[CHR26] J. Charles, A. Hellwig, B. Rumpe:
Generating Language Servers for MontiCore-based DSLs.
In: Modellierung 2026, Volume P378, pp. 123–139, LNI, DOI 10.18420/modellierung2026-08, GI, Bayreuth, Mar. 2026.

Generating Language Servers for MontiCore-based DSLs

Joel Charles [1], Alexander Hellwig [1], and Bernhard Rumpe [1]

Abstract: Domain-specific languages (DSLs) are widely used in model-driven engineering to increase the development speed and quality compared to conventional software projects, but they often lack the sophisticated language support IDEs provide for general-purpose languages. The language workbench MontiCore can be used to efficiently develop DSLs, but editor support has to be implemented from scratch for each language. To decrease the effort of creating editors for MontiCore DSLs, this paper presents a generative approach, which creates IDE plugins using the language server protocol. This is achieved by converting the MontiCore grammar of a DSL into a language server and generating accompanying plugins for the popular IDEs IntelliJ and VSCode. Overall, the development experience using DSLs is brought in line with general purpose languages.

Keywords: Domain-Specific Languages, Model-Driven Software Engineering, Software Language Engineering, Language Server Protocol

1 Introduction

General-purpose programming languages (GPLs) have accustomed developers to editor support through integrated development environments (IDEs) [ALG18]. They increase the efficiency of their users through convenience features such as syntax highlighting, navigation to symbol definitions, and error highlighting [DN06]. Syntax highlighting makes keywords easier to recognize, which helps with language recognition. At the same time, the support features facilitate the early detection of code problems.

In model-driven software engineering (MDSE) [BCW17], editor support has been less prevalent in the past [Bu20, Gr08, KMK19]. In MDSE, domain-specific languages (DSLs) are used to model the domain. Language workbenches are used to design DSLs, for which manual editor support must be created. Due to the additional effort required, editor support was often not implemented in practice, which negatively affected modelers' modeling experience. Accordingly, models were sometimes written in plain text, then checked and further processed using separate tools. This approach requires modelers to switch contexts, which frequently reduces their productivity [KMK19]. Recently, there has been a growing trend towards holistic modeling tools in MDSE [Ch25, Rä25].

In the 21 years since its inception, the language workbench MontiCore [HKR21] has proven in practice that it can be used to develop textual DSLs efficiently. It has led to the creation

1 RWTH Aachen University, Software Engineering, Ahornstraße 55, 52074 Aachen, Germany,
charles@se-rwth.de, https://orcid.org/0000-0003-3427-7772;
hellwig@se-rwth.de, https://orcid.org/0009-0001-1698-4054;
rumpe@se-rwth.de, https://orcid.org/0000-0002-2147-1966

of textual variants of UML [Sc12], which are used, for example, by MontiGem [Ad18]. MontiGem is a model-driven generator for information and management systems that generates full-stack applications. A prominent example that emerged from MontiGem is MaCoCo [Ge24], a management cockpit for controlling, with over 69 running instances at RWTH Acchen University's institutes and faculties, used to manage finances, projects, and time recording for employees.

In this publication, we answer the question of *how automated editor support can be generated for MontiCore-based DSLs*. The generative approach almost eliminates maintenance caused by language evolution or extension for language engineers. At the same time, the established implementation covers a wide range of IDEs with a single implementation.

For tools created using MontiCore, the focus of development is on the tools' functionality [HKR21]. Less attention has been paid to ergonomics for modelers during model editing, e.g., by providing short feedback loops for syntax errors. So far, editor support has been created only for individual MontiCore-based DSLs, requiring significant manual effort. The implementations have always been created for a specific IDE and could not be reused in other IDEs. Due to the effort involved, editor functionalities that support modelers have been implemented only in a rudimentary form, or not at all. In [Ku18], editor support for the MontiCore-based DSL EmbeddedMontiArc was created, integrated directly into an application rather than an regular IDE. Since language support is closely tied to the application, it cannot be reused in common IDEs.

Even after language support has been established, existing editors require significant maintenance work from language engineers, which has already led to the discontinuation of several editors. The concrete syntax of a DSL/GPL changes as the language evolves [Bu18, KRV07]. One example is Gradle build scripts, which are a prominent example of their switch from Groovy-based build scripts to the Kotlin DSL. Language support in editors must keep pace with language changes in order to remain relevant.

Structure In section 2, we explain the basics of the Language Server Protocol and the language workbench MontiCore, for whose DSLs this work introduces Language Server support. This is followed by section 3 which compares our approach to related work. Section 4 specified the external requirements for a solution, which are specified by our MontiCore use case. Section 5 addresses our solution approach. First, subsection 5.1 introduces the MontiCore Language Server Generator, which generates an LSP-compatible language server from an MC grammar. Furthermore, the integration into MontiCore's language engineering workflow is discussed. Next, the internal architecture of our approach is examined in subsection 5.2. The management of artifacts with MontiCore during editing requires an adjustment to support interactive feedback, as explained in subsection 5.3. A selection of supported LSP features is discussed in subsection 5.4 and the application of the generator to an existing language is shown in section 6. Finally, section 7 concludes.

2 Preliminaries

Language Server Protocol (LSP): The LSP was introduced as a standardized layer between text editors and computer languages. The LSP introduced by Microsoft in 2016[Mi16], alongside the early releases of the IDE Visual Studio Code (VSCode), has the primary goal of decoupling language-specific tooling (e.g., autocompletion, diagnostics, refactoring) from the editor (LSP Client) that consumes it.

Modern IDEs provide developers with advanced features such as code completion, symbol navigation, real-time diagnostics, and refactoring support. Traditionally, these features were implemented independently within each IDE, resulting in duplicated effort and inconsistent user experiences across tools. Before LSP, language tooling was tightly coupled to specific IDE frameworks. For instance, the Eclipse ecosystem relied on language-specific extensions such as the Java Development Tools (JDT) or the Dynamic Languages Toolkit (DLTK), while Visual Studio integrated similar capabilities through proprietary APIs. This tight coupling made it difficult to reuse language services across different editors.

LSP introduced a unifying abstraction that decouples the language logic from the editor frontend, thereby enabling interoperability between multiple clients and servers. The separation enables the editor to work in a language-agnostic manner. The editor forwards language-specific queries (in a language-agnostic format) to a language server (LS), which responds in a language-agnostic way, for example, instructing the editor to jump to a specific location in the document. In order to process the responses from an LS, the editor must provide the LSP client implementation once. By moving the heavy lifting to a separate process, any editor that implements the LSP client can instantly obtain sophisticated language features without re-implementing them for each language. At the same time, language developers can support all editors with a single LS implementation, as the LS always communicates with a uniform counterpart (LSP client). This architectural decision has dramatically increased reuse of language support code and lowered the entry barrier for new editors to provide high-quality tooling for a wide range of GPLs and DSLs.

In essence, the LSP provides a common communication interface for typical IDE actions, including text synchronization (e.g., document open, change, save events), code completion, diagnostics and error reporting, symbol navigation and refactoring support. The LSP is comprised of a client–server architecture based on a standardized set of JSON-RPC messages. Their exchange can take place using any transport mechanism (such as standard I/O or TCP). The messages are divided into two categories. First, some requests refer to a specific document. The LS's `TextDocumentService` processes them. Requests that refer to the LS's internal state or to multiple documents are processed by a `WorkspaceService`.

Since its initial release, the LSP has evolved under the stewardship of Microsoft and the open-source community. The specification is maintained publicly on GitHub and has undergone several revisions to support new language features, semantic tokens, and workspace-level operations. The open and extensible design of LSP has contributed to its wide adoption

across the software engineering ecosystem, turning it into a de facto standard for language tool interoperability.

Language Workbench MontiCore: MontiCore [HKR21] is a language workbench that facilitates the modular development of DSLs. Modularity is a key factor in MontiCore's concept of reusability across all areas of language engineering. Context-free grammars are the central development artifact in language development with MontiCore and describe the structure of both the concrete and abstract syntax of the developed language.

Instead of developing the grammar for each language from scratch, MontiCore provides existing grammars as a library, allowing them to be reused and extended through language composition [Ha15]. Based on a MC-Grammar, MontiCore generates a large portion of the language infrastructure, including the data structure for the abstract syntax tree (AST), a parser, and symbol management. A symbol table is constructed from the AST by adding additional links between AST elements, thereby converting its hierarchical representation into a graph-based structure. MontiCore's Context Condition (CoCo) framework allows constraints to be checked at both the AST and the symbol table levels. Using the so-called TOP mechanism, MontiCore provides a way to extend the generated code with handwritten code (HWC) without the risk that subsequent generator calls will overwrite it.

3 Related Work

This paper does not aim to compare and evaluate the advantages and disadvantages of different workbenches. Instead, this section highlights related work in automatic LS generation. Language Workbenches mentioned differ from each other in their design decisions and from MontiCore. Nevertheless, the underlying industry trend toward providing editor support is evident, even in workbenches that are not examined in detail, such as MPS (as they provide an editor using proprietary solutions [VL14]).

Xtext is an Eclipse-based language workbench for developing textual DSLs [Be16]. Its core contribution is a generator that, from a single grammar specification, produces a full-featured language infrastructure. The generated tools include an editor with syntax highlighting, code completion, and refactoring support backed by an language server conforming to the LSP. Overall, language engineering with Xtext is tightly coupled with the Java and Eclipse ecosystem. Nevertheless, the generated LS enables the creation of DSL-compliant models in any LSP client-compatible IDE.

One industrial player in the field is Typefox, whose core competence lies in developing DSLs and related infrastructure. They maintain various open-source projects, including *Langium* [Ty21]. Since its 2021 release, Langium has positioned itself as a modern, lightweight language workbench. Based on the language definition, language support is automatically provided via a generated LS. The LS provides essential language features by default, with no adjustments required. However, the LS provided can be further customized by handwritten

code. Unlike Xtext and MontiCore, which rely on Java, Langium uses a TypeScript-based language server that can run in the browser, but integrating it into the Java-based MontiCore toolchain increases development complexity and complicates integration with existing functionality.

Both the Langium technology stack and technical design decisions prevent direct compatibility between MontiCore and the Langium language definitions. To leverage Langium's capabilities to generate LS for MontiCore-based DSLs, complex AST transformations would need to be implemented. Research is needed to determine how this could be technically realized. The same holds for Xtext. Nevertheless, such an approach would increase the complexity of MontiCore's language infrastructure.

In the modeling community, multiple contributions have been made towards editor support for languages created with language workbenches. In [Ja25] a framework for language engineering and model management is proposed. Similar to our approach, the grammar of the language is used as the central artifact to derive editor support. Via a translation to a Langium grammar, the previously described generator structure can be reused to generate a language server and corresponding editor plugins. Since the infrastructure generated by MontiCore is not compatible with TypeScript or Langium, this approach would lead to a major rewrite of existing MontiCore DSLs. In [Bü19], a language server for a textual DSL is generated using Xtext, and manually integrating it into two IDEs is reported to take several developer hours per language. In our approach, IDE plugin generation is automated, shifting this effort from the language engineer to the tooling developer. Many other contributions from [Ja25], [Ro18], and [HI25] focus on lsp editor support for graphical DSLs. Since MontiCore only supports textual DSLs, these are not applicable here.

Accordingly, DSL development with MontiCore requires its own solution. Both Xtext and Langium demonstrate that a language server can be automatically derived from a grammar. However, neither addresses the specific workflow and architectural implications of MontiCore. Consequently, our contribution lies in systematically adapting MontiCore's model-driven architecture to support the LSP. By extending MontiCore's existing code-generation pipeline, we generate LS artifacts while reusing the DSL's symbol table, type system, and Context Conditions. Our implementation bridges the gap between the model-centric ecosystem of MontiCore and the generated, platform-agnostic LS infrastructure pioneered by Xtext and Langium for language workbenches.

4 Language Support for MontiCore

Providing editor support for MontiCore-based DSLs is highly desirable. On the one hand, it significantly increases modelers' comfort by improving accessibility and efficiency. On the other hand, competing language workbenches already facilitate the creation of editors through a generative approach. To clearify the goals of our editor support, three requirements are defined:

Requirement 1 (RQ1) - Editor support for MontiCore DSLs Advanced IDEs significantly improve the productivity of developers using general purpose languages. For textual DSLs, such as the languages created with MontiCore, most of the features of modern IDEs —such as highlighting, completion, and navigation —can also be implemented. Thus, editor for support for MontiCore DSLs is needed and would improve the usage experience and productivity of modelers.

Requirement 2 (RQ2) – Automatic Generation of Editor Support: One of MontiCores main features is the automatic generation of tool infrastructure from a languages grammar, which significantly reduces the effort to develop new DSLs. To make the development and maintenance of accompanying editors viable, they also need to be generated from the same grammar. Otherwise, the additional effort of adapting handwritten code becomes overwhelming and agile language development is no longer possible.

Requirement 3 (RQ3) – Integration into an established IDE: In principle, a domain-specific editor could be generated from scratch for each DSL, but the effort to write and maintain a specific editor is non-trivial. A domain-specific editor also locks the modeler in the corresponding ecosystem, which might not align with the rest of the development environment. To avoid both the effort of maintenance and lock-in in, the integration into existing IDEs is desirable.

The LSP offers a solution: a single language-server implementation can expose language features (completion, diagnostics, navigation, refactoring, etc.) through a uniform protocol that many editors implement. By implementing one LS per DSL, the same functionality is made available across the entire spectrum of supported editors. Because all major IDEs already support LSP, requirements 1 and 3 can be fulfilled by creating a language server for each MontiCore DSL. By generating the language server from the MontiCore grammar of a language, the necessary effort can be kept low and requirement 2 is fulfilled.

The approach described in the rest of this paper provides an integration of a language server generator into the existing MontiCore infrastructure and demonstrates how a LS can be generated directly from an MC-Grammar.

5 The MontiCore Language Server Generator

5.1 Integrating Language Server Generation with MontiCore

The MontiCore Language Server Generator (MCLSG) is a dedicated component of the MontiCore ecosystem. MCLSG enables automated generation of editor support for MontiCore-based DSLs. It is responsible for generating an LSP-compliant LS, freeing language engineers from manually implementing LS and IDE plugins for each newly created DSL, thereby bridging the gap between language definition and editor support.

Figure 1 shows the language development workflow with MontiCore, which has been extended in this work to include MCLSG. The input artifacts (yellow) are processed by the tools (gray) to generate their output artifacts (green). Further adjustments to the MontiCore generator's behavior via Freemarker templates are possible, but are not relevant to this work and are therefore not illustrated.

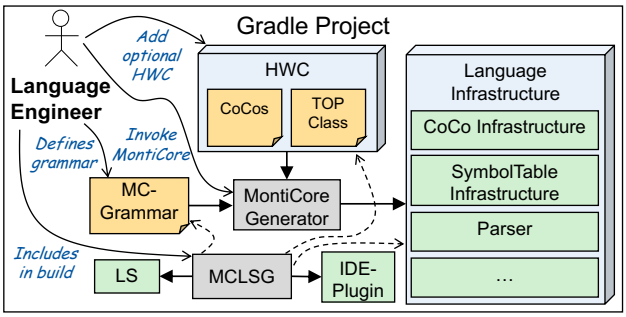

Fig. 1: The MontiCore language development workflow, adapted with the MCLSG

Grammar Definition: The first step involves the language engineer defining both the concrete syntax and the abstract syntax, which are encapsulated as an MC-Grammar artifact. Optional, manual modifications can be integrated by the language engineer using the TOP-mechanism. Additionally, they are able to specify well-formedness rules by means of CoCos.
Invocation of MontiCore Generator: Although MontiCore can be invoked via the CLI, in practice, a language project is often orchestrated using Gradle. The MontiCore generator is called repeatedly during language engineering. First, to create the AST from the MC grammar. Then, the language engineer writes CoCos based on the generated AST and may add other HWC via the TOP-Mechanism. To ensure the HWC is accounted for in the generation, the generator must be rerun. Gradle allows accelerating builds by incrementally reusing previously built artifacts.
MCLSG Invocation: MCLSG is invoked as a post-processing step in a MontiCore language workbench workflow. The MCLSG consumes the artifacts (grammar, AST, symbol table) generated by MontiCore and creates the LS implementation together with IDE plugin projects.

5.2 Internal Structure of MCLSG

In addition to the LS, our approach includes the generation of IDE plugins (see Figure 1). Unlike previous plugins created for MontiCore languages, they do not contain language-specific logic, which minimizes the maintenance needed as the language evolves. Instead, they serve as a light weight integration layer between specific IDEs and the LS, by registering the language to the editor and starting the language server. The generator uses runtime classes, defined by `LSP4J`, to generate concrete subclasses for each DSL, which contain the DSL-specific aspects. The concept abstracts from the protocol while allowing the language

engineer to integrate HWC. Handwritten behavior can be added by providing additional templates or by subclassing the generated classes via the TOP-mechanism, without breaking the overall pipeline. To provide uniform tooling across IDEs for modelers, our plugins rely on the identical IDE-agnostic LS.

5.3 Document Management Lifecylce

Typically, MontiCore relies on the file system to obtain the current state of an artifact. Model changes must be persisted in the file system so that MontiCore can recognize and process them. While modeling in an editor, models are not continuously written to the hard disk. At the same time, a modeler expects immediate feedback from the editor on their input without having to save. Accordingly, the LSP features must be available before model artifacts are persisted on the hard disk. For the interactive editing scenario, the MCLSG introduces the `DocumentManager`, which maintains an internal representation of the workspace that reflects the current state of the language project.

At LS runtime, the `DocumentManager` orchestrates the lifecycle of text documents. For each document, the manager maintains a `DocumentInformation` object containing the latest version of the content, the AST, resolved symbols, and diagnostic information. This representation enables incremental recomputation of analyses and immediate feedback in the IDE.

After the project is initially loaded by the LS, indexing is performed on all relevant artifacts of the DSL. Every time the modeler changes an artifact, the LS receives a *didChange* notification by the LSP client, as described in the protocol, and updates the artifact's contents in the `DocumentManager`. The MontiCore language frontend is executed for each newly added or modified artifact. Therefore, the artifact's full content is parsed, and AST-based CoCos are executed. If no error occurs, the symbol table is created in one of two ways.
When no resolved symbol references are stored in the symbol table, it can be updated by deleting the old scope and all descendant symbols, and inserting the new artifact-level scope at its position.
In contrast, if other artifact scopes save symbols resolved of the changed artifact, a simple replacement will not update these saved symbols. In this case, the entire symbol table needs to be deleted and rebuilt whenever the user changes. Both variants are performant enough for interactive use. Afterwards, the symbol-based CoCos, such as the MontiCore typecheck, are executed.

During these steps of the language frontend, all errors are collected with corresponding source positions. The resulting findings are then converted into user-readable messages and used to annotate the artifacts in the editor via the *publishDiagnostics* notification.
In total, the user is notified of simple errors such as "missing semicolon in line 5", as well as more complex errors such as violated CoCos. Interruptions in the symbol table creation process, such as a failed type check, are also reported.

Through this indexing and change-tracking mechanism, the LS maintains an up-to-date view of the workspace, encompassing the current artifact content, parsed ASTs, and the corresponding symbol table. These three components form the foundation for implementing all other LS features in our system.

5.4 Supported LSP Features

The LSP supports a wide range of editor functions. Table 1 lists the features currently supported by the MCLSG and indicates the extent to which they can be generated. To allow language engineers to replace the generated implementation features, the provider pattern is used throughout the implementation of the LSP services. Each request and server-directed notification defined in the LSP is encapsulated in its own `Provider` interface. By implementing a new subclass, these hookpoints can be used to add handwritten features that currently not automatically generated. The generation of selected features is explained in more detail in the following subsections.

Feature	**Support**	**Comment**
Code Action	Infrastructure Generated	Extension point for handwritten actions
Code Lens	Infrastructure Generated	Extension point for handwritten annotations
Completion	Generated	Completion for keywords and symbols
Diagnostics	Generated	Reports parser and CoCo errors
Document Symbols	Generated	Display of all symbols in a document
Formatting	Generated	Pretty printing of models
Goto Definition	Generated	Navigation from Symbol usages to definitions
Goto References	Generated	Navigation from Symbol definitions to usages
Highlighting	Generated	Semantic highlighting based on symbols
Hover	Generated	JavaDoc Style comments
Rename	Generated	Renaming of symbols

Tab. 1: Implementation level of editor features in the MCLSG

5.4.1 Highlighting

Many editors provide syntax highlighting by processing artifacts with compatible grammar specifications, such as TextMate. Since the MontiCore Grammar language is not natively supported by any of the large IDEs, highlighting needs to be implemented in the language server to make it available to multiple editors with managable effort. This is done with the *semantic token* request of the LSP, which, in most language servers, is used to enhance simple syntax highlighting with annotations based on semantic information, such as type errors. In our case, this feature is used both for simple highlighting, e.g. keywords, and complex situations, such as an unresolved type. This alleviates the need to transform MontiCore grammars into a widely supported grammar format compatible with most IDEs.

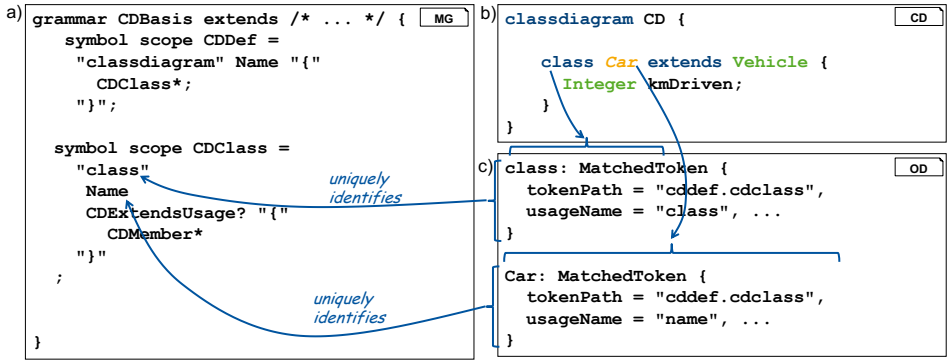

Fig. 2: Calculation of syntax highlighting from parsed token

To enable highlighting, the parser generated from the MontiCore grammar is reused. While parsing a document during indexing or after a user change, all consumed tokens are captured and stored as `MatchedToken` instances. During this step, the stack of all productions used for a token are captured as a token path, as seen in Figure 2. The token `class` of the parsed class diagram (b), is parsed in the context of the production `CDClass`, from the example MC-Grammar (a). `CDClass` in turn is parsed as part of `CDDef`, and thus, has the token path `cddef.cdclass`. The identity of the terminal that is used to consume the token is saved as the usage name by analysing the current state of the parser.

The LSP defines a default set of token types, such as `variable` or `keyword`, and modifiers, `declaration` or `async`, which are converted to apply a style to tokens by the editor, often using user-defined color themes. To turn the captured `MatchedToken` instances into the token types and modifiers required by the LSP, a pattern matching based approach is applied. Multiple patterns based on token paths and usage names are tested, and the first matching rule is applied. Some base rules are added to each generated LS, which provide highlighting for shared language constructs, such as keywords, strings, and comments. To add language-specific highlights, the language engineer can add handwritten rules, for example, to color the name of class definitions in a different color than its usage (as illustrated in 2 (b)).

5.4.2 Navigation of Usage to Definition

Quick navigation between the elements of a DSL increases productivity by reducing the time modelers spend locating symbol definitions or usages. In the example from Figure 3 (a), the user wants to inspects the definition of class `Car`, based on a usage in an association between `Car` and `Person`. The LSP defines the `definition` request to find the defining position of an element.

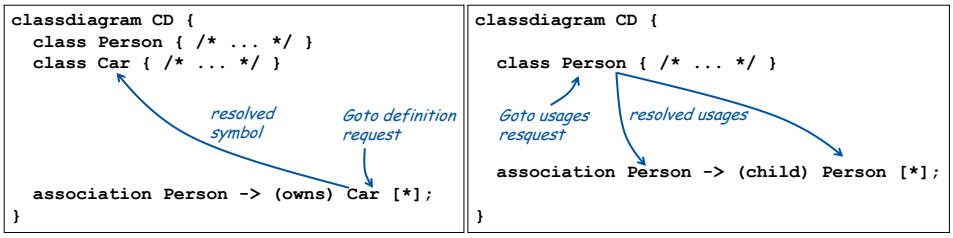

Fig. 3: Symbol related navigation from a) usage to definition b) definition to usages

To implement this request, MontiCore's symbol table and generated symbol resolving infrastructure is used to resolve symbol definitions by their name. It follows the visibility and scoping rules defined in the grammar and handles symbols imported from other artifacts. Each resolve call is limited to one symbol kind; for example, it is not possible to find all attribute- and type-definitions with name `Car` within the same resolving procedure. This increases efficiency and ensures type safety when the symbol kind is known. In the language server, this is only the case if a symbol usage is restricted to a single kind, as in the production `CDExtendsUsage = "extends" Name@CDClass;`, where the `Name` must reference a `CDClass` definition. Otherwise, all symbol kinds are allowed, and one resolving procedure must be executed for each known symbol kind of the language and symbol kinds of the language components it uses. From the resolved symbol definition, the corresponding artifact and position are extracted and sent to the editor, where the user is pointed to the requested definition.

5.4.3 Navigation from Definition to Usages

To estimate the impact of changing a symbol definition in a model, it helps to examine all its usages. In our example Figure 3 (b), triggering the `references` request on the `Person` class definition returns its two references within the self-association of `Person`. Since MontiCore has no built-in functionality to resolve all usages of a symbol, the feature has to be reimplemented on top of the normal resolution algorithm for symbol definitions. In the LS, the internal representation of the workspace contains an index of all documents in the workspace, including all artifact scopes of the symbol table. From these, all symbol usages are identified and a resolve call to the symbol table as in subsubsection 5.4.2 is performed. If the resolved symbol matches the original definition, the reference is considered a *valid usage* and is retained in the result set. Otherwise, if the resolved symbol differs, it is discarded, since it either does not have the correct symbol kind or is not visible in the current scope. This filtering ensures that only true usages of the original symbol are reported by the `references` request, avoiding false positives caused by different symbols with identical names. Since this operation is computationally expensive, each resolve call is only triggered if the name token of a symbol usage is equal to the name token of the symbol definition.

5.4.4 Completion

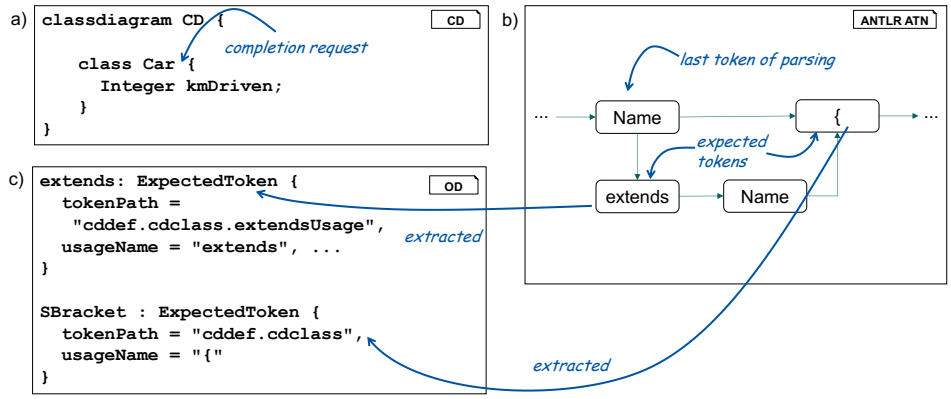

Fig. 4: a) The context for the completion request b) An exerpt of the ATN of the parser c) The two relevant extracted ExpectedToken

During modeling, completion suggestions at the current position improve speed by (1) guiding unexperienced users with valid language constructs such as keywords, and (2) suggesting all available and valid symbols in the current scope. When a `completion` request is triggered by the editor, it uses internal state of the generated MontiCore parser to extract all keywords and symbol names that are allowed at this position. The generated MontiCore parser is based on Antlr, and thus, it contains an Augmented Transition Network (ATN) [PF11]. The ATN is a graph which contains all possible tokens as nodes and the transitions between them as edges. By running the parser on the current document up until the position where the completion should be returned, the ATN can be used to extract the next possible tokens by inspecting all outgoing transitions from its current state.

In the example in Figure 4, a completion is triggered after the symbol definition `Car`, and thus, the document (a) is parsed up to the character "{", which is opening the body of the class. The corresponding ATN (b) is in the state reached by consuming `Car` as a `Name` token. Follwing the transitions, the keyword `extends` and the opening curly bracket are identified as expected token (c).

These expected tokens are then passed into a rule-based approach similar to the one described in subsubsection 5.4.1, where the developer can adapt the completion by adding handwritten rules. The default completion rule added to each LS implements several out-of-the-box features based on the grammar of the language. All keywords are suggested in the appropriate positions, since they can be uniquely identified from the expected token kind. For positions where a symbol reference is possible, all symbols visible in the current scope are suggested, including symbols added via import statements. If the usage type is constrained in the production, e.g. via an annotation such as ‘Name@CDClass‘, all symbols of incompatible kinds are filtered out.

6 Example

To demonstrate that the generator can derive a language server and editor plugins from a MontiCore grammar, it is applied to the existing CD4Analysis language [He23]. Without modification of the language definition or implementation, the language server can be derived by the generator and includes base features, such as syntax highlighting and navigation between symbol definitions and usages. Additionally, VSCode and IntelliJ plugins are generated[2]. A screenshot of the resulting syntax highlighting can be seen in Figure 5.

The architecture of MontiCore and its language server framework allows for hand-written additions via the provider pattern and the TOP-Mechanism. In the CD4Analysis example, these mechansims are used to add code lenses to provide additional information to the modeler. In this case, they give an overview of all associations of a class, which can be seen as *Part of 2 Associations* in the screenshot.

Through this, the requirements, as defined in section 4, are met. `RQ1` is fulfilled, since a framework for editor support in MontiCore exists and can be used for all MontiCore languages. `RQ2` is met, as the language server can be generated from the grammar, reducing the maintenance effort. Based on the generated language server, plugins for VSCode and IntelliJ are also generated. Thus, `RQ3` is satisfied.

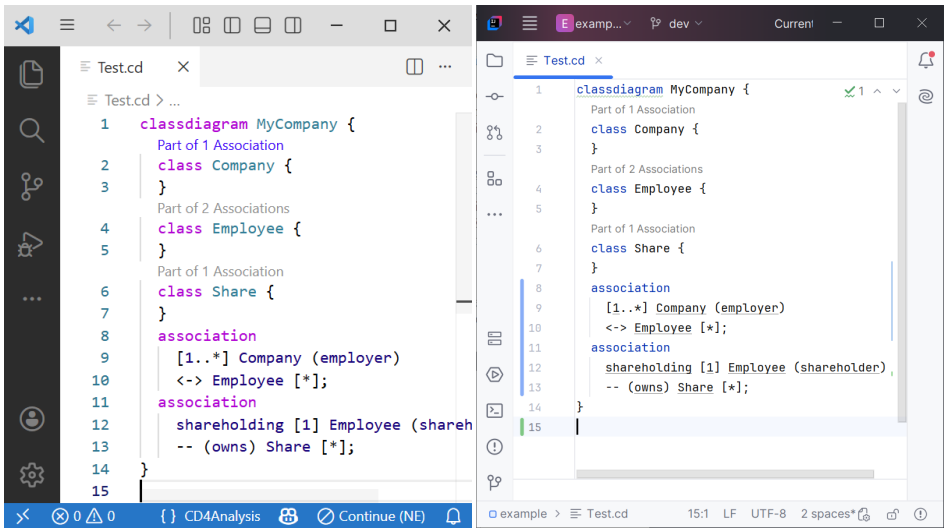

Fig. 5: Screenshot of the generated Visual Studio Code (left) and IntelliJ (right) plugins for the CD4Analysis language

2 The VSCode and IntelliJ plugins are available at `https://github.com/MontiCore/cd4analysis/tree/dev?tab=readme-ov-file#editor-plugins`

7 Conclusion

This paper has demonstrated (1) how the language server generator is integrated into the MontiCore workbench and (2) how MontCore's grammar-based language implementation is used to generate language server features. The MCLSG lays the foundation for broad editor support for MontiCore-based DSLs and opens several avenues for further research and extension of the language workbench.

First, extending the generator to support additional IDE ecosystems would broaden its applicability. Second, integrating more LSP features such as, complex in-place refactorings further narrows the gap between editors for general purpose and domain specific languages.

While a LS is derived directly from the MC-Grammar using the presented approach, many DSLs can benefit from handwritten extension of the editor, which can often not be derived from the grammar. Thus, for each language in the existing MontiCore library, a handwritten adoption of the generated language server needs to be considered. To reduce the effort, the language composition operators of MontiCore have to be included into the generated language servers, which would allow for reuse of existing handwritten language server features when using language components [HKR21]. To compose the language servers of these components, an approach similar to [BCF25] could be used.

In addition, further development to derive more LSP features directly from the grammar is needed. Investigating features not yet covered by the LSP and their inclusion in the protocol is another possible line of work.

Finally, establishing a public catalog of reusable LSP extensions for common MontiCore DSLs would enable language engineers to assemble high-quality editors with minimal effort.

In summary, MCLSG constitutes a decisive step toward systematic, low-effort development of LSP capable editor plugns for MontiCore-based DSLs. By encapsulating generation logic in a modular, template-driven architecture and by building upon a reusable architecture of MC-Grammars, the tool automates LS development while stil preserving the opportunity to introduce HWC extensions with further language-specific features.

Acknowledgements

Funded by the German Federal Ministry of Education and Research - 03G0922A

Bibliography

[Ad18] Adam, Kai; Netz, Lukas; Varga, Simon; Michael, Judith; Rumpe, Bernhard; Heuser, Patricia; Letmathe, Peter: Model-Based Generation of Enterprise Information Systems. In (Fellmann, Michael; Sandkuhl, Kurt, eds): Enterprise Modeling and Information Systems Architectures (EMISA'18). volume 2097 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 75–79, May 2018.

[ALG18] Afzal, Afsoon; Le Goues, Claire: A study on the use of IDE features for debugging. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 114–117, 2018.

[BCF25] Bruzzone, Federico; Cazzola, Walter; Favalli, Luca: Code Less to Code More: Streamlining Language Server Protocol and type system development for language families. Journal of Systems and Software, p. 112554, 2025.

[BCW17] Brambilla, Marco; Cabot, Jordi; Wimmer, Manuel: Model-driven software engineering in practice. Morgan & Claypool Publishers, 2017.

[Be16] Bettini, Lorenzo: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd, 2016.

[Bu18] Butting, Arvid; Eikermann, Robert; Kautz, Oliver; Rumpe, Bernhard; Wortmann, Andreas: Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18). ACM, pp. 75–82, January 2018.

[Bü19] Bünder, Hendrik: Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In: MODELSWARD. pp. 129–140, 2019.

[Bu20] Bucchiarone, Antonio; Cabot, Jordi; Paige, Richard F; Pierantonio, Alfonso: Grand challenges in model-driven engineering: an analysis of the state of the research. Software and Systems Modeling, 19(1):5–13, 2020.

[Ch25] Choudhury, Abhishek; Malavolta, Ivano; Ciccozzi, Federico; Aslam, Kousar; Lago, Patricia: The technological landscape of collaborative model-driven software engineering. Software and Systems Modeling, pp. 1–25, 2025.

[DN06] Dujmović, Jozo J; Nagashima, Hajime: LSP method and its use for evaluation of Java IDEs. International journal of approximate reasoning, 41(1):3–22, 2006.

[Ge24] Gerasimov, Arkadii; Letmathe, Peter; Michael, Judith; Netz, Lukas; Rumpe, Bernhard: Modeling Financial, Project and Staff Management: A Case Report from the MaCoCo Project. Enterprise Modelling and Information Systems Architectures - International Journal of Conceptual Modeling, 19, February 2024.

[Gr08] Gray, Jeff; Fisher, Kathleen; Consel, Charles; Karsai, Gabor; Mernik, Marjan; Tolvanen, Juha-Pekka: DSLs: the good, the bad, and the ugly. In: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. pp. 791–794, 2008.

[Ha15] Haber, Arne; Look, Markus; Mir Seyed Nazari, Pedram; Navarro Perez, Antonio; Rumpe, Bernhard; Völkel, Steven; Wortmann, Andreas: Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In: Model-Driven Engineering and Software Development Conference (MODELSWARD'15). SciTePress, pp. 19–31, 2015.

[He23] Heithoff, Malte; Jansen, Nico; Kirchhof, Jörg Christian; Michael, Judith; Rademacher, Florian; Rumpe, Bernhard: Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report. In: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2023, Association for Computing Machinery, Cascais, Portugal, pp. 194–207, October 2023.

[HI25] Hinkel, Georg; Igler, Bodo: An internal DSL for graphical modeling tools based on GLSP. J. Object Technol., 24(2):2, 2025.

[HKR21] Hölldobler, Katrin; Kautz, Oliver; Rumpe, Bernhard: MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[Ja25] Jaeger, David; Lencses, Adam; Fleck, Martin; Langer, Philip; Bork, Dominik: A Model Management Framework for Next-Generation Web-based Modeling Tools. J. Object Technol., 24(2):2, 2025.

[KMK19] Kos, Tomaž; Mernik, Marjan; Kosar, Tomaž: A tool support for model-driven development: An industrial case study from a measurement domain. Applied Sciences, 9(21):4553, 2019.

[KRV07] Krahn, Holger; Rumpe, Bernhard; Völkel, Steven: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In: Conference on Model Driven Engineering Languages and Systems (MODELS'07). LNCS 4735. Springer, pp. 286–300, 2007.

[Ku18] Kusmenko, Evgeny; Ronck, Jean-Marc; Rumpe, Bernhard; von Wenckstern, Michael: EmbeddedMontiArc: Textual Modeling Alternative to Simulink. In: Proceedings of MODELS 2018. Workshop EXE. October 2018.

[Mi16] Microsoft: Language Server Protocol. `https://github.com/microsoft/language-server-protocol`, 2016. Accessed: 2025-10-24.

[PF11] Parr, Terence; Fisher, Kathleen: LL (*) the foundation of the ANTLR parser generator. ACM Sigplan Notices, 46(6):425–436, 2011.

[Rä25] Rädler, Simon; Berardinelli, Luca; Winter, Karolin; Rahimi, Abbas; Rinderle-Ma, Stefanie: Bridging MDE and AI: a systematic review of domain-specific languages and model-driven practices in AI software systems engineering. Software and Systems Modeling, 24(2):445–469, 2025.

[Ro18] Rodriguez-Echeverria, Roberto; Izquierdo, Javier Luis Cánovas; Wimmer, Manuel; Cabot, Jordi: An LSP infrastructure to build EMF language servers for web-deployable model editors. In: MoDELS (Workshops). volume 2245, pp. 326–335, 2018.

[Sc12] Schindler, Martin: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Ty21] TypeFox GmbH.: Langium. https://github.com/eclipse-langium/langium, 2021. Accessed: 2025-10-24.

[VL14] Voelter, Markus; Lisson, Sascha: Supporting Diverse Notations in MPS'Projectional Editor. In: GEMOC@ MoDELS. pp. 7–16, 2014.