



Article

Generation of Custom Textual Model Editors

Eugene Syriani ^{1,*}, Daniel Riegelhaupt ^{2,†}, Bruno Barroca ^{3,†} and Istvan David ^{1,†}¹ Department of Computer Science and Operations Research, Université de Montréal, Montréal, QC H3C 3J7, Canada; istvan.david@umontreal.ca² Independent Researcher, 2000 Antwerp, Belgium; daniel.riegelhaupt@outlook.com³ Independent Researcher, 1000 Lisbon, Portugal; mailbrunob@gmail.com

* Correspondence: syriani@iro.umontreal.ca

† These authors contributed equally to this work.

Abstract: Textual editors are omnipresent in all software tools. Editors provide basic features, such as copy-pasting and searching, or more advanced features, such as error checking and text completion. Current technologies in model-driven engineering can automatically generate textual editors to manipulate domain-specific languages (DSLs). However, the customization and addition of new features to these editors is often limited to changing the internal structure and behavior. In this paper, we explore a new generation of self-descriptive textual editors for DSLs, allowing full configuration of their structure and behavior in a convenient formalism, rather than in source code. We demonstrate the feasibility of the approach by providing a prototype implementation and applying it in two domain-specific modeling scenarios, including one in architecture modeling.

Keywords: domain-specific language; modeling editor; model-driven engineering; statecharts



Citation: Syriani, E.; Riegelhaupt, D.; Barroca, B.; David, I. Generation of Custom Textual Model Editors. *Modelling* **2021**, *2*, 609–625. <https://doi.org/10.3390/modelling2040032>

Academic Editors: Ludovico Iovino and Amleto Di Salle

Received: 24 August 2021

Accepted: 2 November 2021

Published: 6 November 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

One of the main benefits of model-driven engineering (MDE) is the increased customizability of software applications [1]. Moreover, MDE offers an automated control of customizations based on abstract syntax structures, such as metamodels. This allows software designers to reuse tools generated from models in a wide range of application domains, while being predictable and ensuring a certain degree of correctness. Although MDE has been widely applied in software engineering [2,3], we still observe a considerable shortage of customizable modeling tools for mainstream languages, such as the ones proposed by the OMG (e.g., UML class diagrams, SysML [4]) and SAE (e.g., AADL [5]).

Textual editors are omnipresent in all software tools. Some editors provide basic features, such as writing text, copy-pasting, and searching (e.g., notepads), while others offer very advanced features, such as error checking, formatting, and text completion (e.g., programming IDEs). Current technologies in MDE, such as MPS [1] and Xtext [6], can automatically generate textual editors to manipulate domain-specific languages (DSLs). However, the reuse of these generated domain-specific textual editors is hindered by the substantial manual effort required for adapting the generators to new application contexts. Moreover, the ability to customize or add new features to generated editors is often limited to the specification of both the metamodel of the DSL and the grammar specifying its concrete syntax. More detailed customizations of the editor, such as the structure and behavior of the editor, pose significantly more complex challenges and might not be feasible at all. In general, manual modifications to automatically generated editors are cumbersome, error-prone, and hinder both the reusability and maintainability of the editor.

Explicitly modeling its structural and behavioral aspects can significantly improve the customizability of the editor. Our research hypothesis is that appropriately chosen modeling languages, particularly UML Class Diagrams (for the specification of the structure) and Statecharts (for the specification of the behavior), can be employed for such purposes and lower the customization cost of textual editors. Such an approach enables the reuse of

editors as rich-text controls in possibly any modeling environment. A pertinent application domain of the approach is system and software architecture modeling. The Architecture & Analysis Design Language (AADL) [5] has been widely employed for modeling the architecture of complex systems. AADL editors, such as the Open Source AADL Tool Environment (OSATE) (<https://github.com/osate/osate2>, accessed on 4 November 2021), AADL Inspector (<https://www.ellidiss.com/products/aadl-inspector>, accessed on 4 November 2021), and the Ocarina suite (<https://github.com/OpenAADL/ocarina>, accessed on 4 November 2021), typically provide the user with textual facilities to model the architecture. This is due to the scalability issues of graphical editors as models grow. Thus, advanced textual editing features are required to facilitate an efficient engineering workflow. Additionally, AADL is used in a variety of application domains and platforms, and as a consequence, the extensibility of AADL editors bears particular importance.

In this paper, we explore the feasibility of customizing textual editors for DSLs by explicitly modeling their structure and behavior. The contributions of this paper are the following.

- We study and discuss the feasibility of employing the Statecharts + Class Diagrams formalism to customize the reactive behavior of editors. To the best of our knowledge, there are no other approaches that model the reactive behavior of textual editors using such an expressive modeling language.
- We provide a convenient grammar specification to define parsing and syntax-checking, similarly to the grammar specifications found in EMFText [7] and in Spoofax [8].
- We provide a mapping specification from the grammar specification to a style-sheets specification, specifying syntax-highlighting in a convenient language, similar to the mechanisms found in tools such as MPS [1], that enable the customization of syntax-highlighting schemas for DSLs.
- We provide an additional mapping specification from the grammar elements to an existing metamodel definition which, besides being convenient automatic completion features, can also add basic support for static semantics checking. This mechanism is similar to the ones found in tools such as Spoofax/Stratego, which are based on term-rewriting.
- We evaluate the performance and scalability of the editor using an illustrative modeling scenario. We note, however, that performance was not our primary concern at this stage of development.

The benefits of our approach include: (i) enhanced reusability and platform-independence due to the various available Statecharts compilers for different platforms; (ii) enhanced modularization and visualization capabilities—for instance, UML Statecharts are hierarchical, allowing modularization of the overall behavior inside each state, and there exist several Statecharts editors that can render and edit them in a convenient way; and finally (iii) analyzability and correctness, since UML Statecharts have limited expressiveness (no recursive calls), hence allowing automated analysis, such as in the mapping to existing timed-automata model checkers [9].

The rest of this paper is structured as follows. In Section 2, we briefly outline the state-of-the-art of customizable textual editors. In Section 3, we present the architecture of our solution. In Section 4, we demonstrate the customization of the editor from three different points of view: end user, language engineer, and tool builder. In Section 5, we report on the performance and scalability of the current prototype implementation, using realistic modeling scenarios. Finally, we conclude in Section 6.

2. Customization in Existing Text Editors

A focal point of many modern editors and language workbenches is their support for customization. This is often achieved in concert with an emphasis on the reusability of features, such as parsing and interactive syntax checking (e.g., in atom.io (<https://atom.io/>, accessed on 4 November 2021) and EMFText [7]), type checking and corresponding error

annotations (e.g., in Spoofax/Stratego [8]), automated syntax coloring, (semi-)automatic text completion, and go-to definitions.

The traditional way of expressing customizations is to expose a set of options on a data structure, handled by a class of algorithms, accessible by an API, or ultimately a DSL. Tools like Xtext [6] and EMFText [7] provide productive DSLs to define grammars using the Extended Backus-Naur Form (EBNF). The set of production rules is organized in a way that implicitly defines a parsing state machine. While this is convenient, this comes at the expense of an implicitly imposed workflow, impacting the reusability of these tools. For instance, in EMFText, the customization workflow starts from an existing metamodel (abstract syntax), and then the grammar is defined by implicitly referring to the metamodel types. In Xtext, two workflows are possible: one is similar to EMFText, and the other starts with the grammar specification from which the metamodel is automatically generated subsequently. In contrast, our approach does not impose any particular workflow, as both the structure and behavior of the editor can be completely reformulated in any order. For instance, one could start describing the grammar to provide information for the parser component, while completely neglecting any data structure or model repository.

Spoofax [8] provides customizability through mapping rules from the Syntax Definition formalism to a particular type system, captured as rewrite rules of a generic transformation language. This concept allows other aptly designed solutions, e.g., for syntax highlighting. Like the underlying parsing mechanism of Spoofax, our approach is also modular. Composing grammars is achieved by the same mechanism as referencing non-terminals of one grammar in the bodies of the production rules of another grammar. The mechanism is based on a simple, scanner-less production rule evaluation technique, combined with the power of context-free grammars. The scanner-less approach avoids complex grammar composition in which lexers and parsers are composed separately like in MontiCore [10], for example.

Projectional editors, such as MPS [1] and Gentleman [11], offer a different approach to parsing and rewriting in a modeling environment, as well as different user interaction strategies. MPS is a textual syntax-directed editor that presents limited options to the users on each keystroke. For instance, MPS provides an interactive parse-rewrite option, whereas Xtext uses the regular parse-once-rewrite option. For other options, the users have to resort to writing source code. In contrast, our approach supports the definition of entirely unpredictable interaction strategies. This level of freedom is enabled by Statecharts + Class Diagrams (SCCD) formalism [12]. Our approach is less beneficial in the case of simplistic customizations, as modifying the SCCD model can be more cumbersome than interacting with the tools MPS. However, the benefits of our approach are more evident in the case of more intricate customizations. The SCCD formalism that explicitly models the editor ensures easier changes, better reuse, and a less error-prone customization process, as opposed to the code-based approach of MPS. Furthermore, SCCD has great expressive power; thus, it is especially appropriate for capturing complex behavioral and structural patterns. For example, SCCD allows the definition of parallel and hierarchical states, hence easing the specification of autonomous, reactive systems, such as graphical user interfaces and, in particular, our customizable text editor.

Unlike our approach, MPS and Spoofax cannot be easily reused or ported to a platform other than their intended ones. Spoofax is available for Java and C, and MPS is grounded in C. In the philosophy of their approach, they do not aim for their models to be reused. The lack of tools supporting the easy customization of structure and behavior is mainly due to their strong ties with specific programming platforms. This requires re-implementing and re-compiling many parts of the tool, imposing a significant load on the end-user. In contrast, our approach promotes abstracting editors from specific programming platforms and focuses more on the explicit modeling of their essential components. Customizing and porting explicitly modeled editors is a much simpler endeavor, especially when augmented with the proper generative techniques producing high-quality platform-specific code from the models. Such avenues have been explored by Beard [13], who proposed Statecharts as

the modeling formalism to capture user interactions with a plain text editor. Sousa et al. [14] use a rule-based engine to enable customizability of user interactions with graphical editors, demonstrating the utility of explicitly modeled editors in the realm of graphical modeling as well. Although it is not directly applicable in our approach, further customizing the interaction with a textual editor is an interesting direction.

Significant effort has been dedicated to porting the language engineering and (meta-) modeling capabilities of the Eclipse framework to the web. The Epsilon Playground project (<https://www.eclipse.org/epsilon/live>, accessed on 4 November 2021), provides a web application for experimenting with meta-modeling using the Emfatic syntax (<https://www.eclipse.org/emfatic/>, accessed on 4 November 2021) for EMF models, and the languages of the Epsilon platform [15]. The Eclipse Graphical Language Server Platform (GLSP) (<https://www.eclipse.org/glsp/>, accessed on 4 November 2021) is an open-source framework for building custom diagram editors that can be embedded into web-based tools, or used in a standalone fashion. The Language Server Protocol (LSP) enables reusing the capabilities of existing tools, allowing building complex toolchains by tool orchestration. Although similar from a user's point of view, neither of these frameworks are explicitly modeled, making the validation and verification of interaction models cumbersome. This is an especially pressing issue, e.g., in toolchain certification [16], where compliance to domain-specific standards has to be proven by verification, and often requires clear documentation at the method level.

State-of-the-art AADL editors typically rely on textual editing, with occasional support for graphical interfaces for high-level visualization. The Open Source AADL Tool Environment (OSATE) provides textual and graphical facilities for engineering AADL models. The textual representation provides a comprehensive view of all details of a system, while the graphical provides a high-level overview and quick navigation in multiple dimensions. As an Xtext-based implementation, OSATE is inherently tied to Java and specifically to Eclipse. The challenges of extending Xtext-based editors are well-documented in the scientific literature [17,18]. The AADL Inspector is a textual analysis suite that provides static analysis and simulation tools to inspect and optimize AADL models. The tabs of the user interface can be customized to display different analysis results, but the lack of access to the editor's source code prevents more intricate customizations. The Ocarina suite provides textual facilities for the validation, schedulability analysis, and model checking-based verification of AADL models. However, written in Ada and C, the portability of the tool is minimal.

3. Architecture of TxtME

TxtME is a textual domain-specific model editor that combines high extensibility with web-based implementation. The customizability of web-based textual model editors is typically limited to the definitions and notations of concepts, relations, and properties. However, the interaction with the editor is predefined and common to any DSM editor produced by these tools. In contrast, TxtME enables the tool builder to customize the behavior of the editor by modeling it in Statecharts. Statecharts have been shown to be an appropriate formalism for such purposes by Sousa et al. [14]. TxtME provides essential IDE features, such as syntax highlighting, automatic text completion, and error detection out of the box, that can be used in a standalone fashion or integrated into web-based modeling frameworks.

TxtME follows a client-server architecture, as shown in Figure 1. The *Client*-side provides textual editing facilities that are rendered based on the *Statecharts* used to model their behavior. The same *Statechart* models orchestrate the behavior of the *Parsers* and *Visitors* at the *Server* side. The *model-ware repository* provides storage and manipulation API for models. We elaborate on the architecture in what follows.

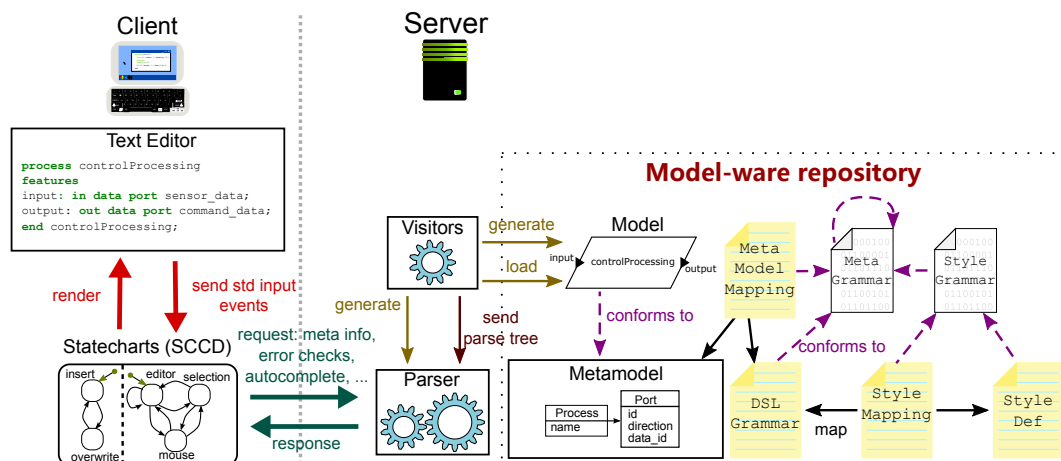


Figure 1. Overview of the main components of the architecture and their relationships.

3.1. Client

On the client-side, users interact directly with an automatically generated web-based text editor. HTML5 and CSS are used for graphical rendering and JavaScript to implement the API of the Editor. Data exchange between components is achieved via JSON. The behavior and the rendering mechanism are orchestrated using the Statecharts + Class Diagrams (SCCD) formalism. For example, the SCCD instance tracks the characters being typed in the editor and performs syntax highlighting; and identifies errors in the textual model based on the metamodel and grammar to trigger automated text completion suggestions. The SCCD formalism uses Class Diagrams to define classes and Statecharts to define their behavior and interactions [19]. This combination provides excellent descriptive power in numerous applications and aligns well with the modeling of model editors. The feasibility of such an approach has been demonstrated in previous work [20].

3.2. Domain-Specific Parsers and Visitors

The parser converts the text into a parse tree that visitors manipulate and interpret based on the SCCD model. The parse tree is defined by the grammar of the DSL, which further conforms to the meta-grammar (Appendix A). As per usual, the parse tree contains information such as tokens, white spaces, and comments. The framework provides a meta-grammar to define rules that DSL grammars must correspond to. The visitor components synthesize various information based on the parse tree, e.g., for automatic model completion, text highlighting, and text decoration. This is achieved by first generating the instance model that corresponds to the metamodel of the DSL from the parse tree. This mechanism is shown in Figure 1 through the example of editing an AADL model. However, the mapping to the metamodel is not required, allowing the modeler to define a metamodel either before or after defining the grammar. This is a characteristic difference between TxtME and mainstream editor frameworks, such as EMFText and Xtext. A style definition contains the information for the visual representation of elements, such as the fonts and colors. This style definition is mapped onto the DLS grammar, allowing the framework to render visual elements in the desired format.

3.3. Model-Ware Repository

The model-ware repository provides a storage and model manipulation back-end. All artifacts (instance model, metamodel, DSL grammar, style definition, metamodel mapping, style mapping, parse tree, and SCCD model) are considered models conforming to their metamodels. They are all stored in the repository. Similar avenues have been explored in the Modelverse [21] and AToMPM [22] projects, but in principle, any model-ware repository (e.g., Neo4EMF [23]) can be used in this architecture, provided that it implements a specific interface. The API provides kernel-specific operations, such as loading a metamodel and

parser; metamodel-specific operations, such as verifying constraints and conformance; and model-specific operations, such as CRUD operations of entities, associations, and attributes. Visitors interact with the repository to perform the operations needed by the text editors. For example, loading the model corresponding to the text in the editor, building a model while entering text, verifying if the model is valid with respect to the metamodel, and looking up references to existing elements in the model when proposing candidates for the automatic text completion.

4. Features and Customization

In this section, we discuss some of the main facilities of TxtME, such as the fully modeled front-end of the editor (Section 4.1), the grammars (Section 4.2), metamodeling (Section 4.3), syntax highlighting (Section 4.4), and automatic model completion (Section 4.5). We elaborate on the customization of these facilities by considering three characteristically different roles: (i) the user interacting with the editor; (ii) the language engineer defining the DSL and its textual concrete syntax; and (iii) the tool builder developing the editor. In the following, we consider a text editor tailored to a DSL for creating AADL models.

4.1. Fully Modeled Editor

As explained in Section 3, the text editor is controlled by an SCCD model. The tool builder can thus customize the structure and behavior of its features by modifying the SCCD model, creating a new SCCD model, and combining it with the SCCD models of graphical interfaces. Figure 2 shows an excerpt of the SCCD model used in the prototype implementation of TxtME. The two main components of the system are captured in the *client* and *editor_main* classes of the model. These classes encapsulate states that belong together and prescribe the behavior of the specific component. Roundtangles denote states, and directed links denote actions that bring a component from the source state to the target state. For example, the *left_mouse_button* action brings the editor from the *editor_mode* state to the *mouse* state. This state is exhibited as long as the *left_mouse_up* action is performed, bringing the editor back to the *editor_mode* or *selection* state, depending on whether any object was selected when the editor exhibited the *mouse* state. The SCCD formalism inherits the semantics of Statecharts and provides a high-level language to model parallelism. Expressing parallelism is achieved by orthogonal regions. For example, the *editor_main* class is further refined into two orthogonal regions, *editor_modes_parent* and *typing_mode*. This example, specifically, means that the behavior of the editor is independent of the typing mode.

SCCD is compositional, thus building increasingly complex models that can be achieved in an iterative-incremental way. This aligns well with nowadays' agile software engineering practices, ensuring a rapid reaction to user needs.

4.2. Grammars for Syntax Definition

The language engineer configures the parsers for the DSL by specifying a DSL grammar. This grammar is expressed in terms of the meta-grammar, which is bootstrapped upon starting the server. Like the Syntax Definition formalism in Spoofox, the tool builder can also configure the parser component's behavior by altering the meta-grammar, requiring changes on the underlying parsing mechanism (i.e., modify the PEG parser). The meta-grammar specifies how to define grammars and mappings, including the meta-grammar itself. Figure 3 shows an excerpt of the AADL grammar defined in [24] and implemented in our prototype.

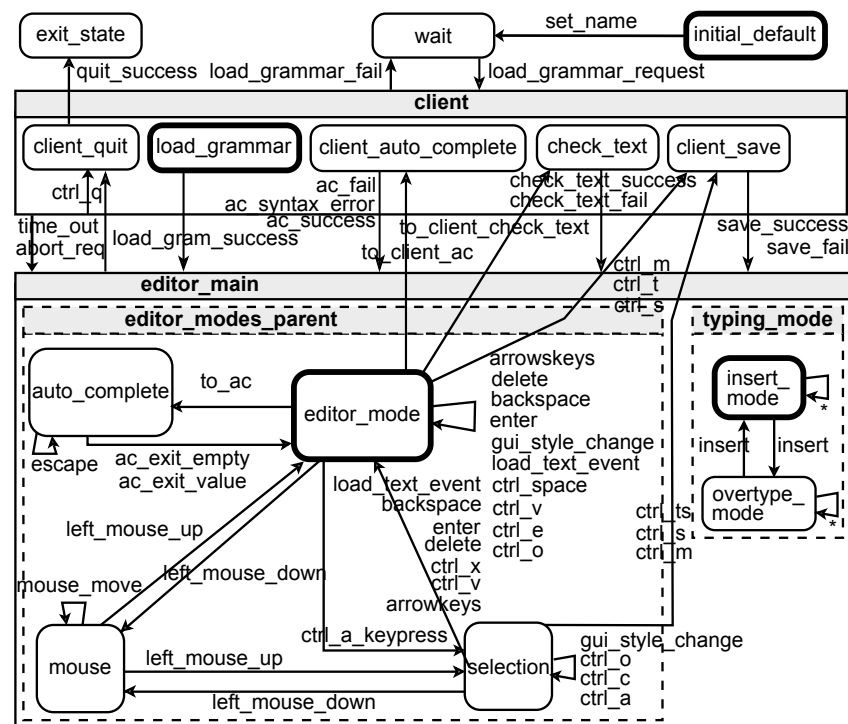


Figure 2. Excerpt of the SCCD model that defines the behavior of the editor.

```

grammar{
  start: PROCESS name (features_block | ... | ...)* END name SEMICOLON;
  features_block: FEATURES (feature)*;
  feature: (abstract feature | port | data access | bus access |
           subprogram access | subprogram group access | parameter | feature group)*;
  port: port_id COLON (IN_DATA_PORT | OUT_DATA_PORT) data_id_reference SEMICOLON;
  ...
  name: IDENTIFIER;
  port_id: IDENTIFIER;
  data_id_reference: IDENTIFIER;
tokens{
  keywords: process { PROCESS: 'process'; FEATURES: 'features'; INPUT: 'input'; OUTPUT: 'output';
                     IN_DATA_PORT: 'in data port'; OUT_DATA_PORT: 'out data port'}
  keywords: general { END: 'end';}
  COLON: ':'; COMMA: ','; SEMICOL: ';';
  DIGIT: [0-9] @Msg 'Digit';
  IDENTIFIER: '[a-zA-Z_][a-zA-Z_0-9]*' @Msg 'Identifier';
  NEWLINE: '(\r?\n|\t)*' @Impl @Msg 'New Line';
  LINE_CONT: '\\[t \f]*\r?\n' @Impl @Msg 'Line Continuation';
  WS: '[t \f]*' @Impl @Msg 'White Space';
  COMMENT: '![^\\n]*' @Cmnt @Msg 'Comment'; }
}

```

Figure 3. Simplified AADL process grammar (adopted from [24]).

Here, we only discuss the meta-grammar at a high level. The complete specification is available from the repository of the project (https://github.com/geodes-sms/txtme/blob/main/grammar_parser_v2/metaGrammar.py, accessed on 4 November 2021). Grammars expressed in terms of our meta-grammar are currently realized by means of a scanner-less Packrat parsing strategy [25,26]. This strategy uses parsing expression grammars (PEG) which have interesting properties, such as being closed under composition, intersection, and complement.

Basing the meta-grammar on PEG enables language engineers to write grammar production rules in an EBNF style. However, unlike other context-free grammars, the alternative operation represented by the ‘|’ symbol is left-associative. Therefore, language engineers must resolve the ambiguity of each token, non-terminal, and symbol: they must produce a correct ordering of alternatives ‘|’ that allows the complete and correct parsing-exploration of the intended grammar, e.g., a linear if-then-else fashion, where the smaller ambiguous choices are selected to be the first alternatives. For example, a rule in the form r :

`a|ab|ba|b;` is unable to match only `b`, since its partially ambiguous version `ba` precedes it in the alternative sequence. Instead, the rule should be formulated as `r: ab?|ba?;`.

Besides production rules and tokens, the language engineer can extend the DSL grammar with the following constructs in the form of macros.

- **Implicit (@Impl)** for token reuse. When a token is marked as implicit, that particular token can appear anywhere in the grammar without explicitly stating it multiple times. For example, in Figure 3, white spaces and newlines are defined once but can be placed anywhere in the text. Note that implicit tokens are by default removed from the parse tree unless specified otherwise by the tool builder.
- **Comment (@Cmnt)** to improve the readability of the textual model. Comment rules are predefined implicit rules, with the difference that they are retained in the parse tree. For example, in the AADL grammar, a line starting with `'//'` is considered a comment in the AADL textual model.
- **Message (@Msg)** to advise the user. The language engineer has the opportunity to return a meaningful message in the case of an error at a specific location in the text. Messages can annotate both tokens and rule definitions. For example, in the AADL grammar, the message `"Identifier"` will appear when expecting an identifier instead of the regular expression.
- **Counters** to count and parse within cycles. Although DSL grammars are context-free, language engineers can parse and increment the count of a given non-terminal whenever evaluating a production rule. In the context of a cycle (i.e., `'*'` or `'+'`), a non-terminal can be parsed an increasing number of times. Language engineers are able to specify how much each of the counters associated with a particular non-terminal is increased or decreased each time the non-terminal is evaluated on the body of a rule. This is useful when it comes to handling textual DSLs that use indentation.
- **Keywords** to group tokens. Grouping tokens is particularly useful for the mappings. A single name can refer to all tokens grouped within a keyword to organize tokens regardless of the grammar structure. Figure 4.

4.3. Metamodel Conformance

The DSL grammar is typically used as the concrete syntax specification of a DSL. A DSL grammar alone can only define a context-independent language. Hence, the resulting parsing of a given sentence will only create a concrete syntax tree. In general, however, language engineers require parsing DSL sentences to generate abstract syntax graphs. For instance, parsing in general programming languages (e.g., Java) generates a graph representing a program. Complex context-dependent rules and constraints, such as scoping (e.g., static and dynamic) and name resolution mechanisms, must also be defined. Therefore, similarly to what happens in other solutions, such as EMFText, we introduce a metamodeling language to define metamodels that represent the abstract syntax of DSLs.

After defining a DSL metamodel, the language engineer specifies a metamodel mapping to assign a concrete syntax to the abstract syntax of the language. As an example, the mapping for our AADL editor prototype is shown in Figure 5. The metamodel referred to by the mapping (in this case, `MyFormalisms.AADLProcess`) is required to exist in the Modelverse. The metamodel mapping specifies the correspondence between a metamodel element and a DSL grammar rule, and any token used within the rule. For example, the `port_decl` rule corresponds to a `Port`, where the name sub-rule (which consists of a single string) represents the name of the `Port`. Furthermore, since the `port_decl` rule uses the `data_id_reference_def` sub-rule, the name rule (which consists of a single string) represents the `data_id_reference` attribute value of the `Port`. As shown in Figure 5, the metamodel (@Model) is assumed to be a class diagram-like structure, with classes (@Class), attributes (@Attr), and associations (@Assoc). Associations can have attributes and references (@Ref) to two classes, referred to by the role names. The separation of the metamodel mapping from DSL grammar allows the language engineer to define multiple

textual syntaxes to the DSL. It also allows him to reuse the same textual specification for other DSLs.

```

styles{
  .def { font-family: 'Nimbus Mono L'; font-size: 12pt; color: rgb(0,0,0); }
  .err { text-decoration: underline; -moz-text-decoration-style: wavy; color: red; }
  .blueBold { color: rgb(51,102,255); font-weight: bold !important; }
  .greenLarge { color: rgb(0,128,0); font-size: 14pt; }
  .greyBold { color: rgb(100,100,100); font-weight: bold }
  .purple { color: rgb(128,100,128); font-weight: bold }
}
stylemap {
  @Default default: def;
  @Error error: err;
  @Keywords process: greenLarge;
  @Keywords general: blueBold;
  COMMENT: greyBold;
  port_decl: purple;
  port_decl.name: blueBold;
}

```

Figure 4. Style definition and mapping.

```

mapper {
  start -> @Model MyFormalisms.AADLProcess {
    start.name -> @Attr name;
    port_decl -> @Class Port {
      port_decl.name -> @Attr port_id;
      port_decl.direction -> @Attr direction;
      data_id_reference_def.name -> @Ref data;
    }
  }
}

```

Figure 5. Excerpt of the metamodel mapping for AADL processes.

Assigning meaning to the DSL grammar by means of the metamodel mapping allows verifying whether the text in the editor represents a correct model, i.e., one that conforms to the metamodel of the DSL. If this verification fails, the user receives the error message specified in the @Msg part of the corresponding rules.

The SCCD model can check errors both in the syntax and the static semantics. For conformance checking, the corresponding visitor requests the verification through the model-ware interface. In case of error, the message received from the model-ware repository is displayed to the user. For example, in Figure 6, the data command_data are not defined in the model.

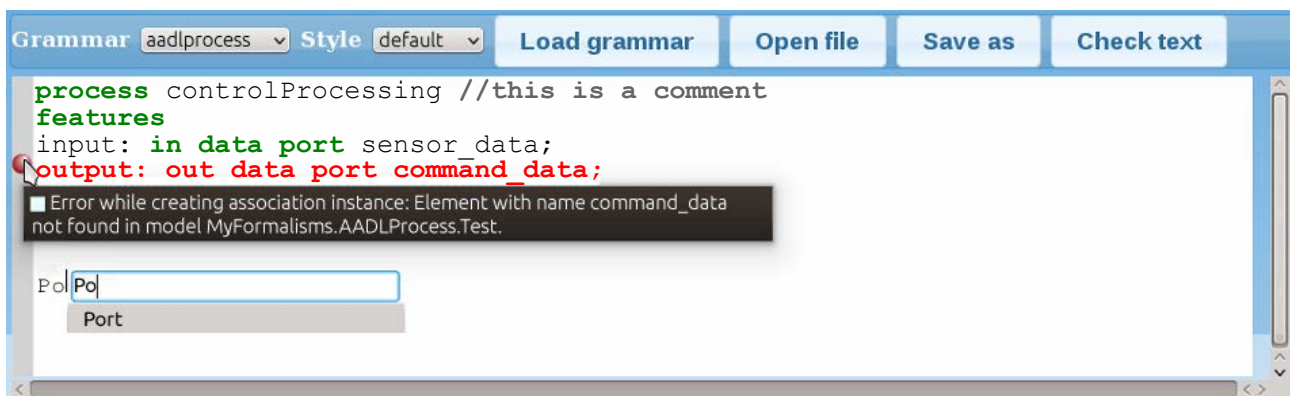


Figure 6. Screenshot of the generated AADL process textual editor.

4.4. Syntax Highlighting

The text typed in by the user is decorated to ease the readability of the textual model. A default style is applied to the text that highlights all occurrences of the keywords defined in the DSL grammar. However, in the philosophy of MDE, the language engineer should offer a model editor that best fits the application domain and the user's needs. Therefore, similarly to Spoofax and MPS, TxtME provides a DSL for style definitions. A style mapping is required to assign a specific style to a DSL grammar. This separation of concerns allows language engineers to reuse styles for different grammars. Figure 4 shows an example of a possible style definition and mapping for AADL models. Note that both artifacts conform to a style grammar which, in turn, conforms to the meta-grammar.

Styles are defined as CSS rules. We have chosen CSS because it is a declarative language, offers a complete decoration palette for text, is widely adopted, and all web browsers support it. The CSS is applied directly in the web editor and therefore works with any CSS supported by the browser, even to perform text effects and animations. For example, Figure 4 defines six named style classes. Here, `def` is the default decoration of any text that is not explicitly mapped to a style and `err` decorates text containing errors.

The language engineer specifies syntax highlighting for tokens, keywords, or rules. Token and rule highlighting are performed only after the text is parsed, as opposed to keyword highlighting. Keyword highlighting assigns a rendering style to groups of tokens. This is defined in the style mapping by referring to the name of the keywords group. For example, all tokens in the `pn` keywords will be displayed in green. Token highlighting decorates individual tokens. This is defined by referring to the name of a particular token, such as `COMMENT: greyBold`. Finally, syntax highlighting can be specified for complete rules, as in `port_decl: purple`. The language engineer can also decorate a token or sub-rule within a specific rule. For example, in `port_decl.name: blueBold`, only the `name` rule in the rule `port_decl` will be in blue and bold. Figure 6 illustrates an example of applying the style of Figure 4 to the AADL process grammar. In this example, the cascading property of the style definition and mapping renders the port declaration in three colors.

4.5. Automatic Model Completion

Automatic text completion is an essential feature of modern IDEs, since it guides the user towards what possible text can be written at a specific location. Internally, we distinguish between two flavors of this feature. Grammar-based automatic completion proposes the next token based on the current rule of the DSL grammar. For example, in Figure 6, either a port, parameter, or access to a data, bus, subprogram, or subprogram group can be declared at the current position of the cursor, according to the grammar in Figure 3. However, since the user already typed the first letters corresponding to a port declaration, `Port` is suggested. Model-based automatic completion suggests valid references to other model elements. This occurs when the current rule is mapped to an association, and the cursor is at the location of a reference. As in MPS, automatic completion is primarily configured by the metamodel mapping. For example, `input: in data port` will suggest `sensor_data` in the current model.

5. Evaluation

We discuss our measurements on performance and scalability in this section. For the sake of the completeness of our case study, we have opted for implementing a simpler language than AADL: an editor for Petri nets. Petri nets have been used in conjunction with AADL extensively, especially for verification purposes, e.g., for safety [27], dependability [28], model viability [29], and virtual timed scenarios [30]. Figure 7 shows the complete grammar of Petri nets.

```

grammar{
  start: PETRINET name (place_decl | transition_decl | arc_decl)* END;
  place_decl: PLACE name place_def? SEMICOL;
  place_def: '{' (tokens_def (COMMA capacity_def)? | capacity_def (COMMA tokens_def)? ) ';';
  capacity_def: CAPACITY COLON integer;
  tokens_def: TOKENS COLON integer;
  transition_decl: TRANSITION name SEMICOL;
  arc_decl: ARC name weight_def? COLON FROM source TO destination SEMICOL;
  weight_def: '{' WEIGHT COLON integer ';';
  name: IDENTIFIER;
  source: IDENTIFIER;
  destination: IDENTIFIER;
  integer: DIGIT+;
tokens{
  keywords: pn { PETRINET: 'Petrinet'; PLACE: 'Place'; TRANSITION: 'Transition';
    ARC: 'Arc'; WEIGHT: 'weight'; CAPACITY: 'capacity'; TOKENS: 'tokens'; }
  keywords: general { END: 'end'; FROM: 'from'; TO: 'to'; }
  COLON: ':'; COMMA: ','; SEMICOL: ';';
  DIGIT: [0-9] @Msg 'Digit';
  IDENTIFIER: '[a-zA-Z_][a-zA-Z_0-9]*' @Msg 'Identifier';
  NEWLINE: '(\r?\n|\t)*' @Impl @Msg 'New Line';
  LINE_CONT: '\\[t|f|\\r?\\n]' @Impl @Msg 'Line Continuation';
  WS: '[t|f|\\r?\\n]' @Impl @Msg 'White Space';
  COMMENT: '/*[^\\n]*' @Cmnt @Msg 'Comment'; }
}

```

Figure 7. Fully functional Petri net grammar for the experiments.

We have performed two experiments. One is to measure the responsiveness of the editor for varying model sizes. The other is to measure the effect of having multiple parallel editors working on models deployed on the same web page.

5.1. Prototype Implementation

The prototype of TxtME used in our experiments is available from its GitHub repository (<https://github.com/geodes-sms/txtme>, accessed on 4 November 2021). It uses the Modelverse [21] as the model-ware repository. The Modelverse Kernel (MvK) provides an API for performing CRUD operations on model elements, checking linguistic conformance, and executing models of computations. The server-side of the prototype is implemented in Python 2.7.8, which is the same as the parser of the MvK.

In our prototype, we implemented automatic model completion with two passes of parsing. The first pass returns all suggestions until the current position of the cursor. The second pass completes the list from the whole model. Subsequently, the parse tree is sent to the MvK, which produces the completion suggestions based on the metamodel and its static semantics.

5.2. Experimental Setup

The experiments were carried out on a virtual machine running Kubuntu 14.10 with 2 GB of RAM and a 2 GHz CPU on one available core. The virtual machine was hosted on a 2 GHz dual-core machine with 8 GB of RAM running on Microsoft Windows 8.1. The browser was Google Chrome 38 due to its support for the User Timing API in JavaScript. The server part of the architecture was deployed locally on the virtual machine as a simple Python HTTP server. The results were calculated by taking the average of 10 collected data points after removing outliers using a variability coefficient under 5%.

5.3. Experiment 1: Responsiveness

In this experiment, we measure the responsiveness of the editor for varying model sizes.

5.3.1. Case Study

The objective of the first experiment is to evaluate the performance of the different features of customized textual domain-specific model editors. More specifically, we want to assess its scalability with respect to the amount of text in the editor. Since the parser, highlight visitors, and automatic completion visitors rely on tokens and not lines of code,

we vary the number of tokens in the text (including white spaces and newlines). The text we vary is the following, where lines 2 and 3 are repeated for $i = 1 \dots 98$ to give $N = 20 + 10i$ tokens with $L = 3 + 2i$ lines.

```

1: Petrinet Test
2: Place Pi;
3: Transition Ti;
4: Arc a : from P1 to T1;
5: end

```

The metrics we use are the completion times of the following operations:

- **Load text:** The time it takes to load the raw text into the editor before parsing.
- **Check error:** The time it takes for parsing the text, creating a new model in the Modelverse, and then checking conformance. The input text is error-free to force the verification of the complete text.
- **Highlight:** The time it takes for the highlighting visitors to determine the style of each token (decorated directly or within a keywords group or a rule) and for the CSS to be applied in the browser. Since the input text is error-free, all the text is highlighted.
- **Open file:** The time it takes to load a textual model in the editor and highlight it. This is computed as the sum of load text and highlight measures, plus the fixed time to load the Petri nets grammar.
- **Auto-complete model:** The time it takes for the server to return all suggestions for a reference, based on the model. To test the worst-case scenario, we force both parsing passes to read the entire text. This is accomplished by injecting an error at line $L - 1$: removing the semicolon and requesting auto-complete at T1. This proposes 98 suggestions for the case of $N = 1000$ tokens.
- **Auto-complete grammar:** The time it takes for the server to return all suggestions, only based on the DSL grammar. This requires parsing the text up until the current position of the cursor. Thus, to test a worst-case scenario, auto-complete is requested on line $L - 1$ at to.

5.3.2. Results

Figure 8 presents the results of this experiment. The time is measured in milliseconds on a logarithmic scale. Measurements were performed for $N = 30, 100, 500, 1000$. Note that this experiment was set up to measure the worst-case scenarios. Best-case scenarios averaged times up to 10 milliseconds for each feature.

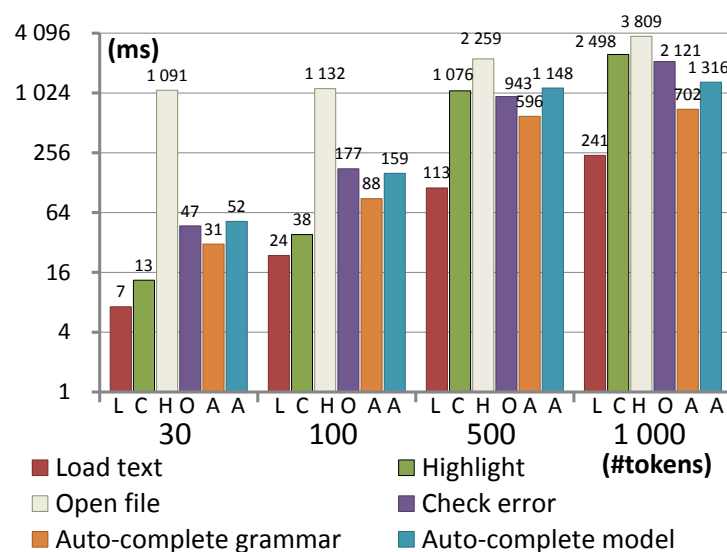


Figure 8. Worst-case features performance for varying Petri nets model sizes.

First, let us look at the performance of the operations to open a file. Note that the input model was chosen so that we compute the worst-case scenarios of each measurement, since otherwise, the results will be under 10 milliseconds. From a practical point of view, the time it takes to open an already existing textual model is crucial. This requires first to load the grammar, which, in the case of the Petri nets grammar in Figure 3, takes a fixed 1.267 s.

As expected, the time complexity for loading text is linear with respect to the model size. Check error is quadratic and one of the most costly operations because it communicates with the Modelverse. Its performance heavily relies on the processing time taken by the MvK, which is outside the scope of this paper. The time complexity for syntax highlighting is quadratic with respect to the number of tokens. This is because three highlighting visitors decorate each line corresponding to a Ti: one for the whole rule `transition_decl`, one for the pn mapping (the token `Transition`), and one for the token name. As a result, more than half of the text in the editor needs to be highlighted, which slows down the browser. Note that running the same experiment with only keyword highlighting increases the performance by 50% on average. This shows that the performance also depends on the amount of text to decorate. Another observation is that loading grammar for smaller models (<500 tokens) takes more than 95% of the time to open the file, whereas highlighting takes up most of the time for larger models.

Finally, both auto-complete operations scale linearly with respect to the number of tokens. More precisely, auto-complete depends on the number of suggestions. Model-based auto-complete performs worse than grammar-based because it communicates with the MvK to filter invalid suggestions based on the context of the request, but also because the grammar-based auto-complete only proposes one suggestion (the `to` token) at the requested location.

5.4. Experiment 2

In this experiment, we measure the effect of having multiple parallel editors working on models deployed on the same web page.

5.4.1. Case Study

From experiment 1, we conclude that highly customizable textual domain-specific model editors perform efficiently on small models that are edited quickly—at least for the current state of our prototype. An application example for such editors is the properties window of a graphical model element. It usually contains as many text boxes as the element has attributes whose value is text (e.g., strings). In an entirely modeled modeling tool, such as AToMPM, types of attributes are defined as metamodels. They can range from simple languages, such as the cardinality of an association end, to more complex languages, such as a constraint language. For such applications, multiple instances of the editor will be running concurrently. Therefore, the objective of the second experiment is to determine if the amount of concurrent editors running on the same page has an impact on the performance of the features.

In this experiment, we measure load grammar, load text, highlight, open file, and check error for the Petri net model with $N = 100$ tokens. We vary the number of editors E that attempt to open a replicate of that textual model concurrently, with $E = 2, 4, 10, 14, 20$. The case where $E = 1$ from experiment 1 is used as the basis to compare the impact of an increasing number of editors. We also perform the same measurements on a larger model with a larger grammar to rule out any bias from the nature of the model or grammar. As stated in Section 4, the meta-grammar has also been modeled to increase the degree of customization. Therefore, we use the Petri nets grammar as a textual model, comprising 522 tokens, with the meta-grammar as its grammar. Note that the auto-complete operations are not considered in this experiment, because it is not possible to request automatic text completion on multiple editors on the same page simultaneously.

5.4.2. Results

Figure 9 presents the results of this experiment run on the Petri net model with $N = 100$ tokens. The shapes of the graphs show that there is a performance penalty the moment a second editor is added. However, interestingly, the time to perform each feature remains constant, regardless of the number of editors. The feature that additional editors most impact is *check error* by a factor of 5. Load text and highlight are increased by a factor of 2, whereas load grammar remains constant. Overall, opening 1 or 20 files is only affected by 7%, which is negligible.

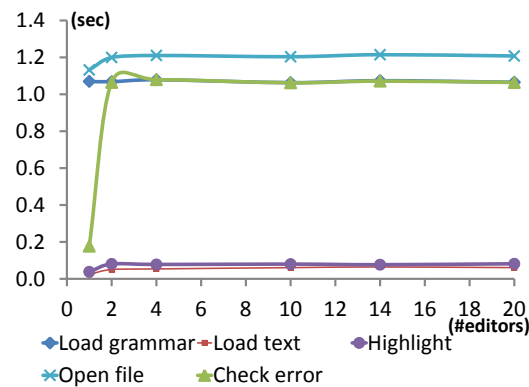


Figure 9. Effect of varying the number of concurrent editors, written in the Petri nets grammar.

Figure 10 presents the results of this experiment run on the Petri net grammar from Figure 3. The shapes of the graphs confirm that even for larger models and grammars, the number of concurrent editors does not affect performance. In this case, the most impacted feature is load text highlight (by a factor of 2), due to the different specifications of the style mapping applied on the Petri nets grammar. Check error is only increased by 20%. Overall, open file is affected by 43% for this case.

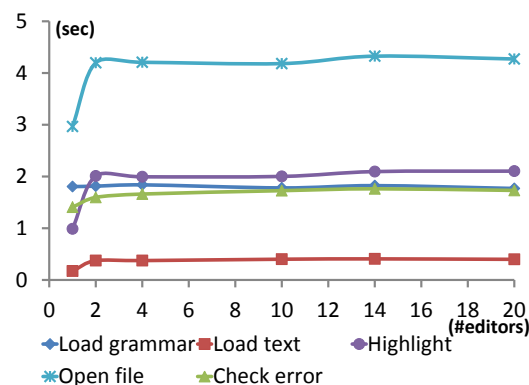


Figure 10. Effect of varying the number of concurrent editors, written in the Meta-grammar.

5.5. Discussion

It is important to note that the results of the two experiments highly rely on the implementation of our proof of concept. The goal of this prototype was not to target time efficiency but to show the feasibility of highly customizable domain-specific textual model editors. Experiment 1 has not shown impressive performance results because we tested the worst-case scenarios. As a matter of fact, as the user types in text, keywords highlighting is instantaneous. However, this experiment has demonstrated that optimizations are required for automatic text completion; for example, we can create the model in the background at regular intervals. One important consequence is that the performance of highlighting is strongly impacted by the specification of the style mapping, especially if multiple mappings must decorate many tokens. An optimization for highlighting would be to determine the

final style of each token to avoid multiple parsing and visitor passes. Another conclusion is that it is not only the amount of text that impacts features such as syntax highlighting and auto-complete but also how the style and metamodel mapping are defined.

For experiment 2, only the SCCD model needed to be adapted to support multiple editors, which shows the modularity of the architecture. This experiment lets us assert that the performance of the editors is independent of the number of editors, although this is subject to an overhead. This is thanks to the ability to dynamically instantiate as many statecharts as there are editors and execute them in parallel by means of orthogonal components. We have already started to incorporate such text editors wherever text can be edited in the graphical interface of AToMPM.

Finally, in what matters to feasibility (and the ability to tackle complexity), we conclude that it is possible to use TxtME to define complex languages. Despite using PEG expressiveness on the grammar definition, we could define a grammar (the meta-grammar) that can parse itself. Therefore, we find no obstacle in defining grammars to parse complex languages such as general programming languages (e.g., Java, C).

6. Conclusions

In this paper, we have explored the opportunities in the customizability of textual model editors. We proposed a solution for the customizability of web-based editors by fully modeling their structure and behavior, using appropriate formalisms. To demonstrate the feasibility of the approach, we have implemented a prototype, carried out various modeling tasks in architecture modeling (using AADL), and provided fully implemented case studies for performance and scalability evaluation (using Petri nets). Although our prototype is integrated with the Modelverse as a model repository, the approach is platform- and middleware-agnostic. Our experiments show that the approach demonstrates a high degree of customizability without manually altering the generated source code of the editor. This is enabled by the generative techniques making use of the modeled internals. We have observed that the increasing degree of customizability implies decreasing performance and scalability. Performance, however, was not in the scope of this study. The contributions of this paper focus on maximizing customization opportunities for textual model editors.

In future work, we plan to explore further the customization limits of our approach by automatically deriving a complete SCCD model from a given DSL grammar specification while selecting several kinds of parsing on this derivation. This will allow, for instance, to switch the parser's implementation to a different platform. Nevertheless, more importantly, this will offer tool builders more refined choices on the kind of parsing required for the application context of their textual editor (e.g., context-free, parser expression, or projectional grammars). Different features can use different parsing strategies: for instance, a context-free grammar to load an existing model, a projectional grammar for automatic text completion on each keystroke, and a PEG for syntax highlighting. We also plan to perform a thorough comparison of the different tools from a usability and performance point of view, to propose the best options to tool builders. Finally, we will re-evaluate our approach in the context of modular language engineering when constructing new complex modeling languages by reusing existing ones: in particular, mature programming languages such as Java.

Author Contributions: Conceptualization, E.S. and D.R.; methodology, E.S. and D.R.; software, D.R.; validation, D.R. and I.D.; writing—original draft preparation, E.S., D.R. and B.B.; writing—review and editing, E.S. and I.D.; visualization, E.S. and I.D.; supervision, E.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

```

grammar {
/*----- Production rules -----*/
start: grammar? mapper? @Msg 'Top level start';
grammar: GRAMMAR LCBR production_rule* RCBR
    @Msg 'Grammar definition';
production_rule: rule_definition | token_collection
    @Msg 'Top level production rule definition';
rule_definition: rule_name COLON
    rule_right_hand_side message? SEMICOL
    @Msg 'Production rule definition';
rule_name: LOWER_CASE @Msg 'Rule name';
rule_right_hand_side: token_name
    | token_value
    | rule_name
    | LPAR rule_right_hand_side RPAR cardinality?
    | rule_right_hand_side OR rule_right_hand_side
    | rule_right_hand_side rule_right_hand_side
    | rule_right_hand_side OPER
    @Msg 'Production rule right hand side';
cardinality: CARD (LSBR MINUS? INT RSBR)? @Msg 'Cardinality';
message: MESSAGE_MOD message_value @Msg 'Error message';
message_value: REGEXP @Msg 'Error message value';
token_collection: TOKENS LCBR
    (token_sub_collection | token_definition)* RCBR
    @Msg 'Top level token definition';
token_sub_collection: KEYWORDS COLON collection_name
    LCBR token_definition* RCBR
    @Msg 'Token collection definition';
collection_name: LOWER_CASE @Msg 'Token collection name';
token_definition: token_name COLON token_value
    (modifier | message)* SEMICOL
    @Msg 'Token definition';
token_name: UPPER_CASE @Msg 'Token name';
token_value: REGEXP @Msg 'Token value';
modifier: IMPLICIT_MOD | COMMENT_MOD
    @Msg 'Possible modifiers';
tokens {
    LOWER_CASE: '[a-z][a-z_0-9]*' @Msg 'Lower case characters';
    UPPER_CASE: '[A-Z][A-Z_0-9]*' @Msg 'Upper case characters';
    REGEXP: '\{, | \n\}?' @Msg 'Regular expression';
    INT: '[0-9]*' @Msg 'Integers';
    IMPLICIT_MOD: '(@Implicit | @Implied)' @Msg '@Implicit or @Implied';
    MESSAGE_MOD: '(@Message | @Msg)'
        @Msg '@Message or @Msg';
    COMMENT_MOD: '(@Comment | @Cmnt)'
        @Msg '@Comment or @Cmnt';
    keywords: general {
        TOKENS: 'tokens';
        KEYWORDS: 'keywords';
        GRAMMAR: 'grammar';
        MAPPER: 'mapper';
    }
    OPER: '[?+*]' @Msg 'operator: ?, * or +'; OR: '|' @Msg '|';
    LPAR: '(' @Msg '('; RPAR: ')' @Msg ')';
    MINUS: '-' @Msg '-'; CARD: '#' @Msg '#'; SEMICOL: ';' @Msg ';'; COLON: '::' @Msg '::';
    LCBR: '{' @Msg '{'; RCBR: '}' @Msg '}';
    LSBR: '[' @Msg '['; RSBR: ']' @Msg ']';
    // Implicit declarations
    NEWLINE: '\r?\n[\t]*' @Msg 'New Line' @Implicit;
    WS: '\t | \f | \r | \n' @Msg 'White space' @Implicit;
    LINE_CONT: '\t | \f | \r | \n' @Implicit @Msg 'Line continuation';
    SINGLE_COMMENT: '/*[^\n]*'
        @Comment @Msg 'Single line comment';
    MULTI_COMMENT: '/*[^\n]*\n*/'
        @Comment @Msg 'Multi line comment';
}
}

/*----- Model mapping -----*/
mapper: MAPPER LCBR model_mapping_rule RCBR
    @Msg 'Top level model mapper definition';
model_mapping_rule: prod_name ARROW MODEL
    concept_name LCBR mapping_rules* RCBR
    @Msg 'Model mapper definition';
mapping_rules: attr_mapping_rule
    | class_mapping_rule
    | assoc_mapping_rule
    @Msg 'Top level mapping rules';
attr_mapping_rule: prod_name ARROW ATTR
    concept_name SEMICOL @Msg 'Attribute mapping rule';
ref_mapping_rule: prod_name ARROW REF
    concept_name SEMICOL @Msg 'Reference mapping rule';
class_mapping_rule: prod_name ARROW CLASS
    concept_name LCBR attr_mapping_rule* RCBR
    @Msg 'Class mapping rule';
assoc_mapping_rule: prod_name ARROW ASSOC
    concept_name LCBR (attr_mapping_rule|ref_mapping_rule)*
    RCBR @Msg 'Association mapping rule';
prod_name: path @Msg 'Name of production rule being mapped';
concept_name: path @Msg 'Name of concept being mapped to';
path: IDENTIFIER (DOT IDENTIFIER)* @Msg 'Dotted name';
tokens {
    // mapper tokens
    IDENTIFIER: '[a-zA-Z][a-zA-Z_0-9]*' @Msg 'Identifier';
    DOT: '.' @Msg '.'; ARROW: '>' @Msg '>';
    ASSOC: '@Assoc' @Msg 'Association mapper symbol';
    ATTR: '@Attr' @Msg 'Attribute mapper symbol';
    CLASS: '@Class' @Msg 'Class mapper symbol';
    MODEL: '@Model' @Msg 'Model mapper symbol';
    REF: '@Ref' @Msg 'Reference mapper symbol';
}
}

```

Figure A1. The Meta-Grammar Definition.

References

- Voelter, M. Language and IDE Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV*; LNCS; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7680, pp. 383–430.
- Hutchinson, J.; Whittle, J.; Rouncefield, M.; Kristoffersen, S. Empirical assessment of MDE in industry. In *Proceedings of the International Conference on Software Engineering*, Honolulu, HI, USA, 21–28 May 2011; pp. 471–480.
- Akdur, D.; Garousi, V.; Demirörs, O. A survey on modeling and model-driven engineering practices in the embedded software industry. *J. Syst. Archit.* **2018**, *91*, 62–82. [CrossRef]
- Friedenthal, S.; Moore, A.; Steiner, R. *A Practical Guide to SysML: The Systems Modeling Language*, 3rd ed.; Morgan Kaufmann: Burlington, MA, USA, 2014.
- Feiler, P.H.; Gluch, D.P.; Hudak, J.J. *The Architecture Analysis & Design Language (AADL): An Introduction*; Technical Report; Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst: Pittsburgh, PA, USA, 2006.
- Eysholdt, M.; Behrens, H. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the Companion on Object Oriented Programming Systems Languages and Applications (SPLASH '10)*, New York, NY, USA, 17–21 October 2010; pp. 307–309.
- Heidenreich, F.; Johannes, J.; Karol, S.; Seifert, M.; Wende, C. Model-Based Language Engineering with EMFText. In *Generative and Transformational Techniques in Software Engineering IV*; LNCS; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7680, pp. 322–345.

8. Kats, L.C.; Visser, E. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In Proceedings of the Object Oriented Programming Systems Languages and Applications (OOPSLA'10), Reno/Tahoe, NV, USA, 17–21 October 2010; pp. 444–463.
9. Behrmann, G.; David, A.; Larsen, K.G. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*; Number 3185 in LNCS; Springer: Berlin/Heidelberg, Germany, 2004; pp. 200–236.
10. Krahn, H.; Rumpe, B.; Völkel, S. MontiCore: Modular Development of Textual Domain Specific Languages. In *Objects, Components, Models and Patterns*; LNBP; Springer: Berlin/Heidelberg, Germany, 2008; Volume 11, pp. 297–315.
11. Lafontant, L.E.; Syriani, E. Gentleman: A Light-Weight Web-Based Projectional Editor Generator. In Proceedings of the Model Driven Engineering Languages and Systems: Companion Proceedings, Montreal, Canada; ACM, Virtual, 18–23 October 2020; pp. 1–5. [\[CrossRef\]](#)
12. Van Mierlo, S.; Van Tendeloo, Y.; Meyers, B.; Exelmans, J.; Vangheluwe, H. SCCD: SCXML Extended with Class Diagrams. In Proceedings of the Workshop on Engineering Interactive Systems with SCXML, Brussels, Belgium, 21–24 June 2016; pp. 2:1–2:6.
13. Beard, J. Developing Rich, Web-Based User Interfaces with the Statecharts Interpretation and Optimization Engine. Masters Thesis, McGill University, Montreal, QC, Canada, 2013.
14. Sousa, V.; Syriani, E.; Fall, K. Operationalizing the Integration of User Interaction Specifications in the Synthesis of Modeling Editors. In *Software Language Engineering*; ACM: Athens, Greece, 2019; pp. 42–54. [\[CrossRef\]](#)
15. Kolovos, D.S.; Paige, R.F.; Polack, F.A. The epsilon transformation language. In *Proceedings of the International Conference on Theory and Practice of Model Transformations*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 46–60.
16. Biehl, M.; El-Khoury, J.; Loiret, F.; Törngren, M. On the modeling and generation of service-oriented tool chains. *Softw. Syst. Model.* **2014**, *13*, 461–480. [\[CrossRef\]](#)
17. Addazi, L.; Ciccozzi, F.; Langer, P.; Posse, E. Towards seamless hybrid graphical–textual modelling for uml and profiles. In Proceedings of the European Conference on Modelling Foundations and Applications, Marburg, Germany, 19–20 July 2017; pp. 20–33.
18. Pech, V.; Shatalin, A.; Voelter, M. JetBrains MPS as a tool for extending Java. In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, 11–13 September 2013; pp. 165–168.
19. De Jonghe, G. *Statecharts and Class Diagram XML—A General-Purpose Textual Modelling Formalism*; Tech Report; University of Antwerp: Antwerpen, Belgium, 2014.
20. Riegelhaupt, D. Web-Based Text Editor for Writing Research. Bachelor's Thesis, University of Antwerp: Antwerpen, Belgium, 2011.
21. Van Tendeloo, Y.; Vangheluwe, H. The Modelverse: A tool for multi-paradigm modelling and simulation. In Proceedings of the 2017 Winter Simulation Conference (WSC), Las Vegas, NV, USA, 3–6 December 2017; pp. 944–955.
22. Syriani, E.; Vangheluwe, H.; Mannadiar, R.; Hansen, C.; Van Mierlo, S.; Ergin, H. AToMPM: A Web-based Modeling Environment. In Proceedings of the Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition Co-Located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, FL, USA, 29 September–4 October 2013; pp. 21–25.
23. Benellallam, A.; Gómez, A.; Sunyé, G.; Tisi, M.; Launay, D. Neo4EMF, A Scalable Persistence Layer for EMF Models. In *Modelling Foundations and Applications*; LNCS; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8569, pp. 230–241.
24. Feiler, P.H.; Gluch, D.P. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*; Addison-Wesley: Boston, MA, USA, 2012.
25. Ford, B. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. *SIGPLAN Not.* **2002**, *37*, 36–47. [\[CrossRef\]](#)
26. Grimm, R. Better Extensibility Through Modular Syntax. *SIGPLAN Not.* **2006**, *41*, 38–51. [\[CrossRef\]](#)
27. Berthomieu, B.; Bodeveix, J.P.; Chaudet, C.; Dal Zilio, S.; Filali, M.; Vernadat, F. Formal verification of AADL specifications in the Topcased environment. In Proceedings of the International Conference on Reliable Software Technologies, Brest, France, 8–12 June 2009; Springer: Berlin/Heidelberg, Germany, 2009; pp. 207–221.
28. Rugina, A.E.; Kanoun, K.; Kaâniche, M. A system dependability modeling framework using AADL and GSPNs. In *Architecting Dependable Systems IV*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 14–38.
29. Renault, X.; Kordon, F.; Hugues, J. From AADL architectural models to Petri Nets: Checking model viability. In Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, Tokyo, Japan, 17–20 March 2009; pp. 313–320.
30. Monteverde, D.; Olivero, A.; Yovine, S.; Braberman, V. VTS-based specification and verification of behavioral properties of AADL models. In *Proceedings of the International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES'08)*; Springer: Berlin/Heidelberg, Germany, 2008.