that is the model of the code. Programs are created

by assembling the tree (AST) out of predefined building blocks of selected languages. The definition of a language in MPS dictates where in the AST certain elements can be placed and how they can be nested inside other elements.

The building blocks of MPS models are called nodes. Code of any program in MPS is built from nodes, which represent instances of concepts from the languages that the program is written in. In MPS, a *concept* is a language element, i.e. a building block of a language definition. We use the terms *MPS concept* and *AST node* when needed to avoid confusion.

One of the key advantages of projectional editing stems from the separation of abstract and concrete syntax. While AST provides a complete and precise representation of the code, the way it is displayed on the screen and the way the user interacts with it are unconstrained. The editor can take any visual form and shape. The language author can define multiple alternative visualizations and let the developer choose one that fits best the task at hand. In particular, the visual representations are not bound to be just textual at all.

The definition of an MPS concept (language element) consists of several aspects, where each aspect codifies a different part of the AST nodes' behavior. The essential aspects are the following: *Structure*, *Editor* and *TextGen*. Structure represents the abstract syntax (types and hierarchy of AST nodes), Editor defines the concrete syntax (i.e., how the code is visualized and edited) and TextGen specifies how AST nodes are transformed into textual representation. If, instead of generating text directly, programs in the language are supposed to be transformed into another language that is available in MPS, the Generator aspect must be used to specify the model-to-model transformation rules. Since only the Structure, Editor and TextGen aspects are relevant for the contribution of this paper, we describe them below in more detail, and neglect other aspects such as type constraints.

Languages are built from the concepts using techniques known from object-oriented programming — containment, inheritance, interfaces, and so on. Therefore, a definition of a whole language in MPS typically has an object-oriented and hierarchical nature.

Structure. The fundamental aspect of any MPS language is Structure. It must be created first for each intended language concept. Structure specifies core attributes of an MPS concept such as the name, inheritance relationships, child concepts (their types and cardinalities), implemented interfaces, and references to other AST nodes. Figure 1 shows the Structure aspect for the if-then-else statement.

Structure also restricts the type of AST nodes that can appear at a particular place in the tree. For example, one can restrict the condition in if-then-else to be a boolean expression, and the then-block to be a list of statements.

```
concept IfStatement extends Statement
        implements IContainsStatementList
                   IDontSubstituteByDefault
                   IConditional

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

  properties:
  forceOneLine   : boolean
  forceMultiLine : boolean

  children:
  condition        : Expression[1]
  ifFalseStatement : Statement[0..1]
  ifTrue           : StatementList[1]
  elsifClauses     : ElsifClause[0..n]

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

Figure 1: *Structure aspect of the* if-then-else *statement*

Editor. The Editor aspect is where the language designer specifies what the projectional representation of a code fragment (an AST) looks like on the screen and how the user interacts with the code.

JetBrains have developed a cellular system that enables placing properties and children of a node (concept) into different cells. The author usually incorporates all of the node's children, references, and properties inside the representation, so that future users of the language can insert all values that the node expects. Additionally, cells of the editor can be styled using a language similar to CSS. Supported visual characteristics include color and indentation.

Figure 2 shows an example of what the definition of the Editor aspect for the if-then-else statement might look like. While here we indicate positions of cell borders on each line by spaces (just for illustration), MPS GUI actually uses a graphically much more appealing way of displaying the Editor aspects, which involves vertical and horizontal lines of different colors and also background colors other than white for some cells. The symbols [- and -] represent layout information, which specifies mutual positioning of the contained cells (vertical, horizontal, indentation).

```
<default> editor for concept IfStatement
  node cell layout:
    [-
      if ( % conditions % ) [-
      {
      [- % ifTrue % -]
      }
    -]
    ?[- else % ifFalseStatement % -]
    -]
```

Figure 2: *Editor aspect for the* if-else *statement*

BaseLanguage. Another important feature of MPS that we need to describe is *BaseLanguage* [19], a clone of Java implemented using the MPS constructs. BaseLanguage was developed in the early days of MPS in order to implement MPS itself and also to support the basic set of language-definition DSLs, Although BaseLanguage is syntactically almost identical to Java, it is edited in a projectional editor, just like all the languages in MPS. The language-definition DSLs, used to define custom languages by their authors, are generated into the BaseLanguage. Similarly, all the custom DSLs that are meant to be generated into Java choose BaseLanguage as their generation target. The conversion to textual Java sources is handled by BaseLanguage (without any further manual effort), through its TextGen aspect.

TextGen. The TextGen aspect specifies how a given AST node will be translated into plain text representation. It is typically needed only for the bottom-line base languages. DSLs, on the other hand, need to define rules for model-to-model conversions (Generators), since programs in such languages are rarely converted to text directly. After TextGen has generated textual sources from an AST, a compiler for the particular GPPL is invoked to compile the textual sources into binary code. The TextGen definition follows a very straightforward pattern — each node outputs its textual representation into a buffer, while calling TextGen of its children nodes at the right moments. MPS calls the corresponding method on the root AST nodes of the given program.

```
text gen component for concept IfStatement {
  (context, buffer, node)->void {
    append \n;
    indent buffer;
    append {if (} ${node.condition} {) {};
    with indent {
      append ${node.ifTrue};
    }
    append \n {}} $list{node.elsifClauses};
    if (node.ifFalseStatement.isNotNull) {
      append { else} ${node.isFalseStatement};
    }
  }
}
```

Figure 3: *TextGen aspect definition for the* if-else *statement*

TextGen aspect for each AST node (concept of the language) has to be defined using the BaseLanguage. Again, we include an example for the if-else statement (Figure 3).

2.2 Related Projects with Similar Goals

We are aware just of a few attempts to create ports of general-purpose languages into IDE tools based on projectional editing. All of them (described below) were conducted manually.

The BaseLanguage [19] is an almost full port of Java, which was created by JetBrains and extended with MPS-specific features. The C language has also been manually tailored for MPS within the mbeddr project [6].

The following three projects — PE4MPS, ANTLR_MPS, and mps-metabnf — aim to provide certain support for GPPLs within the MPS platform. We describe their main features and limitations in this section.

PE4MPS [17]. This is a project that addresses the lack of information about code layout in grammars by using so called PE grammars [18], which is a new notation proposed by the same author. PE stands for projectional editing.

PE grammars extend the syntax of the ANTLR v4 notation by constructs that provide information about the possible layout of AST nodes. The current version, as of November 2016, supports just horizontal lists, vertical lists, and some indentation rules. However, even these few features make the extended syntax of ANTLR v4 much more complicated.

The PE4MPS tool creates an MPS language in a single atomic step. An input PE file is loaded and processed by the PE parser, and then all concepts (AST nodes) and their aspects are directly generated inside MPS. Information about the code layout, extracted from the PE file, is used when generating the projectional editor for every AST node.

The main disadvantage of the PE4MPS approach is that it only shifts the tedious manual work from creating projectional editors in the MPS IDE to writing PE grammars in plain text. Language developers are forced to specify the code layout manually in PE grammars, through the corresponding extensions. Usage of a standard text editor is inferior (and much more error-prone) with respect to MPS, which has been designed for the purpose of specifying the code layout and provides lot of support to developers. Like for INGRID, every grammar might require non-trivial adjustments before it can be processed using the PE4MPS tool.

We discuss application of PE4MPS on JavaScript in Section 4, and compare its results to our INGRID approach.

ANTLR_MPS [14]. This project also uses the ANTLR v4 grammar notation, but otherwise works quite differently from PE4MPS. Its author created an ANTLR v4 MPS language, which captures the syntax of ANTLR v4 notation using MPS concepts. Given the textual grammar of some language as input, the grammar is imported automatically into MPS, taking the form of the ANTLR v4 MPS language. The next step would be to create a new MPS language from the grammar definition in MPS, but it is not implemented yet.

However, the project has also other important limitations. The tool does not generate any parent-child relationships in the Structure aspect, and it provides no support for the projectional editor at all (by completely neglecting the Editor and TextGen aspects).

mps-metabnf [15]. This tool, implemented by members of the DSLFoundry group, builds upon similar ideas as the ANTLR_MPS project described above. Authors of this project, too, created an MPS language to capture ANTLR grammars. Every imported grammar is represented using this ANTLR v4 MPS language. A user is then able to adjust the grammar as he wishes, specifying the code layout in the projectional editor. In the last step, the target MPS language is derived from the adjusted grammar definition.

The main limitation of this approach is the need to perform the last step manually, because MPS currently does not provide any support for automated transformation of a grammar captured by the ANTLR v4 MPS language into the definition of the actual target MPS language.

Other IDEs. Note that while the INGRID method has been designed primarily for MPS, and can be easily adapted to other IDE tools based on projectional editing, there also exist several language workbenches that use parser generators (such as ANTLR) and support free editing of textual source code. A prominent member of the latter category of tools is Spoofax [4], which essentially restricts the syntax of DSLs to linear text. INGRID is not applicable for creating languages inside such language workbenches.

3 Creating Language Definition in MPS

The proposed INGRID method accepts grammars in the ANTLR v4 notation [5] as input. Syntax definition in the form of an ANTLR v4 grammar exists for all widely used programming languages [1]. Unlike some of the related projects, we did not extend the notation with any custom features, and we also did not create an MPS language for the ANTLR notation.

The process of language construction by INGRID consists of four phases — the first is parsing of the input grammar, followed by definition of the essential aspects of the language (Structure, Editor, and TextGen, respectively).

INGRID is currently able to create (i) a full Structure aspect for each element of the given language, (ii) a very basic Editor, and (iii) a basic TextGen aspect. Therefore, the resulting MPS language typically still has to be adjusted manually to improve its usability, and the remaining aspects not yet supported by INGRID must also be defined.

Our approach differs from the existing projects with similar goals (Section 2.2) especially in the level of automation, and it also has much better support for Editor and TextGen.

3.1 Running Example

We will describe the INGRID method using a simplified XML language defined in Figure 4. The *SimpleXML* language is small but complex enough to be used for illustration of the main challenges and behavior of the proposed algorithms.

```
grammar SimpleXML ;
document  : prolog? comment? element ;
prolog    : '<?xml ' attr* '?>' ;
comment   : '<!--' TEXT '-->' ;
element   : '<' Name attr* '>' content* '</' Name '>'
          |  '<' Name attr* '/>' ;
attr      : Name '="' TEXT '"' ;
content   : TEXT | element | comment | CDATA ;
Name      : NameStCh NameChar* ;
DIGIT     : [0-9] ;
NameChar  : NameStCh | '-' | '_' | '.' | DIGIT ;
NameStCh  : [:a-zA-Z] ;
TEXT      : ~[<"]* ;
CDATA     : '<![CDATA[' .*? ']]>' ;
```

Figure 4: *Grammar of the SimpleXML language in the ANTLR v4 notation*

The grammar of SimpleXML contains elements of the kinds that are listed below. Each color in Figure 4 corresponds to one kind in a way indicated by the list item headers.

- **ANTLR v4 keywords** are required by the notation.
- **Parser rules** describe the structure of a language. Alternatives on the right side of a rule are separated by the pipe character (|).
- **Lexer rules** describe terminal symbols that the parser matches against the input. A terminal symbol can be encoded as a string value or a regular expression.
- **String literals**, always written inside of a pair of single quote marks, also represent terminal symbols, but by exact match to a string constant.
- **String tokens** are described by regular expressions with a special ANTLR v4 regex notation[2].

[1] https://github.com/antlr/grammars-v4
[2] https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md

Elements on the right side of a rule can be annotated with standard EBNF operators (?, +, *) that specify the allowed number of occurrences.

3.2 Phase 1: Parsing Input Grammar

The first phase in the process of MPS language construction is parsing of the input ANTLR v4 grammar. It is done using an ANTLR parser that was automatically generated from the grammar of the ANTLR v4 notation itself. Nevertheless, the full parse tree, which comes out of the parser, is quite complex and contains information not relevant for the INGRID algorithm. In order to get a simple representation that is easy to process by the later phases and keeps only information necessary for the construction of MPS languages, several steps of post-processing of the parse tree are performed in this phase. An output is a custom AST-like structure that represents the grammar, especially the hierarchy of parser rules, in a way more suitable for MPS.

The representation of lexer rules (tokens) in the full parse tree has to be simplified too. In the ANTLR notation, lexer rules can be built from alternatives just like the parser rules — see, for example, the lexer rule Name from our SimpleXML language (Figure 4). For each lexer rule, the parser produces a tree that captures its structure. The lexer rules are, in fact, just regular expressions used to recognize tokens in the input string.

Every tree that represents a lexer rule is flattened into the equivalent regular expression by the recursive algorithm in Figure 5. A sequence is created from the elements of each alternative, and then all those sequences are joined by the alternation operator | to form a regular expression. Our implementation of the algorithm detects rules defined in a recursive way (e.g., using the pattern N : 'text' N) and rejects the input grammar if it contains such rules. However, note that we use the flatenning algorithm only for lexer rules, where recursion is quite rare — specifically, the algorithm is not applied to parser rules.

```
Flatten(R):
  T = empty list
  for each alternative A in rule R:
    R = ''
    for each element E of A:
      if E is not yet flattened then Flatten(E)
      R.append(E)
      R.append(E.operator)
    T.add(R)
  build string S from elements of T:
    S = t1 | t2 | ... | tn
  return S
```

Figure 5: *Flattening algorithm*

The output of `Flatten(Name)`, i.e. application of the algorithm to the Name rule, is this regular expression:

```
[:a-zA-Z]([:a-zA-Z]|\-|_|\.|[0-9])*
```

It defines the syntactically valid identifiers of elements and attributes in SimpleXML. Each identifier must begin with a letter, followed by any combination of letters, digits, underscore, dash, or a dot.

3.3 Phase 2: Structure

In the next phase, the complete structure of the MPS language is automatically generated from the AST that represents syntax of the input language. Elements of the Structure aspect are derived from the AST nodes, and then linked appropriately. Therefore, structure of the MPS language is usually similar to the original ANTLR grammar.

When designing the procedure for translating AST nodes (i.e., grammar rules) into the MPS language structure, we faced several challenges. The main challenge, as we already mentioned, is that a grammar typically contains rules that do not directly correspond to programming language constructs.

Such rules exist at the intermediate layers of a syntax hierarchy. Their main purpose is to enable easier understanding and maintenance of the grammar by humans. Nevertheless, presence of the intermediate layers would significantly complicate usage of the given language in MPS, and the layers are actually not necessary for construction of an MPS language. We show examples illustrating this problem, which we call a *layer problem*, later in this section at a point where we describe our solution. As a part of the INGRID method, we have designed an approach to eliminate the unnecessary layers during construction of the Structure aspect — we call it the *shortcut* approach (Section 3.3.1). However, before focusing on the intermediate layers, we describe the basic principles of translation from the AST into the language structure.

In MPS, each concept of the language (i.e., every AST node) is represented by an object that may have a parent, some children, and properties of any data type. Additionally, the object may implement any number of interfaces, and it may also contain references to objects representing other AST nodes. The parent-child relationships between objects that make the Structure aspect are derived from rules of the grammar. For each rule, the object corresponding to the left-hand side is in the parent-child relationships with sets of objects that represent language elements referenced by the right-hand side. If a rule has multiple alternatives, then a distinct object (MPS concept) has to be created in the structure for each alternative. The name of a concept (object) in MPS is composed from (1) the name of the AST node, which is equivalent to the string encoding of the left-hand side of the corresponding rule, and (2) the number indicating the position of the respective alternative on the right-hand side of the rule.

```
concept Element_1 extends BaseConcept
        implements IContent, IElement

  instance can be root: false
  alias: < > </ >
  short description: Element

  properties:
  Name_1 : Name
  Name_2 : Name

  children:
  Attribute_1 : Attribute[0..n]
  Content_2   : IContent[0..n]
```

Figure 6: *Structure aspect of the* Element_1 *concept*

Consider the parser rule element from the grammar in Figure 4. Its first alternative represents the full XML element with content. Figure 6 shows the Structure aspect for the MPS concept (object) named Element_1 that corresponds to the alternative. The object contains two properties, one for each reference to the lexer rule Name. Values of these properties will be restricted using the regular expression that corresponds to the rule. String literals, such as the opening and closing brackets ('<', '/>') in XML, are omitted because they will be defined only in the Editor aspect for this concept.

References to other parser rules are captured by pointers to child objects. The types of child objects, such as IContent in the Element_1 concept, are determined as follows. Consider the content rule from the SimpleXML language. We show just the relevant fragment of the grammar again in Figure 7. An object corresponding to any one of the four alternatives could be the actual value anywhere the content rule is referenced.

```
content : TEXT | element | comment | CDATA ;
```

Figure 7: *Parser rule content*

Our solution is to use interface concepts. For each rule with more than one alternative on the right side, first an interface concept is defined in the MPS language structure, and then one object that implements the given interface is created for each alternative. The resulting fragment of the language structure looks like the one in Figure 8. It contains one interface IContent that is implemented by four object concepts Content_1, . . ., Content_4.

Now we can illustrate the layer problem on the behavior of auto-completion in MPS. Suppose that a user is creating a new document in the SimpleXML language, just inserted a fresh node of the type Ele-

```
IContent : Content_1 | Content_2 | Content_3 | Content_4
```

Figure 8: *MPS interface* IContent *and the types of implementing objects*

ment_1 (see above), and would like to insert another nested XML element inside. The auto-completion mechanism of MPS offers four options that are displayed in the left part of Figure 9. Each option represents one of the MPS concepts that implement the interface IContent, and therefore also one alternative of the content rule.

Figure 9: *Layer problem in auto-completion*

However, in order to correctly insert another nested element, a user has to perform two steps:

1. Insert an object (node) of the type Content_2 inside the Element_1 node. The Content_2 object has only a single child node of the interface type IElement.
2. Then, the user must again trigger auto-complete and insert either an Element_1 node or Element_2 into the Content_2 node. See the right part of Figure 9.

The main difficulty here is that, in the first step, the user has to (i) either correctly guess what option offered by the auto-complete menu to select, or (2) to remember the order of alternatives in the grammar rule.

Similarly, if the user would like to replace a nested Element_1 node with, for example, an XML comment (i.e., the Comment node), then both intermediary layers have to be deleted before she gets back to the original selection among the concepts Content_1, . . ., Content_4.

Intermediary layers have no visual appearance, and therefore it is very difficult for users to see what is actually happening and they may get confused very easily. The layer problem is addressed by the shortcut approach, which we describe in the next subsection.

3.3.1 The Shortcut Approach

The key idea of this approach is to skip all the intermediary layers (nodes) in the syntax tree, and consider just the nodes to be directly offered to the user through the auto-completion menu. Specifically, the content rule from the SimpleXML grammar, given in Figure 10 together with rules that determine the relevant fragment of the syntax hierarchy, expands ultimately into the nodes highlighted using the bold font in Figure 11.

```
content : TEXT | element | comment | CDATA ;
element : '<' Name attr* '>' content* '</' Name '>' | '<' Name attr* '/>' ;
comment : '<!--' TEXT '-->' ;
```

Figure 10: *The parser rule content with other rules that determine the relevant fragment of the syntax hierarchy*

For the input that consists of a particular AST node N and the grammar rule R that expands N, the procedure implementing the shortcut approach systematically traverses the parser tree (AST) built in the earlier phases in order to identify each AST node that (i) may appear in some derivation chain starting by the rule R from the node N and (ii) does not belong to any intermediary layer. We call them *end nodes*. Here we must emphasize that end nodes do not correspond to terminal symbols from grammars, because they may represent AST nodes with children (i.e., syntax elements that can be further expanded using grammar rules).

Figure 12 shows the algorithm that finds all paths to some end node from a given parser rule R. The algorithm is based on recursive traversal of the parser tree. At each level of recursion, it gathers all

Content_1 (TEXT)
Content_2 → **Element_1**
Content_2 → **Element_2**
Content_3 → **Comment**
Content_4 (CDATA)

Figure 11: *Leaf nodes of the parser tree fragment that has the content rule as its root*

```
 1 FindAllPathsToEndNodes(R):
 2   CurPath = empty list of rules and nodes
 3   return FindPaths(R, CurPath)
 4
 5 FindPaths(R, CurPath):
 6  Paths = empty list
 7  for each alternative A in rule R:
 8    NewCurPath = Clone(CurPath)
 9    if A contains only a single element E:
10      NewCurPath.Add(N_E) where N_E is the node representing E
11      P = FindPaths(R_E, NewCurPath) where R_E is the rule that expands E
12      Paths = Merge(Paths, P)
13    else:
14      NewCurPath.Add(R)
15      NewCurPath.Add(N_A) where N_A is the node representing A
16      Paths.Add(NewCurPath)
17  return Paths
```

Figure 12: *Algorithm to find all paths to end nodes for a parser rule*

paths that lead from the current parser rule R to some end node through its alternatives (line 7). Two cases may occur:

- When a particular alternative A of the rule R contains only a single element E that is a reference to another rule (line 9), the alternative A is an intermediary layer that can be hidden from the user of the MPS language. A run of the algorithm continues, at line 11, by recursively processing alternatives of the rule corresponding to E, which lies at the next level of the parser tree.
- Otherwise, an end node of a derivation chain was found and the recursion stops (lines 13-16).

By appending the node corresponding to the current alternative (lines 10 and 15) and to the rule that leads to the alternative (line 14), the algorithm creates a full path that contains the target end node as the last element of the chain.

We use the name *shortcut approach* for this algorithm, because the paths collected by the algorithm provide shortcuts from the given rule to end nodes, by the virtue of hiding all intermediary layers. For example, the result of this algorithm for the content rule (Figure 10) is the list shown in Figure 11.

The primary use case for shortcuts is to generate options for the auto-completion menus, such that only the end nodes are offered. Shortcuts have to be considered when nodes are inserted, and also when they are deleted. In each case, the whole chain including possibly multiple intermediary AST nodes (up to the end node) must be added, respectively deleted. When the user wants to delete some end node from the AST of a program or document written in the MPS language, the effect of an insertion must be fully reversed.

3.4 Phase 3: Editor

Having the complete structure of the new MPS language, the next phase is to define the visual representation of all concepts (AST nodes) in the projectional editor. As we said in Section 2.1, MPS uses a cellular system that enables the language developer to arrange the children and properties of an AST node in a table-like manner. Cells of different types are supported by MPS — for storing property values, references to child nodes, and keywords (string literals), and also cells that influence layout (e.g.,

indentation). The specific goal of this phase is to create the Editor aspect for each language concept, such that all its attributes — name, properties, children — are projected using the respective cell types.

The main problem that we had to address is the absence of information about the code layout and whitespace in ANTLR grammars. Rules forming the grammar only tell what the syntax tree looks like and how the program text is decomposed into AST nodes. Here we present a solution that is only partially automated for reasons explained below.

We observed that the most tedious and error-prone step in the manual definition of the Editor aspect is the creation of cells for all literals (keywords), properties, children, and other fields of a given concept that should appear in its visual representation. This step can be very easily automated.

Our solution that we implemented in the current version of the INGRID method is to create all the cells and place them in a single row. The resulting basic layout is illustrated on the example of the Element_1 concept in the upper left corner of Figure 13. Further adjustments of the layout, such as indentation and line breaks, can be done very efficiently by the user in the MPS IDE. The bottom right part of Figure 13 shows a fully customized layout for the Element_1 concept.

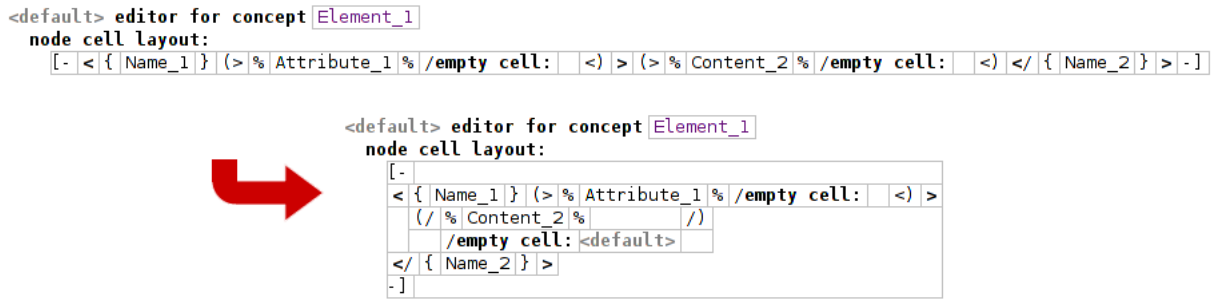

Figure 13: *Editor aspect of the Element_1 concept*

Based on our experience, it takes a very short time to manually adjust the layout into a form much better than any fully automated heuristic could achieve. A typical user of the INGRID method will be able to use the projectional editor in MPS quite efficiently. We actually experimented with several heuristics to derive a useful code layout automatically, but all of them produced rather suboptimal results — it is quite hard to automatically identify language concepts for which the default layout should be refactored, and the user would have to adjust the layout by hand anyway. More details are provided in Section 4, including our experience with several mainstream programming languages.

We have found that the combination of two steps, (1) automated placing of cells into a single row and (2) subsequent manual adjustment of the layout, is a very fast and efficient way of creating a nice code layout. Nevertheless, we plan to work on a more automated approach to the definition of Editor aspects in the future, based on machine learning. The key idea would be to derive the code layout from a set of valid input source files.

3.5 Phase 4: Text Generation

The purpose of the last phase of the MPS language construction is to generate the TextGen aspect for each concept. We use the Element_1 concept again for illustration. Figure 14 shows a fragment of BaseLanguage code that can be used as its TextGen aspect. The code appends all literals, properties, and children of Element_1 to the output buffer.

Like in the case of a projectional editor, the main challenge associated with TextGen is to produce valid code with a reasonable layout. An implementation of the TextGen aspect must determine properly where to put line breaks, spaces, and indentation. For example, in the case of the concept representing an XML element, there must be a space between the element's name and the first attribute, while it is not required between the opening bracket '<' and the name.

Our INGRID method targets mostly text-based languages, for the concepts of which the visual representation in a projectional editor must be almost equivalent to their plain-text representation. An obvious choice would therefore be to use the same approach for Editor and TextGen. Nevertheless, since the Editor aspect has to be adjusted manually, we decided to use a different approach for TextGen. We designed a procedure based on a simple fully automated heuristic that provides acceptable results — the output is human-readable and close to standard formatting of source code used by popular IDE tools, although minor tweaks are often needed.

```
text gen component for concept Element_1 {
  (context, buffer, node)->void {
    append {<};
    append ${node.Name_1};
    append { };
    append $list{node.Attribute_1};
    append {>};
    append $list{node.Content_2};
    append {</};
    append ${node.Name_2};
    append {>};
  }
}
```

Figure 14: *Basic TextGen aspect for the Element_1 concept*

The procedure creates the TextGen aspect for a given language concept in two steps. First, it generates a basic variant that inserts spaces in between every two tokens of the textual representation of the concept. In the second step, spaces are eliminated from places where they are not really needed. The main criterion is whether the generated textual output can be accepted by a parser of the original language (using the ANTLR grammar). We discuss all the cases here:

- When there is a non-alphabetical literal that is used as a token in the grammar, and that might get recognized by the parser without the need for whitespace separators around it, then we can omit the spaces. An example of such literal is '<' in SimpleXML.
- In the case of an arbitrary string, the whitespace may be omitted when the adjacent literal ends, respectively begins, with a non-alphabetical character. This applies especially to the values of properties defined in Structure aspects, such as the name of an XML element. Based on this heuristic, redundant spaces inside of quotes will be eliminated, as well as spaces next to semicolons and around brackets. On the other hand, it will separate language keywords from other literals by preserving the space characters between them.
- Spaces can be safely omitted also when optional child nodes are not present, and in the case of empty lists of child nodes, so that whitespace does not accumulate.

A space must be always inserted when two child nodes are next to each other. Elements from a sequence of child nodes are separated with spaces or line breaks.

Figure 15 contains the complete, automatically generated, TextGen aspect for the Element_1 concept. Such code may be adjusted by hand very easily in order to produce nicely indented XML documents. We only need to wrap the Content_2 child node with indentation and change the sequence separator to a new line character. A fragment of the resulting adjusted code is in Figure 16.

3.6 Remarks about Grammars

During our work on the INGRID method, we have also observed that practical usability of a resulting MPS language depends on the specific manner in which the input ANTLR grammar is defined. We discuss several issues and our workarounds in this section.

Adjusting grammars. In some cases, a small adjustment of the input grammar before the run of INGRID might yield a better and more useful MPS language. Here we illustrate this on two simple examples.

For the first example, we use the definition of an XML attribute in Figure 17, which is taken from the original XML grammar. The lexer rule STRING actually says that quotes make a part of the attribute's value. If the MPS language would faithfully match the grammar, the user would have to always input the leading and trailing quote together with the value in the projectional editor.

In our SimpleXML language, we adjusted the grammar easily in a way that we show in Figure 18. We turned quotes into literals, ensuring that they will only appear in the projectional editor as string constant cells.

```
text gen component for concept Element_1 {
  (context, buffer, node)->void {
    append {<};
    if (node.Name_1.isNotEmpty) {
      append ${node.Name_1};
    }
    if (node.Attribute_1.size > 0) {
      append { };
      append $list{node.Attribute_1 with  };
    }
    append {>};
    if (node.Content_2.size > 0) {
      append $list{node.Content_2 with  };
    }
    append {</};
    if (node.Name_2.isNotEmpty) {
      append ${node.Name_2};
    }
    append {>};
  }
}
```

Figure 15: *Full TextGen aspect for the Element_1 concept*

```
if (node.Content_2.size > 0) {
  append \n;
  indent buffer;
  with indent {
    append $list{node.Content_2 with  };
  }
  append \n;
}
```

Figure 16: *Fragment of the TextGen aspect of Element_1 with adjusted indentation*

The second example relates to a fragment of the JavaScript language, also known as ECMAScript. Every statement in JavaScript needs to be terminated by a semicolon, newline, end of the file, or end of the block — see the Figure 19, which contains the corresponding fragment of the ECMAScript grammar. Since there are multiple options, a user would have to select one of them for each statement in the editor. Technically, every language concept representing a statement would contain one child node of the interface type IEos, which has to be assigned a proper object (i.e., an AST node corresponding to one of the options listed above).

Since MPS can differentiate between statements on the AST level, there is no need for an explicit separator. For example, we can put each statement on a distinct line, as is usual for JavaScript code, and keep just the semicolon as a fixed literal in the projectional editor. The eos rule would have to be changed to this form: eos : ';'. A small adjustment of the grammar, like this one, is a very quick solution and makes the generated language more usable in MPS. On the other hand, this adjustment of the original ANTLR grammar, performed just for the purpose of creating the MPS language, changes the grammar in such a way that it does not precisely describe the standard JavaScript language. To ensure compatibility with JavaScript, the TextGen aspect in the MPS language should put the semicolon after each statement in the plain-text representation of a program code.

Breaking original grammars and parsers. We also want to point out a general problem with grammar adjustments, which all potential users of the INGRID method should be aware of. The original ANTLR grammar for an input language can be very easily changed in a way that, at first, seems harmless and valid inside MPS, but the parser generated out of the adjusted grammar stops accepting the original language. In particular, adjustments needed to improve the resulting MPS language can break down the parser.

This problem has two main causes: (1) low-level implementation details of token matching in the ANTLR parser and (2) usage of ANTLR grammars for a different purpose (that was not expected) in

```
attr : Name '=' STRING ;
STRING : '"' ~["]* '"'
       | '\'' ~[']* '\'' ;
```

Figure 17: *Definition of an XML attribute*

```
attr : Name '="' TEXT1 '"'
       | Name '=\'' TEXT2 '\'' ;
TEXT1 : ~["]* ;
TEXT2 : ~[']* ;
```

Figure 18: *Adjusted fragment of the SimpleXML grammar*

our project. Since, for practical reasons, it is very important that a parser works for programs written according to the original grammar, adjustments have to be performed carefully by the users of INGRID.

4 Evaluation

We implemented the INGRID method as an MPS plugin. While most of the plugin is written in Java, small fragments of BaseLanguage code were needed to bind with the MPS API, which is used to programatically generate language elements and their aspects. The plugin uses the ANTLR library [13] for parsing of ANTLR v4 grammar files, and several MPS libraries that implement the MPS API. Our complete implementation is available at https://github.com/premun/ingrid.

For the purpose of evaluation, we applied INGRID to several well-established and widely used languages, including JSON, JavaScript (ECMAScript 5.1 [10]), and C#. MPS projects that contain definitions of all three languages are also released at https://github.com/premun/ingrid. In the rest of this section, we discuss our experience with application of INGRID to these languages, and then we highlight few general observations.

However, first we must emphasize that MPS languages automatically produced by the INGRID method, are not ready-to-use full-fledged MPS counterparts of the original input languages. The structure of a generated MPS language fully corresponds to the respective ANTLR grammar, but its other aspects have to be manually tweaked (e.g., the Editor) or defined from scratch by the end user. INGRID also does not yet support advanced features of MPS, such as type checking.

For each of the three languages (JSON, JavaScript, C#), we provide (1) its MPS definition in the form that incorporates all tweaks performed manually by authors of this paper and (2) the adjusted ANTLR v4 grammar used as input for INGRID.

JSON. The least amount of manual adjusting after the import into MPS was needed in case of the JSON language [11], because it is the simplest language from all that we used for our experiments. Specifically, the first author spent less than 20 minutes in order to get a language that is ready to use.

JavaScript. In the case of JavaScript, which is an example of a widely-used complex general purpose programming language, automated generating of the language definition in MPS from the ANTLR grammar and subsequent manual adjusting was done in less than one hour.

We are aware of other projects that aim to create a manual port of JavaScript into MPS. For example, there is ECMAScript4MPS [16] developed by the author of the PE4MPS project [17] that we described in Section 2.2.

The main advantage of INGRID over ECMAScript4MPS is that, despite its current limitations, INGRID fully automatically produces the definition of JavaScript in MPS that needs just minor adjustments to be really useful — for example, the language designer has to define actions that improve editing experience by automatically transforming the AST in the background. INGRID achieved a very good result, when compared to PE4MPS, especially in the case of language structure, concept aliases, and support for auto-completion. The Structure aspect generated by INGRID is very similar to that of the ECMAScript4MPS project, which was created manually over a large number of hours.

```
eos : SemiColon | EOF | lineBreakAhead()?
    | _input.LT(1).type() == CloseBrace? ;
// example reference to the eos rule
breakStmt : Break Identifier? eos ;
```

Figure 19: *Statements in JavaScript*

C#. Another very complex programming language that we used for experiments is C# [9]. Before we could run INGRID, we had to manually adjust the ANTLR grammar of C# in order to ensure that INGRID produces at least a reasonable structure (hierarchy of concepts) that can be practically used in MPS just with minor alterations. Necessary adjustments include removal of intermediary rules such as statement_list : statement* ';' by inlining the right-hand side. The Editor aspect was modified afterwards, but only for some of the concepts. Roughly one hour of manual effort was needed in total. Nevertheless, additional modifications of the grammar and selected aspects of the MPS language are still needed. A great flexibility and complexity of the C# language is the main reason behind all the necessary changes to the language definition in MPS and to the ANTLR grammar. The definition is quite large, involving more than 800 concepts.

Other languages. We also tried to apply INGRID on few other languages, such as Python and Ruby. Results are mixed because ANTLR grammars of these languages are written in a style that is not fully compatible with INGRID. For example, the structure and hierarchy of rules in the Python grammar are quite different from JavaScript or JSON, and consequently the language definition created by INGRID is rather badly organized. We plan to address this limitation in the future and extend INGRID such that very few modifications of the grammar would be necessary even for these languages.

General observations. The main overall benefit of the INGRID method is partial automation. Most of the languages discussed above are quite complex regarding their structure and syntax variety. Therefore, completely manual definition would be a very time-consuming and error-prone process. Fully automated generation of the Structure aspect is where INGRID spares the user from many hours of tedious and sometimes quite challenging work.

On the other hand, our experiments with complex languages show that, in the case of the Editor and TextGen aspects, manual adjustment (e.g., adding line breaks and indentation) is a very effective approach that takes only a short time — in particular, an order of magnitude less time than we initially expected. The first author spent between 20 and 60 minutes of work on each language, when adjusting the result of automated generation into a more useful and readable form. MPS IDE provides good support for efficient tweaking of the code layout in Editor and TextGen, allowing designers to produce really useful languages very fast. Despite that, we also tried to design some automated heuristics, but so far all yield rather mediocre results when compared to what human users can achieve efficiently instead.

Regarding future changes to the ANTLR grammar of a subject language, it is certainly possible to directly update the corresponding MPS definition, but in some cases it may be easier to run again the whole import process that will overwrite the previous definition with a new one.

5 Conclusion and Future Work

Our contribution is the INGRID method for constructing language definitions in MPS from grammars in the ANTLR notation. The method is only partially automated, but nevertheless it greatly reduces the amount of manual work that developers of MPS languages have to perform. Although we discussed all challenges, details of our solution and general observations just in the context of ANTLR grammars and JetBrains MPS, we believe they are more general — in particular, relevant to everyone tackling a similar problem in the field of DSLs and language workbenches.

In the future, we would like to add support for generating other aspects of MPS languages and increase the level of automation. Our primary research goal is to investigate the usage of automated (machine) learning for the purpose of generating editors that support nice and readable code layout. First, we would have to collect a set of input source code files to be used for training. This step might involve a user study designed in order to identify examples of layout that a majority considers as nice

and readable. The information about code layout, once it is available, will be then leveraged and used also to improve TextGen. Another subject for future work is automated construction of parser-based tools that could be used for translating source code in plain-text to instances of MPS languages.

References

[1] F. Campagne. The MPS Language Workbench, Volume I. CreateSpace Independent Publishing Platform, 2014.

[2] V. Donzeau-Gouge, G. Huet, B. Lang, and G. Kahn. Programming Environments Based on Structured Editors: The Mentor Experience. Research Report RR-0026, INRIA, 1980.

[3] J. Levine. Flex & Bison: Text Processing Tools. O'Reilly Media, 2009.

[4] L.C.L. Kats and E. Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. OOPSLA 2010, ACM.

[5] T. Parr. The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2013.

[6] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. SPLASH 2012, ACM.

[7] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. SLE 2014, LNCS, 8706.

[8] M. Voelter, J. Warmer, and B. Kolb. Projecting a Modular Future. IEEE Software, 32(5), 2015.

[9] C# Language Specification, Version 5.0 (2012), Microsoft Corporation, https://msdn.microsoft.com/en-us/library/ms228593.aspx

[10] ECMAScript Language Specification, Edition 5.1 (June 2011), http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf

[11] The JSON Data Interchange Format, 1st Edition (October 2013), http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[12] JetBrains MPS, https://www.jetbrains.com/mps/

[13] ANTLR parser generator, http://www.antlr.org

[14] F. Campagne. The ANTRL_MPS project, https://github.com/CampagneLaboratory/ANTLR_MPS

[15] DSLFoundry. The mps-metabnf project, https://github.com/DSLFoundry/mps-metabnf

[16] M. Lombardo. The ECMAScript4MPS project, https://github.com/mar9000/ecmascript4mps

[17] M. Lombardo. The PE4MPS project, https://github.com/mar9000/pe4mps

[18] M. Lombardo. PE grammars, https://github.com/mar9000/pe

[19] MPS BaseLanguage, https://confluence.jetbrains.com/display/MPSD34/Base+Language