An MPS implementation for SimpliC

Raphaël Castanier
D11, Lund University, Sweden
raphael.castanier@grenoble-inp.org

Abstract

The goal of this study was to evaluate JetBrains language editor provided as an Open Source project called MPS (Meta-Programming System). One way to evaluate MPS performances for designing simple language compiler was to implement the SimpliC language in MPS through its projectional editor. We will present in this paper what a projectional editor is and its opportunities for language design. Then we will introduce basic concepts of language design in MPS through our SimpliC implementation, and discuss the lessons we learned from this. We also discuss an evaluation of this process and present related academic work.

1 Introduction

Designing compilers is a complex task, requiring different steps. Usualy a compiler [2] first translates *source code* written on text files into *tokens*. Then the token list is *parsed* to an *Abstract Syntax Tree* (AST). Now we can add semantic analysis aspects like name analysis, type checking and error checking. There are several ways to do that, namely through *visitors* [2] or with *Attributed AST* [3]. From this last production, compiler can generate intermediate code and *target code*. All these steps require different techniques, usually with different tools.

The parsing step is often a bottleneck when designing language [9]. Indeed, if we want to *extend the language*, we will need to change the grammar to introduce new language constructs and keywords. This step may be difficult and may introduce unparsable grammar. It can be avoided if we directly work on AST, adding and editing tree nodes.

That is why we introduce *projectional editing* [9]. Instead of writing programs on text files and then have a parsing process that can be complex, we use projectional language. Using a projectional editor, developer will modify directly the AST stored in XML or database. Thus, we can easily edit program in just editing AST nodes instead of complete language edition. It avoids concrete grammar or parser use and enables designer to extend language without changing language grammar.

JetBrains provides a tool called *Meta-Programming System* (MPS). It stores the code into *models* instead of text files, providing projectional edition [7]. MPS is an Open Source project[1], under Apache License. MPS Editor provides a structure that allows better flexibility for language design, including extension and visualisation [9].

MPS represents programs in tree *models* stored in XML files. The tree represents the *structure* of program and each node is called *concept*. Each concept can have hierarchical inheritance and specific rules, allowing complex structure constructions. Thus, the program can be represented in the editor, i.e. projected, into different views, using *editor* aspects. For instance, the same program can be projected into textual notation, diagram editor or symbolic editor [7]. Moreover, *constraints* and *behaviours* add rules to construct program structure. Finally, previous aspects of language contruction enable MPS to perform a lot of static analysis such as *type checking*, *name analysis* and *code generation*.

For educational purpose, we used a C-like language called *SimpliC*. SimpliC is a reduced-concepts language with most of C language constructs: function declarations, function calls, typed variables, expressions, control structures .... Compiler course uses this simplified language to teach students how to handle main compiler concepts. SimpliC is implemented in Java, in connection with JFlex[2] (lexical analyzer generator), Beaver[3] (LALR(1) parser generator) and JastAdd[4] (meta-compilation system that supports Reference Attribute Grammars (RAGs)).

We will study on this paper how to implement SimpliC language and compiler in MPS. Our goal is to build a compiler for SimpliC language into MPS environment instead of using previous mentioned tools. We want to implement SimpliC language in a standalone SimpliC IDE (Integrated Development Environment) that will provide projectional editor to allow SimpliC programs edition and compilation.

The main activities are to implement the core concepts of SimpliC on MPS. Then we try to implement simple SimpliC programs as study cases to demonstrate the language use. Finally, we release a standalone SimpliC IDE.

All this activities have as backgroud aim to compare the two ways of implementing compiler: using JetBrains MPS or using Java with JFlex, Beaver and JastAdd. The benefits of MPS implementation for a language is that in the same time you get an IDE and a compiler for SimpliC without adding other features.

We try to answer the following question: "What is the extra work for creating an editor for a language we already have a compiler?"

[1] https://github.com/JetBrains/MPS

[2] https://jflex.de/

[3] http://beaver.sourceforge.net/

[4] http://jastadd.org/web/

Course paper, EDAN70, Lund University, Sweden
January 30, 2020.

program.in

```
1  int fac(int n) {
2      if (n <= 1) {
3          return n;
4      }
5      return fac(n-1) * n;
6  }
7  int main() {
8      print(fac(7));
9      return 0;
10 }
11
```

Figure 1. Simple SimpliC program

The rest of this paper is structured as follows: Section 2 introduces the useful backgroud to understand what we expect from MPS. Section 3 presents how we implemented SimpliC in MPS Project, using IDE functionalities, sandbox and standalalone IDE generation. Section 4 introduces the criteria we can use to evaluate SimpliC implementation. Section 5 presents other tools than MPS as projectional editors. Section 6 provides a short summary and a conclusion.

2 Background

2.1 SimpliC language

The SimpliC language is a small C-like language. It allows developers to create programs composed of named function with parameters and return values.

This language is namely composed of function declarations, statements, expressions and variable uses. Variable are typed (`void`, `int` or `bool`) and numerals are allowed. Comparison, binary and unary operators are defined for expressions and common control structures (`if/else`, `while` and `return`) are defined.

We usually edit SimpliC programs in text editors or classic IDE. See Figure 1.

2.2 Projectional editing

The classical approach to write software source code is to write down the program on simple text files. This way is really easy, any text editor allows to open and edit text files and modern IDE allow developers to manage projects with a lot of source files. Nevertheless, this approach may lead to issues when designing and extending languages: one may want to add new constructs to a language to solve a specific aspect of an application.

For example, one may want to use tabular representation for some computations into C programs, but textual edition does not support this format. Scanner and parser will not handle this construct and will fail to compile the program. Or one could want to represent process using diagram notation but most classical IDE do not provide this feature and text files cannot support diagram storage *as is*.

For thoses reasons we try to use projectional editing [8]. This approach allows to manipulate directly the AST, instead of working with a parser. The editor has special rules to allow user to edit tree, add, remove or edit nodes.

Then developers and non computer science specialists do not need to know the language constructs and check the correctness of their source code implementation because the language is already built when they create new nodes in the AST. The programs are stored in IDE internal format, generaly XML files, and language users only manipulate models. This projectional edition allows several program views: text-like, tabular, diagram, tree…Moreover, the same program can be viewed in different views at the same time. Finally, this approach allows language extenions, just by adding new nodes types and tree construction rules. This new types do not break parsing rules because no parsing rule is applied.

2.3 Meta-Programming System

JetBrains MPS is an open source projectional language workbench [7]. MPS is an Integrated Development Environment allowing language design using projectional edition. MPS is based on JetBrains Java core architecture. MPS is bootstraped, i.e. the languages we use for desing inside MPS are projectional languages themselves (MPS base language).

MPS allows to create two project types: 1) Custom language design and 2) Development using custom languages. Custom language design (1) allow developers to create new languages for their specific application with their own rules. Development projects (2) allow developers to apply their newly created languages on specific application solutions.

MPS represents programs in tree *models*. The tree represents program *Structure* and each node is called *concept*. Each concept can have hierarchical inheritance and specific rules, allowing complex structure constructions like ancestry, children and reference concepts. Thus, the program can be represented in the editor, i.e. projected, into different views, using *editor* aspects. Default editor for a concept is a textual view with indentation of each child. Moreover, one program can be projected into textual notation, diagram editor or symbolic editor [7]. *Constraints* add rules to construct program structure. *Behaviours* allow languague designer to add methods to nodes. Previous aspects of language contruction enable MPS to perform a lot of static analysis such as *type checking* and *name analysis*. Finally, a program an be exported in text format using *Text Generation* rules.

Figure 2. Comparison between Conventional compilers and MPS

Conventional Compiler	MPS
Programs are written in files	Programs are written on Models
AST	Structure
Parser	No parser / Editor
Attributed AST	Behaviour
Type Analysis	TypeSystem
Code generation	Text Gen

2.4 Summary

Conventional compiler use several steps to translate programs written in text files into executable code. They are modualble and efficient but may be difficult to extend and improve.

MPS solves parser bottleneck in proposing a projectional editor and many other structures to implement custom languages.

We can compare conventional compilers and MPS in table 2.

3 Implementation

We will explain how we implemented SimpliC language in MPS using different language design aspects. Then, we will present the embedded sandbox system we used to check SimpliC implementation in MPS. Finally, we will introduce the stadalone IDE generation from MPS and some interesting functionalities of projectional editing.

3.1 SimpliC aspects

MPS manipulates programs as trees of concepts, like an Abstract Syntax Tree. A concept is an program element caracterized by several language aspects. An aspects view in MPS can be seen Figure 3.

One main aspect of a concept is the Structure aspect (1): it gives concept a name and its relationship with other nodes. Then the Editor aspect (2) gives MPS rules to project each concept in projectional editor. Behaviour aspect (3) allows language designer to add methods to nodes like JastAdd. TypeSystem aspect (4) adds types support for variables and return types. Finally, Text Generator aspect (5) enables to translate a concept into text, that is the compile step to get assembly code from a SimpliC program.

3.1.1 Structure

A Structure aspect is the first aspect to implement. It has a name, it extends a BaseConcept or another abstract Structure, it has own members and has links to other nodes (children or reference). MPS provides also classical concepts interfaces to be implemented like `INamedConcept` or `ITypedConcept`, providing automatic name/type analysis and suggestion.

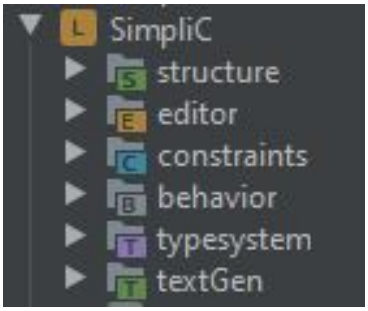

Figure 3. SimpliC language aspects

Implementing SimpliC Structure aspect was quite easy: we simply translated the concrete grammar definition for SimpliC into MPS concepts. We have a `Program` root structure and abstract structures like `Statements` and `Expressions`. We also have concrete structures like `FunctionDeclaration` and `Param` or `Assignment`. We have `IdDeclaration` statements and `IdUse` expressions. Finally we implemented all unary and binary expressions like comparison operators or mathematical operators.

At this step, we are able to create any SimpliC program in the default editor.

3.1.2 Editor

The Editor aspect is really important to give our programs the projections we expect to edit SimpliC programs. This step is tedious because we choose to mimick the text-like behaviour of projectional editor for SimpliC in order to let developers feel comfortable with our IDE.

An Editor aspect for a concept consists in cell layout design. Each concept is projected in projectional editor through its cell. Therefore we defined every concepts editors as indentation layout for text-like behaviour. Then we added concepts keywords and properties to make SimpliC programs look text-like editable.

As example, take a look at IdDeclaration Editor (Figure 5). The Editor for this concept starts with the type, is followed by the name and finally has an optional semicolon. The inspector MPS tool allows us to show semicolon only if the IdDeclaration concept is a standalone Statement. Otherwhise, IdDeclaration is member of a function parameters list or a declaration/assignement and should not be followed by a semicolon.

We also added comments lines and implemented simple syntax highlighting.

3.1.3 Behaviour

The Behaviour aspects add methods to the concepts to extend their abilities, like JastAdd.

For example, we added a builtin concept for `print` function in SimpliC programs. As we can see Figure 6, every new

SimpliC program has a builtin function `print` and a bare `main` function.

We can also add new methods but we did not exploit this opportunity.

3.1.4 TypeSystem

A TypeSystem aspect is available in MPS to handle type checking and compatibility.

In this study, we only implemented simple types (`void`, `int` and `bool`), allowing projectional editor to suggest them. In future work, we can use this aspects to check expression types and function call return types for example.

3.1.5 Text Generation

The TextGen aspects are really similar to Code Generation we implemented for SimpliC in compiler course. Thier implementation was relatively easy because it is the same as previously done.

We have basic text generation, i.e. each concept appends it's own assembly code to MPS output. We implemented all parts of assembly code except labels, that require more effort using Behaviour aspects.

At this step, we were able to generate the compiled ASM code corresponding to a SimpliC program.

Another aspect exists in MPS, called Generator aspect. This aspect name can lead to confusion with Text Gen and is only related to translation from one language to another by MPS.

3.2 Sandbox

One powerfull tool in MPS is the Sandbox system.

When creating a language design project in MPS, one can add an attached Sandbox that uses the new language. It allows language designer to create dynamically simple programs and check that changes in language design corresponds to what is expected.

In particular, one can use the new language as a released version and work completely using MPS framework. All features are directly available like name and type suggestion and checking, projectional editor interaction and AST manipulation. One can also use the Text Gen to check that compiled code corresponds to what is expected.

3.3 Standalone IDE

MPS is able to generate a standalone IDE that embedds our specific designed language with all JetBrains editor features.

To do so, one should use the generic version of MPS to have the required binary artifacts for all supported operating systems (Windows, Linux, MacOS).

Once generated, the SimpliC standalone IDE can be installed in any environment and run an instance of a JetBrains editor with all projectional editor features. One can namely create new program using SimpliC language and use JetBrains editor features (name and type check, name suggestion, autocorrection).

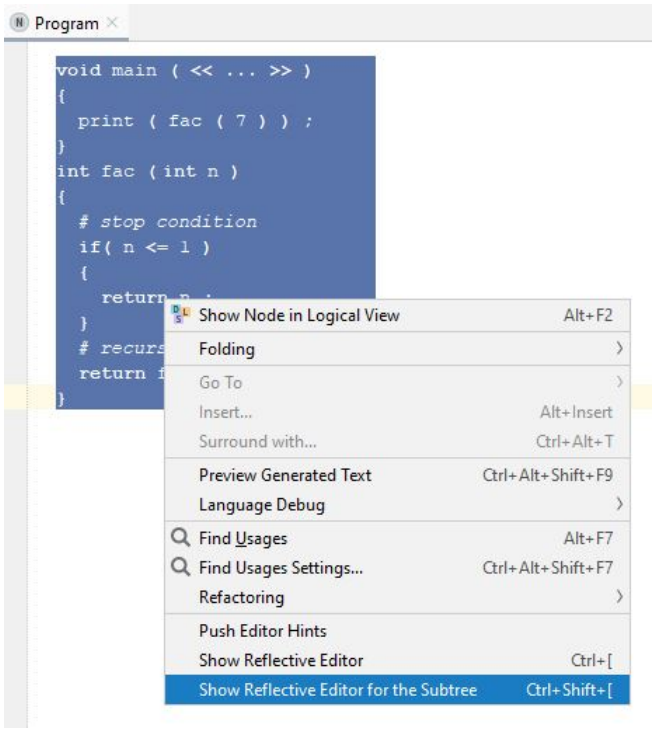

Figure 4. Reflective editor menu

One interesting feature is the ability to switch between Regular and Reflective Editors as program projection. This can be applied to all program or just some nodes. See Figure 4 as example on a simple program.

Finally, text generation is available to compile any SimpliC program into assembly code.

3.4 Summary

We have been able to implement some SimpliC language constructs on MPS using different language aspects and MPS features. Projectional editor is used in text-like mode to allow easy understanding by SimpliC designers.

We were able to release the implementation as a MPS project, containing SimpliC implementation, Sandbox examples and standalone IDE build scripts. We also released an Alpha version of Standalone IDE for various operating systems.

Some SimpliC constructs are not fully implemented and functionnal but can be improved following same steps than used previously.

4 Evaluation

This study was focused on implementing SimpliC in MPS.

The metrics we used was a sample program in SimpliC as factorial (see Figure 1). This factorial program uses most of all SimpliC aspects:

1. Main function and another function definition with parameter
2. Function call, builtin function call and recursive call
3. If statement and return statement
4. Boolean and integer binary expressions

Moreover, we used to add some of id declarations and assignement to have a complete coverage of all kind of language constructs.

We were able to test that it was easy to use MPS projectional editor to write this kind of simple programs using the Sandbox system and through the Standalone SimpliC IDE we generated.

Note that MPS allows embeded automated unit test suite [6], but this feature was outside the scope of this study. The problem for an automated test approach was that the project was not mature enough to provide relevant test cases.

One other way onf evaluating this study is the self-grader tool for compiler course. This tool tests our SimpliC implementation by running generated compiler. Howerver, our production is not a Java-based compiler but a complete IDE solution.

5 Related work

Voelter studied 2010 MPS projectional language workbench for modular languages implementation [9]. This study introduces projectional edition and JetBrains MPS. It introduces most of concepts presented above and includes a clear step by step example for language implemention on MPS. This paper has been the first step of our study.

Three years later Voelter studied MPS projectional editor for language and IDE modularization and composition [7]. He introduced new MPS features for language design and modularization. The concepts were more focused on language modularization and it was out of the scope of this study.

One year later Voelter, Siegmund, Berger and Kolb studied user-friendly projectional editor instead of using text files to represent source code [8]. They focused on challenge for most of developers to interact with projectional editors and how to solve the drawback of unfamiliar editing experience. In spite of projectional editors usability issues, their results show benefits of flexible language composition when projectional editors emulate parser-based editors. We used their advices for implementing our SimpliC editor.

One example of practical application of specific language implementation in MPS is Klimeš study [4]. Klimeš implemented a prototype of his own language Eddie in MPS and produced a Standalone IDE. His work is more complete than this one, with static analysis and more complete type system. However, it was an inspiring paper, presenting clearly MPS concepts from a more practical point of view than previous ones.

An interesting study focused on automated testing for specific language implementation [6]. This study presents several approaches to increase the automation of language testing. Automated testing is an important aspect of language implementation but it comes after a good knowledge of language specifications and tools handling. This is why we did not deepen this aspect of our project.

Finally, some other solutions than MPS exist for language design [1]. For exemple Eclipse Xtext, Spoofax or MetaEdit+. Most of them are released as Eclipse plugins for specific language design. They provide an interesting alternative to MPS solutions.

6 Conclusion

We will conclude on the work we released, answer the question we asked and expose personal feelings.

6.1 Released work

We implemented several aspects of SimpliC in JetBrains MPS. We used a Sandbox system to check that most of SimpliC language constructs are implemented and usable. We generated a standalone IDE embedding SimpliC implementation to create SimpliC programs and compile them into assembly code.

We can say that we produced an IDE for SimpliC for the first time.

Our results can be useful for educational purpose, to enable students to learn language design through several aspects. MPS approach is original with projectional editor, language design and embeded features. The standalone IDE may be used to design simple SimpliC programs, use name and type suggestions and text generation. The projectional edition is particularly interesting through editor switch, to see hax AST nodes are directly modified when editing programs.

6.2 Future work

Our implementation is focused on previously introduced code example (Figure 1). Standalone editor allows a straightforward implemention of this code snippet. Howevern, SimpliC implementation could be improved through several aspects.

For example, we lack some expressions implementation (rule priority or parenthesis). These rules can be easily implemented using work from previously implemented SimpliC aspects and with help from online documentation. Moreover, `if/else` and `while` statement support are not fully operational. Their implementation may be improved and new statament (`for` loops) may be introduced.

Text Gen should be fixed for function calls and structure with labelling system. Note that we did not check generated

code to compile using `ld` and `as`. Name analysis may be improved for scope declaration and use. Syntactic coloration can be implemented for keywords, structure, ID declaration/use…

One may implement `struct` constructs to join Object Oriented programming to SimplC, as well as pointer support or modular language extension. One may also add a new projectional editor for a diagram view, like what we can find in DrAST [5].

6.3 Research question

Through this study, we tried to answer the question "What is the extra work for creating an editor for a language we already have a compiler?".

For one who never experienced projectional editor, an extra work is necessary to handle this concepts. Then it is strongly recommended to follow MPS user guide and project examples to become friendly with MPS tool and languages. Finally, if previous steps are done correctly, we estimate that implementing complete SimpliC language support in MPS can take approxiately 20 hours.

The entire furnished work previously written can easily lead to a Standalone SimpliC editor for educational purpose.

6.4 Feelings

MPS seems to be easy to learn, as it is presented. But one have to get deep in knowledge of the tool to be able to get the same thing you might do with another tool.

MPS uses internal Base Language, where no reference is given. Then it is sometimes difficult to get a quick idea of what is needed and relevant for our specific project.

Projectional editing is a real advantage in editing AST and avoid parser step bottleneck. Approaching this edition aspect is sometimes difficult but really powerful when we got it in hands.

Acknowledgments

We would like to thank Alfred Åkesson for his support in carrying out the study and for his practical advice.

References

[1] 2016. Language Workbench Challenge. https://2016.splashcon.org/track/lwc2016#About. (2016).

[2] Andrew W Appel. 2004. *Modern compiler implementation in C*. Cambridge university press.

[3] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.

[4] Jonáš Klimeš. 2016. Domain-Specific Language for Learning Programming. (2016).

[5] Joel Lindholm and Johan Thorsberg. 2016. DrAST - An attribute debugger for JastAdd. (2016). Student Paper.

[6] Daniel Ratiu and Markus Voelter. 2016. Automated Testing of DSL Implementations: Experiences from Building Mbeddr. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST '16)*. ACM, New York, NY, USA, 15–21. https://doi.org/10.1145/2896921.2896922

[7] Markus Voelter. 2013. *Language and IDE Modularization and Composition with MPS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11

[8] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 41–61.

[9] Markus Völter and Konstantin Solomatov. 2010. Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. (01 2010).

A Appendix

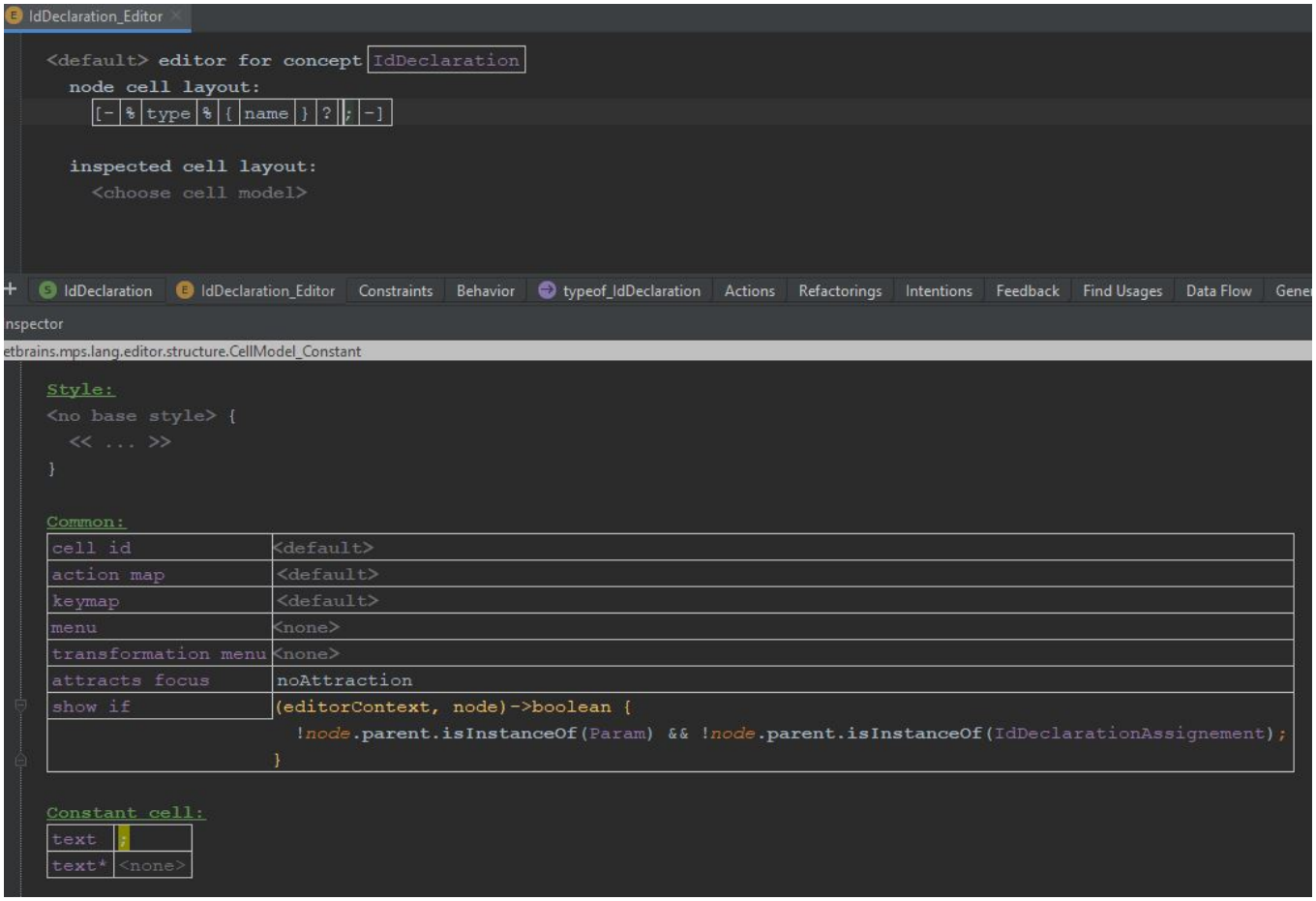

Figure 5. IdDeclaration Editor aspect

```
concept behavior Program {

  constructor {
    // Add print function
    this.functionDecl.add(new node<BuiltInPrint>());

    // Add main function
    node<FunctionDeclaration> main = new node<FunctionDeclaration>();
    main.name = "main";
    main.type = new node<VoidType>();
    main.block = new node<Block>();
    this.functionDecl.addFirst(main);
  }

  <<concept methods>>
}
```

Figure 6. Program Behaviour aspect