# Consistent Projectional Text Editors

Stian M. Guttormsen[1], Andreas Prinz[1] and Terje Gjøsæter[2]
[1]*Department of ICT, University of Agder, Grimstad, Norway*
[2]*Department of Computer Science, Oslo and Akershus University College of Applied Sciences, Oslo, Norway*

Keywords: Language Workbench, Projectional Editor, Grammar, Ambiguity, User Interface.

Abstract: For modelling and domain-specific languages, projectional editors have become popular. These editors implement the MVC pattern and provide a direct connection to the underlying model. In particular, projectional editors allow much more freedom in defining the concrete syntax than traditional grammars. The downside is that it is possible to define presentations that are of bad quality, and that this is not easily visible. In this article, we identify some of the problems with projectional editors and propose ways to resolve them. We also demonstrate a proof-of-concept solution, showing how problematic editor presentations could be identified automatically.

## 1 INTRODUCTION

Editing a program in a textual programming language means editing text and making sure that it follows the syntactical rules of the language. These rules are specified by a formal grammar and form the concrete syntax of the language. The program then gets parsed and transformed into an abstract syntax tree: a representation of the program in memory as a tree structure. This tree structure can then be analyzed or transformed further.

Projectional editors (Völter et al., 2014) avoid the parsing of concrete syntax and instead allow editing the abstract syntax tree directly. This is enabled by the model-view-controller pattern (MVC), where the editor shows a view of the model and changes the model based on user input. This way, the concrete syntax is just a projection of an already-existing abstract syntax tree. With access to the abstract syntax tree, projectional editors can easily analyze programs as they are written (similar to earlier syntax-directed editors (Lunney and Perrott, 1988)). This type of editing is common in modelling and domain-specific languages (DSLs), where the model is an abstract structure that the user edits through a projection (Völter et al., 2013). It is the method of choice when editing graphical UML diagrams.

In projectional editors models are edited in their internal representation and not as text, therefore they are very flexible in terms of how the concrete syntax is defined. Basically, only a pretty printing of the underlying model is defined. This is particularly useful for graphical grammars. It is also true for textual projections because the text is also just a projection, and each text element is directly connected to the model through the projection. The projection can take on many forms and it is even possible to combine different types of projections, such as including tables inside a textual presentation. There is no room for ambiguous or conflicting grammars because there is no need for grammars or parsers at all. Defining the concrete syntax of a language just means defining how the abstract syntax gets projected. However, a projection can still be ambiguous or confusing to the user. Some projections are really bad and should be rejected, which this paper advocates. On a higher level, this problem is discussed in (Karsai et al., 2014). Our paper is a more concrete version of their Guideline 16: “Make elements distinguishable."

Meta Programming System (MPS) is a language workbench that is built around projectional editing (Toporov et al., 2013). It allows textual, tabular and graphical projections. As a language workbench it supports the development of DSLs and the tools needed to use them. The projectional editor in MPS plays a key role in allowing different DSLs to be combined to form a single solution; language composition is one of the benefits that comes easy with projectional editors (Völter, 2011; Völter and Solomatov, 2010).

Guttormsen S., Prinz A. and GjÃÿsÃęter T.
Consistent Projectional Text Editors.
DOI: 10.5220/0006264505150522
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 515-522
ISBN: 978-989-758-210-3

Mixing multiple languages and notations in a single program has traditionally been hard with textual programming languages. It is a difficult task to compose textual syntaxes to form a well-defined grammar. On the other hand, with projectional editing it is natural to combine projections because there is no requirement stating that the syntax should be unambiguous.

Projectional editors offer a completely different editing experience compared to traditional text editors, and this way of editing can be seen as difficult and cumbersome by developers. Research on current limitations of projectional editors (as found in MPS) shows that there are some real issues, and especially (Völter et al., 2014) highlights many of them. However, current research seems to mostly focus on comparing features which are lacking or difficult in projectional editors, but which are default or simple in text editors (such as e.g. support for copy/paste). In this paper we will look at it the other way; there are a lot fewer restrictions on the concrete syntax with projectional editors compared to textual ones and this may cause problems. The freedom in defining the concrete syntax is not problematic for the computer, but it can cause problems for the language user.

Not having to worry about unparseable programs can be seen as a benefit of projectional editing, but we argue it is only beneficial to the language designer and that it can actually be a hinderance to the user. Ambiguous syntax is not only a problem for parsers, but a problem that affects humans as well. With projectional editors there are no sanity checks regarding the concrete syntax. In this paper we present a solution to this problem that we developed in (Guttormsen, 2016), showing how projectional editors can be extended with sanity checks. These checks makes it possible to automatically identify ambiguities in the concrete syntax.

Surveying related work shows that there have been similar experiments on automatic analysis and model-checking. (Völter, 2014) discusses some different methods for analyzing and checking models, but the methods discussed are aimed at analyzing programs—not languages. It differs from our work in that we focus on analyzing the concrete syntax of languages.

So the novel contribution of this paper is its focus on the user when looking at the quality of a concrete syntax. This is opposed to the focus on machine analyzability. The solution we implemented is built around projectional editors, but the ideas are also applicable to non-projectional approaches like Xtext.

We continue by discussing text analysis from a human-machine point of view in Section 2. After that, we look into human-human communication in Section 3, and ambiguity as a problem. We provide a solution in Section 4, before summarizing in Section 5.

## 2 TEXT ANALYSIS AND PROCESSING

In order to get a better understanding of the issues related to projectional editors, we will look into the handling of program texts and syntax in general.

Handling of syntax started with the advent of compilers, which analyse the input text, make sense out of it and generate some output text. As far as the input text is concerned, grammars were found to be the best theory to describe the structure of the input. Several classes of grammars were identified. An example is the Chomsky classification, which was introduced by Noam Chomsky. He divided grammars into four classes: unrestricted, context-sensitive, context-free, and regular grammars.

Finally, for most applications it turned out the context free grammars would do the trick for defining the syntax, balancing expressivity and analyzability. However, even those were not easily analyzable in all cases, and for automatic generation of analysis tools more restricted classes were introduced. LL(1) was identified to fit for top-down parsing, and LR(1) for bottom-up parsing. Finally, analysis techniques became more advanced and LL* tools appeared (e.g ANTLR (Parr and Quong, 1995)).

At the same time, it turned out that the concrete syntax was too detailed for efficient analysis. This led to the focus onto the definition of abstract syntax, in order to capture the essential features of the language. From there, the relation to the concrete syntax was given by a mapping.

To handle graphical languages, meta-modelling was introduced which started with the abstract syntax (called meta-model), and attached the concrete syntax to the abstract syntax. This way, the traditional path from concrete syntax via abstract syntax and abstract code to concrete code was reversed at the beginning. Now, the abstract syntax was the central element. Thus, it was possible to define graphical languages, like UML. As an example we show the graphical syntax for a simple class diagram modelling a *library* in Fig. 1, here defined using EMF (which is essentially a subset of UML class diagrams). The class diagram is just the presentation of the abstract syntax, whose structure is shown in Fig. 2.

Defining a projectional editor involves specifying how each element of the abstract syntax should be presented. The editor definition specifies how the user input should be translated into changes in the abstract

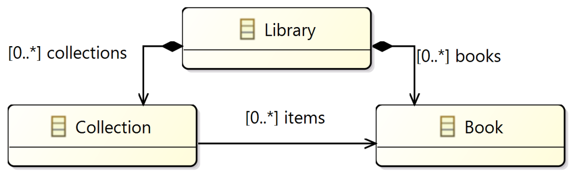

Figure 1: A graphical syntax projection in EMF of a language for libraries. A library can here contain books and collections of books.

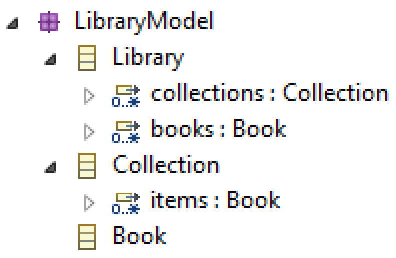

Figure 2: The abstract syntax in EMF of the library language, presented in the tree view. (Bear in mind that the EMF tree view representation is still just a projection of the actual abstract syntax).

model, and also how the model should be viewed by the user. In MPS, for instance, (textual) editor definitions of an element consist of a number of cells. Cells can contain keywords, collections of cells, references to properties of the element, etc. A cell that references a property in the abstract syntax means that user input on this cell is entered as the property value in the model. These references ensure a direct connection between the concrete and abstract syntax, leaving no room for ambiguity in parsing (from a machine's point of view). And so it is up to the language designer to create reasonable editors so there is no doubt about which element is edited.

So far, we have discussed abstract and concrete syntax, but languages do not only have syntax, they also have semantics. It is important that the semantics is unambiguous as well, and sometimes an ambiguity in syntax can be remedied by having the same semantics for both cases. In this article, the semantics is not considered, because it provides a link between the abstract syntax and the semantic domain of the language, hence coming *after* the problem discussed in this article.

## 3 THE HUMAN ASPECT

As described in the previous section, the development of grammar classes was geared by the need to facilitate the interaction between human and machine. The programmer (human) has to be able to formulate the tasks in a way that the computer (machine) is able to understand. Advances in technology have led to the situation that many earlier restrictions on the input languages could be dropped. We are no longer restricted to using LL(1) analysis, we can use LL(*) or even LR(*). We might even consider the advances in natural language processing as having even less restrictions.

However, there is a second side to program texts and that is about communication between humans. Modern development methods enable teams of developers to work together. That implies a far larger amount of reading of programs than before, such that reading and understanding and discussing programs is the main activity. With the advent of (formal) modelling in domain specific languages, this situation is even more serious, since the people communicating are not necessarily from the same background, but can involve administrators, users, and project managers.

For this situation, it is important that programs and specifications can be easily understood and discussed. This means they have to be optimized for easy understanding by humans. A minimum requirement in this context is the unambiguity of the concrete syntax, without a need to rely on the abstract syntax. This requirement is often not met by languages built on the frameworks of projectional editing, since there is no need to enforce it (for the human-computer interaction).

Let us take a graphical language as an example. We use a projection of our language for modelling libraries (see Fig. 3). It is based on the same abstract syntax as shown in Fig. 2. The projection is very similar to the one shown earlier in Fig. 1, but two of the names of the associations have changed places.

If we write code based on this perception, we might want to access the `items` reference in a library:

```
Library lib = new Library();
Collection<Book> items = lib.items;
```

However, this code will lead to a compiler error "attribute not accessible". Why is this the case? The answer is obvious when we consider that the abstract syntax of the diagram is the same as in Fig. 2.

Here, it becomes clear that the concrete syntax is misleading; the `items` and `books` references have

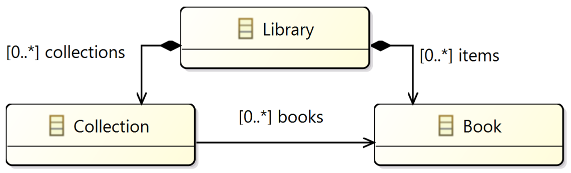

Figure 3: A different graphical syntax projection in EMF of the same language for library modelling. Note that the names on the associations are changed.

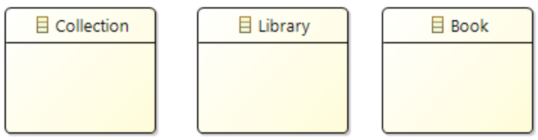

Figure 4: Lay out the classes.

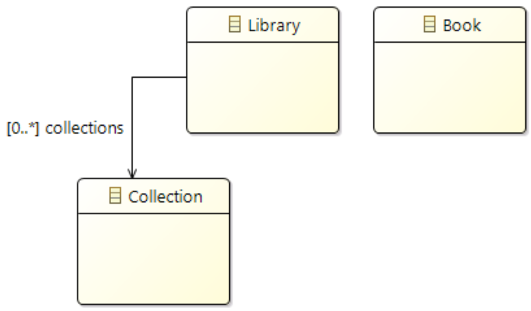

Figure 5: Add reference and adjust.

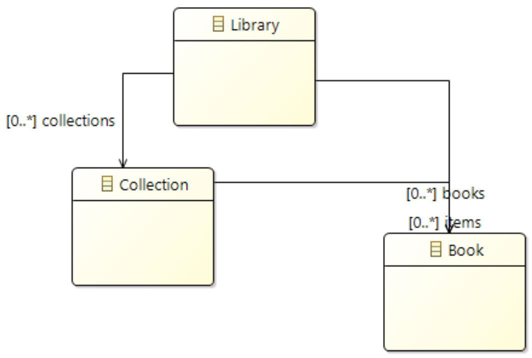

Figure 6: Add more references and adjust.

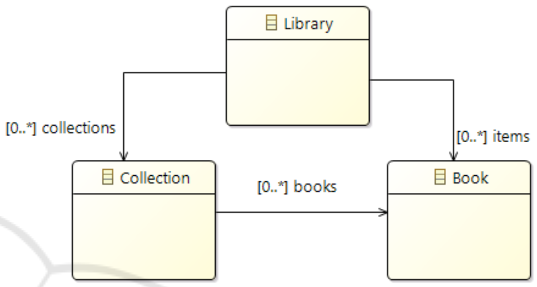

Figure 7: Sort out overlappings.

swapped places in the diagram compared to the abstract syntax, but it is just the projection that is inaccurate—it is still the same abstract syntax underneath. Of course, this is an extreme example, but similar situations can easily appear in large diagrams where the names are close to each other leading to ambiguities.

Can this realistically happen in the real world? Fig. 4 to Fig. 7 show step by step how such a situation can occur while creating a model and moving the labels around between the references to make them more easily readable. In some situations it may be impossible to distinguish which name belongs to which reference, and rearranging them may lead to accidentally placing them wrongly. In Fig. 4, we lay out the classes, in Fig. 5, we add the first reference and successfully and unambiguously lay out the classes and labels for that part. The problems start as we see in Fig. 6, when we add the two references to `Book` and after moving around the labels and classes we reach a situation where the references and labels are ambiguous. In Fig. 7, we try to fix it, but accidentally place the labels wrongly. Some but not all model editors will show a connection between the label and its reference when it is being moved, but confusion can still occur, particularly in more complex models. With complex models it is typical that the standard layout is not sufficient and the graphical elements are moved on the screen. It can very well happen that such a move leads to unintended results as shown in Fig. 3. In a larger project, such errors are even more difficult to spot.

Similar situations can appear in textual projectional editing (in MPS), where the editor does not always behave the way the user expects. The editing in MPS can be made very text-like, and e.g. creating a new Java class in MPS is done in almost exactly the same way and with the same keystrokes as in the Eclipse Java IDE. After creating the file for the new class you are left with a skeletal implementation with no name, and the text cursor is at the position for the class name. There you can enter a name for the class and press enter; the text cursor then moves to the body of the class. A field can be created by e.g. writing "public String name;", i.e. it is identical to Eclipse. Code-completion can also be used in the same fashion as in Eclipse. The projectional editor in MPS imitates textual editors very accurately, but there are things that are different. For instance, pressing enter in a text editor would create a new line at the current cursor position, but in MPS this is not always the case; pressing enter with the cursor at the class name would create a new line inside the class instead of before the brackets surrounding the class (as is normal in Eclipse).

To demonstrate how ambiguities can occur in MPS we created a language in MPS for defining constants and variables. The language is very simple and the syntax for the two concepts are similar to those found in many other popular languages. A constant is specified with a datatype, followed by the name of the constant, then an equals-sign and the value of the constant. The syntax for a variable is identical except that only the datatype and name needs to be specified. The following example shows a program written in the language.

```
int x = 2
int y
```

It describes a constant, x, and a variable y. Opening the inspector MPS allows us to see the nodes in the abstract syntax tree. The above two lines are displayed as follows in the inspector.

```
[constants]
  Constant "x"[6953060390811085594]
    in Example.sandbox
[variables]
  Variable "y"[6953060390811085603]
    in Example.sandbox
```

Inspecting the program reveals that the first part is a Constant and that the second part is a Variable. The editor for this language was defined in MPS as follows.

```
editor for concept Definitions
  node cell layout:
    [-
      (- % constants % /empty cell:  -)
      (- % variables % /empty cell:  -)
    -]
```

This describes a collection of constants followed by a list of variables. The only difference syntactically is that constants always come before variables. The empty cell defines what will be shown if there are no constants or variables, and in this case they are shown as whitespace. The editor for constants is defined as:

```
editor for concept Constant
  node cell layout:
    [- {type} {name} = {value} -]
```

The constant node contains three properties: type, name and value. The equals sign is just part of the projection to make it look more like an assignment. The editor for variables is identical except that the value is set as optional. Furthermore, if the value is empty then the equals sign is hidden as well, indicated by the question mark in the following definition.

```
editor for concept Variable
  node cell layout:
    [- {type} {name} ?= {value} -]
```

The problem with these editor definitions is highlighted in the following example. Similarly to the previous example there are two definitions of what appears to be a constant and a variable:

```
int a = 24
int b
```

However, the MPS inspector shows that there are two variables:

```
[variables]
  Variable "a"[6953060390811111363]
    in Example.sandbox
[variables]
  Variable "b"[6953060390811085603]
    in Example.sandbox
```

So far, we have argued that ambiguous presentations are not a good idea. However, even unambiguous grammars can lead to problems in understanding. The grammar for SDL-2000 (ITU-T, 1999) is given in a projectional fashion, and it is intended to be unambiguous. However, as it stands, the grammar is ambiguous. Moreover, it has plenty of conflicts for typical parser tools. There have been attempts at solving these problems in the scope of other projects (Prinz, 2000), but without success. Instead, several severe problems have been identified and reported to the standardization group. There are several valid SDL specifications that have more than one interpretation when LR(1) is used, even though there is a unique correct solution.

A similar attempt was done using ANTLR (Parr and Quong, 1995; Schmitt, 2003). Also in this attempt, more than 20 inconsistencies are reported in the comments to the grammar. Even with an infinite lookahead, it is not possible to parse all SDL specifications correctly. In some cases analysis is only possible with prioritization of alternatives, which is not faithful to the standard.

The task of disambiguation with a long lookahead might be feasible for tools, and it is of course also possible for humans. In some sense, it is done every day. However, humans use context to aid in disambiguation, and the large amount of language-related jokes proves that this is still difficult and not always successful.

We would therefore consider long lookahead as bad style and advocate LR(1) as best practice. Typically, LR(1) languages suffice for reasonable languages and in particular for most DSLs. Using a long lookahead would not necessarily pose a problem for computers trying to parse the input text, but it can make it difficult for humans to understand. The idea behind limiting ourselves to LR(1) languages is to make it easier for humans to understand—not computers. The problem might be that a grammar for a language is not LR(1), while the language itself is LR(1).

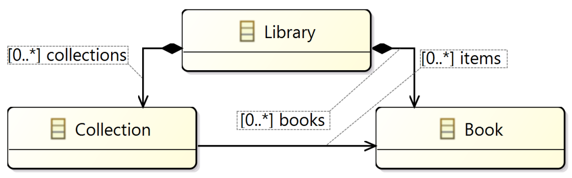

Figure 8: A graphical projection with explicit connections between references and their labels.

## 4 SOLUTION

We will now present our ideas for solving some of the problems related to projectional editors.

### 4.1 Graphical Languages

Graphical specifications are normally created using graphical editors, which allow users to create graphical elements and present these elements in the graphics. So they are all projectional editors and parsing of graphical specifications is not an issue. Creating full analyzability of graphical specification is still a major effort for the future. Even for the handling of projectional editors, there are many open research questions related to best ways to describe the relationship between concrete syntax and abstract syntax.

In this paper, we therefore only solve the simpler problem of hidden items in the presentation as also given in the problem example in Section 3. The graphical syntax presented earlier showed how the concrete syntax is not always an accurate presentation of the abstract syntax. Continuing with our example we propose that all connections between elements should be presented explicitly. Fig. 8 shows how this is done for the example we showed earlier (Fig. 3). The labels are connected to their references with a dashed line (similar to how EMF does when you move the labels around).

Making explicit the implicit connections between elements makes the graphical syntax less ambiguous. However, this is only true if the syntax used to present the implicit connections does not introduce other ambiguities; ambiguous syntax that is explicitly defined is no better than ambiguous syntax that comes from implicit connections.

The SDL graphical syntax is described using graphical grammar constructors. It features the following spatial relationships: *contains*, *is associated with*, *is followed by*, *is connected to*, *is attached to*. In addition, there are graphical base symbols. We will relate to these basic relationships and symbols in order to present rules for unambiguity. The following rules should allow basic unambiguity for such graphical grammars.

- The graphical base symbols should be unambiguous, i.e. no two of them should be equal.
- In order to have a proper presentation of *contains*, it is important that symbols do not overlap otherwise, only when they are contained.
- The two relationships *is followed by* and *is connected to* are given by explicit connection symbols, so they do not pose problems.
- The relationship *is attached to* is exactly the problematic implicit connection, and should be represented by an explicit connector.

### 4.2 Textual Grammar

Introducing explicit syntax does also fix the issues with the textual syntax shown earlier. By e.g. introducing a keyword, `const`, for the constants in the example, there would be no ambiguities between variable- and constant-declarations. Such a solution is often easy to implement, but it is more critical to find out when such a solution is needed in the first place.

We have devised a way to find problems. In order to achieve a check of quality for the projectional syntax, it is first translated into a grammar. The projectional syntax is very similar to a grammar, so the translation is not too difficult. The grammar is then checked according to the LR(1) criteria. The user is warned when something bad is happening, allowing him or her to fix possible ambiguities.

We have developed a proof-of-concept solution for MPS, demonstrating how a projectional editor could be checked for possible inconsistencies. Our solution is implemented in a series of steps, where first the editor definition is mapped into a textual language for grammars. Then the generated grammar is used as input to a parser generator. Finally, any output from the parser generator will be reported back to the user in MPS. In this way, the parser generator is responsible for detecting anything bad in the grammar (and indirectly the projectional editor definition).

For the example, the following translations are used. We start with the original syntax definitions as follows.

```
editor for concept Definitions
  node cell layout:
    [-
      (- % constants % /empty cell:  -)
      (- % variables % /empty cell:  -)
    -]
editor for concept Constant
  node cell layout:
    [- {type} {name} = {value} -]
  editor for concept Variable
    node cell layout:
      [- {type} {name} ?= {value} -]
```

The above editor descriptions are automatically translated into the following grammar rules.

```
  Definitions ::= Constant* Variable*
    NL ;
  Constant    ::= IDENTIFIER IDENTIFIER
    EQUALS INTEGER NL ;
  Variable    ::= IDENTIFIER IDENTIFIER
    (EQUALS INTEGER)? NL ;
```

In this grammar, `IDENTIFIER`, `EQUALS`, `INTEGER`, and `NL` are terminal symbols for identifiers, equals sign, integers, and newline, respectively. As it stands now, it will lead to conflicts in the LR(1) analysis. The problem is that variables and constants are overlapping, which means the grammar is ambiguous; when reading an identifier, it is not clear if the constant part should be finished or not.

Our addition to the editor language in MPS is able to pick up on this type of error. When using our checking tool on the above editor definition, the parser generator reports the following back to us in MPS.

```
Analyzing: definitions.editor.cup
Warning : *** Shift/Reduce conflict found
    in state #3
  between Constant_0_Arb ::= (*)
  and     Constant_0 ::= (*) IDENTIFIER
    IDENTIFIER EQUALS STRING
  under symbol IDENTIFIER
  Resolved in favor of shifting.

Warning : *** Shift/Reduce conflict found
    in state #4
  between Constant_0_Arb ::= (*)
  and     Constant_0 ::= (*) IDENTIFIER
     IDENTIFIER EQUALS STRING
  under symbol IDENTIFIER
  Resolved in favor of shifting.

Error : *** More conflicts encountered
  than expected -- parser
  generation aborted
------- CUP v0.11b 20150930 (SVN rev 66)
  Parser Generation Summary
------- 1 error and 2 warnings
```

The errors reported above also give some indication as to where they come from. `Constant_0` tells us that it comes from the editor for constants, and the zero means that it comes from that editor's first child-node. So the tool cannot decide whether it should read the line "int a = 10" as a `Constant_0` (Shift), or finish reading constants and read the line as variable (Reduce). At the moment, the error feedback to the user still can be improved, but the principle stays the same.

The problem with the grammar given by the conflicts can easily be resolved by e.g. introducing a `const` keyword as mentioned earlier. The grammar would then look like this.

```
  Definitions ::= Constant* Variable*
    NL ;
  Constant    ::= const IDENTIFIER
    IDENTIFIER EQUALS INTEGER NL ;
  Variable    ::= IDENTIFIER IDENTIFIER
    (EQUALS INTEGER)? NL ;
```

When we build the language again, we get a much better output.

```
Analyzing: definitions.editor.cup
------- CUP v0.11b 20150930 (SVN rev 66)
  Parser Generation Summary
------- 0 errors and 0 warnings
```

With the current implementation, MPS can have several false positives, i.e. warnings that are not really problems. However, most of them are easily solved, and when the description does not have problems, then it is good (no false negatives).

## 5 SUMMARY

In the past, the development of different grammar classes was geared by the need to facilicate human-machine interaction. Then along came graphical languages, and with them projectional editors. Projectional editors avoid parsing concrete syntax and instead work directly with the abstract syntax. The concrete syntax is defined as projections of the abstract syntax (like views from the MVC pattern). This allows for much more flexible concrete syntax definitions, but there are also downsides to this. Projectional editors can handle programs that would be otherwise difficult to parse. This is positive because it allows very flexible language definition, but it is negative when trying to explain programs to other humans; syntax ambiguities do not only affect parsers, but humans as well. The problem not only applies to projectional editing, and we also consider textual grammars with long lookahead (LL*) to be of bad style.

Advocating LR(1) as best practice, we outline how to improve the concrete syntax definitions for projectional editing. Avoiding implicit connections between elements, and avoiding same syntax for different elements are some measures that can be taken. Furthermore, deriving LR(1) criteria for graphical grammars (similar to how it is done with SDL-2000) can make these grammars more easily analyzable. Textual syntaxes for projectional editors can be checked similarly using LR(1) criteria. The problem with this approach is that the syntax definitions are often not given as grammars in projectional editors. A possible solution is to map the editor definitions into textual grammars so that these in turn can be analyzed for possible ambiguities. We have created a proof-of-concept for the MPS language workbench using this method. It proves to be a viable method for ambiguity checking of projectional editors.

In the future, we want to extend the prototype towards graphical languages. To help leverage the tools used for analyzing textual grammars also for the graphical case, we propose deriving LR(1) criteria for graphical grammars. The graphical grammar is mapped into an LR(1) grammar, which is then analyzed. This could be achieved similarly to how SDL-2000 (ITU-T, 1999) uses an extended EBNF in order to capture its graphical grammar. This way, spatial relationships are mapped onto normal grammar sequence constructs, which would allow sanity checks for graphics. This approach could be combined with the sanity checks proposed in (Moody, 2009).

Before projectional editors, technology was the limiting factor requiring unambiguous grammars. Now, humans are more limiting, needing unambiguous concrete syntax for communicating with other humans.

## REFERENCES

Guttormsen, S. M. (2016). Changing meta-languages in MPS. Master's Thesis, University of Agder, Grimstad.

ITU-T (1999). SDL - ITU-T Specification and Description Language (SDL-2000). ITU-T Recommendation Z.100, ITU-T.

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., and Völkel, S. (2014). Design guidelines for domain specific languages. *Computing Research Repository*.

Lunney, T. and Perrott, R. (1988). Syntax-directed editing. *Software Engineering Journal*, 3:37–46(9).

Moody, D. (2009). The physics; of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng.*, 35(6):756–779.

Parr, T. J. and Quong, R. W. (1995). ANTLR: A predicated-LL(k) parser generator. In *Software – Practice and Experience, Vol. 25(7)*, pages 789–810. ACM Press New York.

Prinz, A. (2000). *Formal Semantics for RSDL: Definition and Implementation*. PhD thesis, Humboldt-Universität zu Berlin.

Schmitt, M. (2003). Parser for sdl-2000. URL: http://patakino.web.elte.hu/SDL/Parser/SDLParser.g, accessed 2015.

Toporov, E., Pech, V., and Shatalin, A. (2013). *MPS User Guide for Language Designers*. Confluence - JetBrains. https://confluence.jetbrains.com/display/MPSD32/MPS+User's+Guide, accessed 2015-06-05.

Völter, M. (2011). Language and IDE modularization, extension and composition with MPS. *Pre-proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 395–431.

Völter, M. (2014). *Generic tools, specific languages*. PhD thesis, TU Delft, Delft University of Technology.

Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C., Visser, E., and Wachsmuth, G. (2013). DSL engineering: Designing, implementing and using domain-specific languages. *Implementing and Using Domain-Specific Languages. dslbook. org*.

Völter, M., Siegmund, J., Berger, T., and Kolb, B. (2014). Towards user-friendly projectional editors. In Combemale, B., Pearce, D. J., Barais, O., and Vinju, J. J., editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 41–61. Springer International Publishing.

Völter, M. and Solomatov, K. (2010). Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering, SLE*.