Derivation and Refinement of Textual Syntax for Models

Florian Heidenreich, Jendrik Johannes, Sven Karol,
Mirko Seifert, and Christian Wende

Institut für Software- und Multimediatechnik
Technische Universität Dresden
D-01062, Dresden, Germany
{florian.heidenreich,jendrik.johannes,sven.karol,
mirko.seifert,c.wende}@tu-dresden.de

Abstract. Textual Syntax (TS) as a form of model representation has made its way to the Model-Driven Software Development community and is considered a viable alternative to graphical representations. To support the design and implementation of text editing facilities many concrete syntax and model mapping tools have emerged. Despite the maturity of these tools, users still spend considerable effort to specify syntaxes and generate editors even for simple metamodels. To reduce this effort, we propose to refine a specification that is automatically derived from a given metamodel. We argue that defaults in a customisable setting enable developers to quickly realise text-based editors for models. In particular in settings where metamodels evolve, such a procedure is beneficial. To evaluate this idea we present EMFText [1], an EMF/Eclipse integrated tool for agile TS development. We show how default syntax can easily be tailored and refined to obtain a custom text editor for EMF models and demonstrate our approach by two examples.

1 Introduction

Formal languages are the basis for most activities in computer science (e.g., to describe data or to specify behaviour). They consist of a syntax and a semantics, where the former describes the layout of valid sentences and the latter assigns meaning. Well designed languages usually have a syntax that can be easily understood and that supports the languages' semantics.

For a long time, syntaxes were textual, but during the last two decades graphical representations have become very popular. In particular Model-Driven Software Development has pushed the widespread use of graphical methods to represent (i.e., to model) software. Both types of syntaxes have their pros and cons. Graphical syntaxes are strong in showing relationships, quantities and provide the opportunity to zoom, which is useful to obtain an overview of the presented sentence. In contrast, textual syntaxes have a predefined reading order, which is valuable if sentences are interpreted in sequential order. Furthermore, in model environments that lack a central version controlled model repository textual representations can be easily compared and merged. Thus, graphical and textual syntax must not be considered alternatives, but rather complements. While each one can perfectly serve one purpose it might be inadequate for another one.

Textual syntax can be either generic or custom. Again, both have their pros and cons. Generic syntaxes are defined on the level of metamodelling languages. Thus, they are either instantly available or can be derived automatically for concrete modelling languages. Custom syntaxes need manual specification effort. The former are also often more verbose and do not reflect the semantics of a language nicely, in contrast to the latter, which are more compact and use symbols that ease reading. To exemplify the difference, consider the two definitions of a UML Transition shown in Listing 1.1.

```
1 Transition { source : State "StateA" target : State "StateB" }       // generic syntax
2 StateA -> StateB                                                      // custom syntax
```

Listing 1.1. Two concrete syntaxes for a UML Transition

Clearly the customised syntax in Line 2 is easier to read, but also requires additional specification effort. If users have metamodels for their languages and want to use textual syntax for their models, the question is, how the benefits of generic syntaxes (low effort) and those of customised syntaxes (easier readability, reflection of semantics) can be combined. This paper presents an approach to achieve this and therefore allows rapid and agile development of tooling for TS. By deriving a default syntax, which can optionally be refined, our tool EMFText aims to reduce the effort needed to develop tool support for TS. As we will show in this paper, not only the initial specification of syntaxes, but also the adaptation of syntaxes to evolving metamodels, can profit from the approach. With this we follow a new direction in TS development, as opposed to previos approaches which we discuss in Section 5, which decreases the effort of syntax development in particular in the initial development phase.

The remainder of this paper is organised as follows: Section 2 describes how models can be represented as text. In particular, we discuss the options that exist both for deriving a default syntax as well as customising it later on. Section 3 provides detailed information about EMFText—the tool that implements our approach. Equipped with this, we present the concrete syntax definition for two modelling languages in Section 4, revealing the strengths and drawbacks. Section 5 compares the presented work with related approaches and Section 6 concludes this paper.

2 Approach

In this section, we first study how concepts used in metamodelling map to concepts used in grammar engineering. Based on this knowledge, we explore how default values for a concrete TS can be automatically derived for a concrete modelling language (i.e., a concrete metamodel). Then we discuss how the derived TS can be further refined and tailored if needed. As one example for the derivation of default syntaxes, we will use the Human-Usable Textual Notation

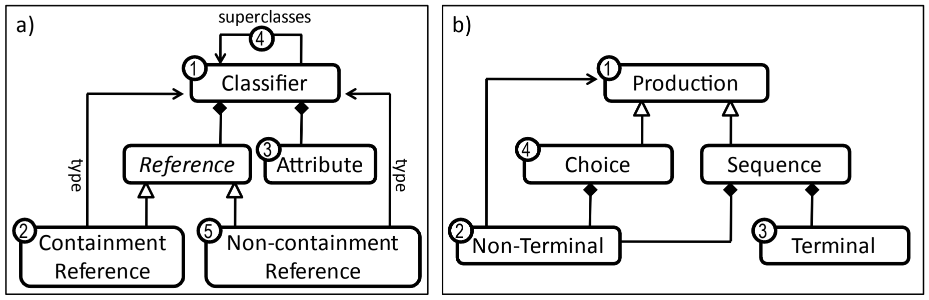


Fig. 1. Concepts of modelling languages (a) and text languages (b).

(HUTN) [2]—which is one but not the only possible way of deriving an initial syntax[1].

2.1 Mapping Modelling Concepts to Text Language Concepts

Before we can derive and refine a TS for a concrete modelling language, we must recapitulate which concepts are available in textual syntaxes and how they relate to the concepts used in modelling languages. Typically, textual languages are defined by context-free grammars and modelling languages are defined by metamodels. Hence, a mapping between metamodelling and grammar engineering concepts must be established. We show these concepts side by side in Fig. 1. In the following we first summarise the core concepts of metamodelling (based on [3]) and of grammar engineering (based on [4, 5]). Afterwards (Sect. 2.2), we discuss the mappings between the concepts. This discussion is based on [6].

Models defined in modelling languages are composed of elements that in turn consist of attribute values and other contained elements. In addition, cross-references between elements can exist. Which elements are allowed is defined in a metamodel through classifiers (1a). By defining containment references between classifiers, the possible containment relations are defined (2a). Which attribute values can be defined for an element is declared by attributes in the metamodel (3a). Classifiers are also connected through superclass relationships, which expresses exchangeability (4a). Possible cross-references are defined by non-containment references (5a).

Sentences written in a textual languages consist of a sequence of elements, where an element is either represented by a single symbol (i.e., a set of connected characters) or another nested sequence, which again consists of symbols and possibly other nested sequences. The allowed parts of textual languages are defined in a grammar through productions (1b). By referencing other productions through non-terminals in a sequence, the possible nesting of elements is defined (2b). The symbols that may appear in a sequence are defined by terminals (3b). Choices of different non-terminals can be defined to express exchangeability (4b).

[1] see Line 1 in Listing 1.1 for an example of HUTN syntax

From these core concepts of both language engineering approaches, the following mapping can be derived. (1a–1b) Element types are defined through classifiers on the one side and by productions on the other. (2a–2b) The composition of elements from others is expressed by containment references on the metamodel side. Non-terminals that appear in a sequence express a similar relationship on the grammar side. (3a–3b) Attribute values are also part of a model element and can therefore be expressed by symbols. Consequently the attribute concept is mapped to the terminal concept. (4a–4b) The superclass relationship can be mapped to the alternative concept because both express exchangeability. (5a–3b) For cross-references, there exists no direct correspondence in text. They can however be mapped to symbols as well. In this case, the symbol would represent an identifier that identifies the element to be referenced.

2.2 Derivation and Refinement

After discussing how modelling languages can be mapped to textual syntaxes in general, the question is, what this mapping looks like for concrete modelling languages. Our design objective for such a mapping is to enable rapid and compact text syntax definition. Keeping this in mind, we want to automatically derive a default text syntax and then allow the developer to refine the generated syntax in an intuitive and incremental manner.

Deriving a default text syntax can be performed using a) the language's metamodel and b) assumptions based on standards and best-practices. Hence, these are the two parameters for this derivation process. We will shortly see where one or the other is used to obtain parts of the default syntax. Following the argumentation from Sect. 2.1 the derivation of syntaxes can be studied starting from the elements of modelling languages. For example, each metaclass must be mapped to a production. Thus, we will examine what such concrete default mappings can look like. Once an initial mapping is obtained, it can be refined. Since the options for the design of the default mapping and the customisation are similar, we will discuss both together for each modelling concept.

Classifiers define the types of model elements that can be used in a model. A representation of model elements as text, must disambiguate which class an element belongs to. The most basic way to do so is to use a distinct terminal symbol (keyword) for each classifier. HUTN realises this idea and uses the names of the classifiers from the metamodel as keywords. In subsequent customisation steps these default keywords can be changed. If classifiers can be disambiguated based on other properties of their text syntax, the keywords can be removed during the refinement process.

A *Containment* of model elements is expressed by *nesting* sequences. This is achieved using non-terminals in sequences. These define the children (elements in containment references) of the element defined by the parent sequence. Since the containment relation between metaclasses is fully defined by the metamodel the default nesting can therefore be directly derived. As the tree structures of the metamodel and the grammar must match, the nesting is dictated by the containment relation. However, the order in which contained elements are given in TS can be refined. HUTN allows an arbitrary order, which can be restricted by further customisation steps.

Attributes are typed properties of classifiers. Classifiers can have multiple attributes. Consequently, the textual representation of an attribute must identify both the attribute itself (e.g., using its name) and its value, as well as, the classifier instance the attribute belongs to. A default approach to have this three things available is to a) define attributes in the context of their classifier (e.g., directly after the keyword), and b) state the name of the attribute in the text together with its value. Thus, two terminals are sufficient to represent an attribute and its value. HUTN adds a third one (a double colon) to ease reading for humans. The default definition for the terminals used for attibute values can be obtained using the type of the attribute. For primitive types this mapping is straightforward. Numbers are mapped to text using their arabic representation, boolean values are `true` and `false`, strings are enclosed in double quotes and characters are surrounded by single quotes. Refining this mapping can be done in different ways. The tokens for the attribute names can be omitted if an order is defined upon the set of attributes or if all attributes have distinct types. Values can be mapped differently (e.g., boolean values can be represented as `yes` and `no`).

The *Inheritance* hierarchy of the metamodel defines that different types of elements are allowed at certain positions. Consequently, *Alternatives* of sequences are allowed at positions where different types of model elements (with a common supertype) may occur. The alternatives can be derived from the inheritance hierarchy and can not be altered (without changing the metamodel).

Non-containment references connect model elements and establish a graph structure. To represent such references in text, names (or identifiers) must be used. To obtain unambiguous identifiers an attribute having some uniqueness constraint (e.g., *name* or *id*) can be used. A default setting is thus to use the values of such attributes to reference model elements. However, sometimes it is not clear which attribute must be used or scoping rules apply. In this case the handling of names must be refined. To do so, both the identification of elements and the referencing rules (e.g., scoping) must be defined.

To summarise, a complete instance of a *Classifier* (i.e., a complete model element with attributes and references) is represented by a *Sequence* of terminal symbols (e.g., keywords and identifiers) and non-terminals. In the particular case of HUTN, an initial sequence definition for a metaclass starts with a keyword that equals the name of the metaclass name followed by terminal symbols and names for the element's attributes and references in an arbitrary order.

3 Overview of EMFText

To first derive and later refine a TS, developers require a tool that 1) performs the derivation, 2) provides the facilities to refine the derived syntax, 3) supports the evolution of the TS along with the metamodel it is based on and 4) generates the tooling to create and manipulate models using the TS. Existing TS tools lack

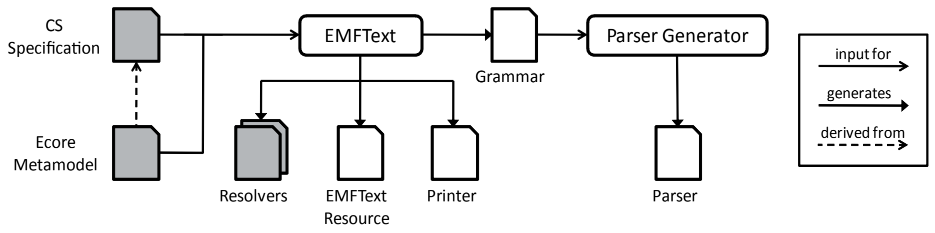


Fig. 2. Overview of Specified and Generated Artefacts

support for one or more of these requirements (see Sect. 5). Therefore, we implemented EMFText that supports all the four functionalities in combination. With EMFText we evaluate the theoretical analysis of Sect. 2 on different examples in Sect. 4. This section therefore introduces EMFText, and how it supports the four requirements (cf. Fig. 2).

To provide the refinement capabilities discussed in Sect. 2.2, EMFText provides a specification language called *CS*. An initial CS definition is therefore the result of the derivation performed by EMFText (upper left of Fig. 2). Consequently, an initial CS is obtained without any effort (as we will see in Sect. 4). In CS specifications, the syntax for model elements is defined using concepts from the Extended Backus Naur Form (EBNF). This is, because these concepts from grammar engineering are well-known and are therefore a practical tool for manual refinements. For refinements that go beyond grammar engineering as the mapping of identifiers to cross-references (Sect. 2.1), EMFText generates a Java API. Here a specification of how identifiers are resolved to the respective elements can be implemented (lower left of Fig. 2). Again, to minimise specification effort, a default implementation is provided by EMFText, where identifiers are resolved to elements of the correct type having the correct name or id.

When metamodels evolve over time, the syntax might become incomplete or invalid. In the first case, the textual syntax is incrementally derived for new metamodel elements and syntax of removed metamodel elements is deleted. Developers do not need to spend time to adapt their syntaxes to the changes made to metamodels in these cases. However, changes in existing metaclasses with a refined textual syntax cannot automatically be merged into an existing CS specification. In such situations EMFText informs the developer of necessary syntax refinements by carefully analysing the CS specification. This analysis process, which also ensures correctness during refinement, checks that the CS definition matches the metamodel, by checking that 1) each production (rule) corresponds to a concrete metaclass, 2) that there is a rule for each concrete metaclass and 3) all references defined in the metamodel are present in the syntax. Since CS is itself defined with Ecore and EMFText, all these checks are performed on a CS model (i.e., an instance of the CS metamodel).

Technically, EMFText is implemented as a set of plug-ins for the Eclipse platform [7], tightly integrated with the Eclipse Modelling Framework (EMF) [3]. In addition to the mentioned resolvers, several artefacts are generated by EMF-

Text based on a CS to provide runtime tooling (right side of Fig. 2). First, a parser is derived that can read text syntax conforming to the CS productions and that creates a model. Second, the counterpart—a printer—that transforms models to text is derived from the same specification. Third, an implementation of the *Resource* interface of EMF is generated through which parser and printer are plugged into EMF's resource management (making them accessible to all EMF-based modelling tools). For the parser generation, EMFText is generally not bound to a specific technology (i.e., additional technologies can be plugged in). However, currently we support only ANTLR [8]. We mention this, because the concrete parsing approach used can influence the CS specification. Therefore, the CS analysis mechanism described above can be enriched with checks specific to the used parser technology. For instance, ANTLR cannot handle left recursion. Thus, a mechanism for automatically resolving direct left recursive rules was provided. Other left recursive rules are detected on the CS level and users are encouraged to refactor their syntax definitions. If a different parser generator is used, the detection of left recursive rules might become obsolete, but other analysis might be needed to make sure the CS meets the criteria of the respective parser generator.

4 Examples

The previous sections introduced the conceptual and technological foundations for development of concrete syntax for modelling languages with EMFText. Here, we exemplify development with EMFText by two well-established modelling languages—feature models and UML state machines—and show how an automatically derived TS can be tailored by several refinement steps. More examples can be found at [9].

4.1 Parsing and Printing of Concrete Syntax for Feature Models

Variability modelling is at the heart of product-line engineering [10]. One widely used notation to express variability between different products in a product line are feature models [11]. The feature metamodel depicted in Fig. 3 was taken from the FeatureMapper [12], a tool for mapping features to models. A central requirement for the development of the FeatureMapper was its compatibility with different existing tools. Thus, it uses a simple and flexible abstraction as feature metamodel which covers a broad range of existing feature model derivatives.

Feature models describe hierarchical tree structures between features and groups of features. These models are usually created using a diagrammatic notation, but as these models can grow for large product lines, an alternative textual representation seems to be beneficial. Hence, providing adequate means for both parsing and printing nested language elements are requirements for our concrete syntax specification.

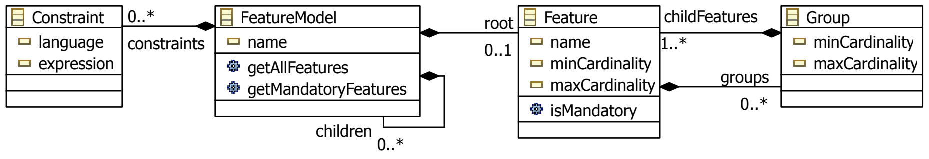


Fig. 3. Metamodel for Feature Models

Generating a Default Concrete Syntax The starting point in the development of a TS for feature models is the concrete syntax specification language *CS* of EMFText. As discussed in Section 3, CS specifications are tightly coupled to one ore more EMF-based metamodels whose structure implicitly defines a grammar skeleton. For every non-abstract metaclass a production rule can be specified. Hence, a production's left-hand side does not only correspond to a non-terminal in a context-free grammar but also to a metamodel element with the same name. Elements on the right-hand side of the production are mapped to features (i.e., attributes or references) of the metaclass. To kick-start the development of a new concrete syntax, EMFText generates a default syntax for every non abstract metaclass based on the conventions introduced by the HUTN standard. HUTN aims at providing a generic, human-readable textual syntax for all kinds of MOF-based metamodels. For this purpose, it derives the syntactical representations of model elements from the properties and relationships of the corresponding metaclass.

An example using the specified syntax for the initialisation of a concrete Feature with the name "SimpleFeature" is shown in Listing 1.2. Every Feature declaration starts with the keyword "Feature". Encapsulated in curly braces initialisations for all properties and references of the Feature in arbitrary order can be given. Every single property initialisation is preluded by a keyword corresponding to the name of the property and a colon.

```
Feature { name : "SimpleFeature" }
```

Listing 1.2. Feature declaration using the HUTN syntax

Concrete Syntax Tailoring We argue that HUTN is not acceptable for a broad audience since it neither provides syntax tailored to a language's application domain nor reflects language-specific semantics. Thus, we propose an optional but recommended tailoring of the generated grammar productions to construct a more usable syntax. The code listing in Fig. 4 compares the initially generated HUTN-based syntax specification with the adapted feature model syntax after tailoring. Lines that have not changed during syntax tailoring span the full width of the listing. Differing lines are shown side by side.

Line 1 assigns a name to the concrete syntax, which can later be used as the file name extension for model instances (here `featuremodel`), refers to a Ecore metamodel registered in the system under the given URI, and defines which

```
 1  SYNTAXDEF featuremodel FOR <http://www.tudresden.de/feature> START FeatureModel
 2  RULES {
 3  FeatureModel ::= "FeatureModel"
 4    "{" ( "constraints"  ":" constraints |         name['"','"']
 5    "root"  ":" root |                              ( "{" "constraints"  ( constraints ";")? "}")?
 6    "name"  ":" name['"','"'] )* "}" ;              root;
 7  Feature ::= "Feature"
 8    "{" ( "name"  ":" name['"','"'] |               name['"','"']
 9    "minCardinality"  ":" minCardinality[INTEGER] | ("(" minCardinality[INTEGER] ".."
10    "maxCardinality"  ":" maxCardinality[INTEGER] | maxCardinality[INTEGER] ")")?
11    "groups"  ":" groups )* "}" ;                   ( groups* )?  ;
12  Group ::= "Group"
13    "{" ( "minCardinality"  ":" minCardinality[INTEGER] |   ("(" minCardinality[INTEGER] ".."
14    "maxCardinality"  ":" maxCardinality[INTEGER] |         maxCardinality[INTEGER] ")" )?
15    "childFeatures"  ":" childFeatures  )* "}" ;            ("{" childFeatures* "}")?;
16  Constraint ::= "Constraint"
17    "{" ( "language"  ":" language['"','"'] |       language[] ":"  expression['"','"'];
18    "expression"  ":" expression['"','"']  )* "}" ;
19  }
                 HUTN-based Syntax                            Tailored Syntax
```

Fig. 4. CS specification for Feature Models

metaclass is used as the root element of a model and the start symbol for the parser.

The rules section (Lines 2–19) contains one production rule for each concrete metaclass. On the right-hand side of each rule we find 1. containment references (e.g., `childFeatures` in Line 15) that define the positions of nestings, 2. attributes (e.g., `name` in Line 6) that define the position of attribute values, and 3. keywords (e.g., ”`FeatureModel`” in Line 3) that are pure concrete syntax and have no counterpart in the metaclass.

For attributes, which in contrast to references are followed by square brackets, the allowed symbols as well as the mapping from such a symbol to a value in the model can be customised. The first can be performed by using predefined (e.g., `INTEGER` for `minCardinality` in Line 9) or custom defined token types (can be defined with regular expressions in an extra section). A standard type that allows a sequence of all letters and numbers is used by default if nothing is specified (e.g., `name` in Line 6). The mapping can be adjusted by specifying a prefix and a suffix instead of a token type (e.g., ” and ” for `name` in Line 4). This causes EMFText to remove the prefix and suffix during parsing and to add them during printing such that they never occur in the model.

The comparison of the generated and the tailored syntax shows that many parts of the generated specification were reused or only slightly changed. In some productions unneeded keywords were deleted, a few productions were changed more invasively. In summary, the automatically generated syntax based on HUTN provided guidance for refinement and tailoring of the syntax specification.

From the tailored CS specification, EMFText derives a context-free grammar which is exported as an ANTLR grammar specification. In accordance to the inheritance structure in the metamodel alternatives are derived to express the interchangeability of a type’s subtypes. The ANTLR specification is also annotated with semantic actions to instantiate the model-representation for the

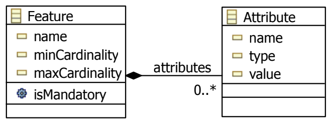


Fig. 5. Extension of Metamodel for Feature Models

parsed expressions. Finally the ANTLR tooling is used to generate the implementation of the lexer and parser. The LOC-ratio between a CS-Specification and the resulting ANTLR grammar is about 1 : 10.

As a second implementation artefact, a pretty printer is generated from the syntax specification. This printer transforms a given model to its textual representation. Often special layout conventions which do not interfere with language syntax and semantics are used to provide a more readable and comprehensive syntax. Two constructs can be used in CS specifications to incorporate these layout conventions. First, the `LineBreak` element (`!<n>`) causes a line break during printing and an indentation in the next line to a level of `<n>`. Seconds, the `Whitespace` elements ♯`<n>`) inserts `<n>` whitespaces during printing. Using these constructs is optional.

Evolving the Concrete Syntax During the development of the FeatureMapper we faced several situations where the feature metamodel had to be adapted. Such an evolution is a common issue in language development. However changes in a languages metamodel necessitate the co-adaptation of its textual syntax. A concrete example for the feature metamodel was the introduction of `Attribute`s for `Feature`s (cf. Fig. 5).

To adapt the tailored textual syntax for feature models in accordance, the CS specification is automatically extended by EMFText with an additional HUTN-based syntax rule for `Attribute`s. In addition, the syntax rule in Line 3–6 of Figure 4 has to be extended to include `Attribute`s in `Feature`s. We decided not to automate the inclusion of a corresponding containment reference, since the rule was already tailored. Instead, EMFText adds a warning to the CS specification to inform the developer of the necessary syntax refinement. This exemplifies how derivation and stepwise refinement helps the co-adaptation of TS for evolving languages.

4.2 Resolving References in UML State Machines

Our second example defines a textual syntax for UML State Machines. It uses on an existing UML metamodel implementation that can be found at [13]. Listing 1.3 specifies a text syntax for state machines and Fig. 6 shows an exemplary state machine opened in the EMF text editor. Note that the metamodel defines `Vertex` as a superclass of `State`, `FinalState` and the initial state (represented as `Pseudostate` if kind initial). Thus, their productions can be alternatively used for the non-terminal `subvertex` in Line 5. Since the syntax defined in Listing 1.3

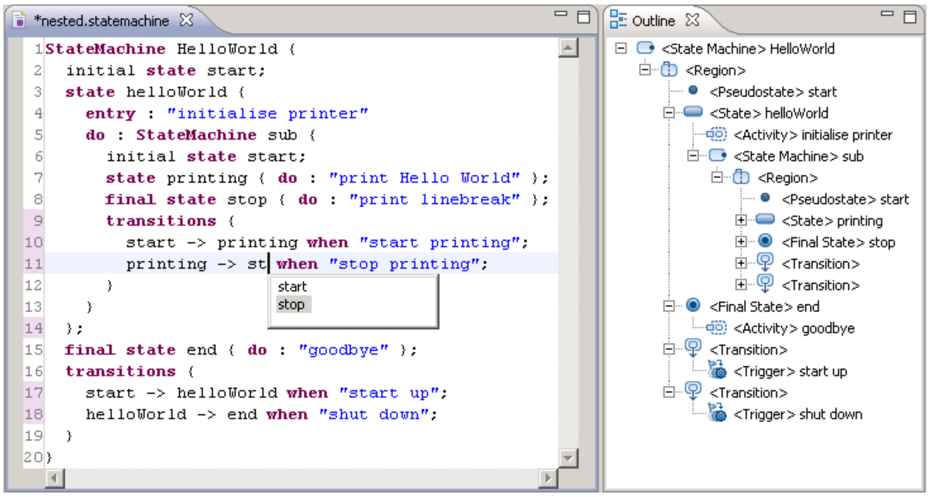
```
 1 SYNTAXDEF statemachine FOR <http://www.eclipse.org/uml2/2.1.0/UML>
 2 START StateMachine
 3 RULES {
 4   StateMachine  ::= "StateMachine" name[] "{" region "}" ;
 5   Region        ::= subvertex* "transitions" "{" transition* "}";
 6   State         ::= "state" name[] "{" ("entry" ":" entry)? ("exit" ":" exit)?
 7                     "do" ":" doActivity "}" ";";
 8   Pseudostate   ::=  kind[] "state" name[] ";";
 9   FinalState    ::= "final" "state" name[] "{" ( "entry" ":" entry )?
10                     ( "exit" ":" exit )? "do" ":" doActivity "}" ";";
11   Transition    ::= source[] "->" target[] "when" trigger
12                     ( "do" ":" effect )? ";";
13   Trigger       ::= name['"','"'];
14   Activity      ::= name['"','"'];
15 }
```

Listing 1.3. Concrete syntax for UML state machines

concentrates on state machines, only the `name` of an `Activity` (Line 14) can be specified (and no further details).

In contrast to the metamodel for feature models—which defined a strict tree structure, by only utilising *containment* references (represented by filled diamonds in Fig. 3)—the UML metamodel also utilises *non-containment* references. Non-containment references are used when an element should point to an element that is defined elsewhere.

In a CS specification, terminals can be defined for non-containment references in the same way they are defined for attributes. A token parsed for a terminal represents the *name* of the referenced element. These names have to be resolved to the actual elements. In the UML state machine syntax, the non-containment references `source` and `target` of `Transition` (Listing 1.3, Line 11) are used.



```
StateMachine HelloWorld {
  initial state start;
  state helloWorld {
    entry : "initialise printer"
    do : StateMachine sub {
      initial state start;
      state printing { do : "print Hello World" };
      final state stop { do : "print linebreak" };
      transitions {
        start -> printing when "start printing";
        printing -> st when "stop printing";
      }                start
    }                  stop
  };
  final state end { do : "goodbye" };
  transitions {
    start -> helloWorld when "start up";
    helloWorld -> end when "shut down";
  }
}
```

```
<State Machine> HelloWorld
  <Region>
    <Pseudostate> start
    <State> helloWorld
      <Activity> initialise printer
      <State Machine> sub
        <Region>
          <Pseudostate> start
          <State> printing
          <Final State> stop
          <Transition>
          <Transition>
    <Final State> end
      <Activity> goodbye
    <Transition>
      <Trigger> start up
    <Transition>
      <Trigger> shut down
```

Fig. 6. A UML state machine edited using it's concrete syntax in the EMFText editor

Default Reference Resolving EMFText provides a default algorithm to resolve references. It uses type information provided by the metamodel to find candidates in the parse tree (e.g., the `target` of a `Transition` must be a `State`). Further, it compares the name of each candidate with the terminal representing the reference to find a match. For that the candidates need to provide a `name` attribute. For every match found, the terminal is replaced with a direct reference to the found element (a `State` in this example). During pretty printing, the references are de-resolved. That is, the referenced object is replaced by a terminal constructed from the `name` attribute.

Tailoring Reference Resolving By default, a `name` attribute is expected to identify the referenced element unambiguously. For models defined in a large language like UML, this convention does not always apply. For instance, the `doActivity` containment reference of `State` is of type `Behaviour` (Listing 1.3, Line 10). `Activity` and `State` are two subclasses of `Behaviour`. Thus, the UML metamodel for state machines allows to specify the behaviour of a state by embedding a subordinated state machine. In the state machine in Figure 6 for example, there are two `States` named *start* (Lines 2 and 7). UML specific scoping rules apply for the visibility of `States` to avoid name clashes between nested machines. Naturally, the default resolving does not know these rules and therefore requires customisation.

For this purpose, EMFText provides the means to flexibly adapt the semantics for matching a name with potential reference candidates. For every non-containment reference a `ReferenceResolver` with two template methods, `doResolve()` for resolving and `doDeResolve()` for de-resolving, is generated. By default, the methods call their super implementation which implements the default reference resolving behaviour defined above. Hence, a developer can extend or override the default behaviour in this methods. In successive generation steps, the already generated and adapted resolvers are not overwritten.

In the case of our simplified UML state machines two references need to be resolved. Thus, two resolvers: `TransitionSourceReferenceResolver` and `TransitionTargetReferenceResolver` are generated by EMFText. Both can be implemented in a uniform manner. Listing 1.4 shows the implementation of the `doResolve()` method for the latter. Every ReferenceResolver extends the generic type `AbstractReferenceResolver<T>` which gets bound with the type the resolver is meant to resolve (in this case `Transition`). The parameters for `doResolve()` include the `name` to be resolved, the `container` of the unresolved reference (in this case `Transition`), the boolean parameter `resolveFuzzy`, and a `ResolveResult` into which results of the resolution as well as error or warning messages can be placed. In Lines 5–15 only `Vertices` that are defined within the same `Region` as the current `Transition` are considered. This realises the scoping.

If the `resolveFuzzy` flag is turned on, the proxy resolver is asked for all elements that might match the given name. This is used by the text editor to collect suggestions for code completion. Figure 6 shows the completion propos-

```
 1 public class TransitionTargetReferenceResolver extends AbstractReferenceResolver<Transition> {
 2   ...
 3   protected void doResolve(String name, Transition container,
 4           boolean resolveFuzzy, IResolveResult result) {
 5     for (Vertex targetCand: container.getContainer().getSubvertices()) {
 6       if (resolveFuzzy) {
 7         if (targetCand.getName().startsWith(name)) {
 8           result.addMapping(targetCand.getName(), targetCand);
 9         }
10       } else {
11         if (targetCand.getName().equals(name)) {
12           result.addMapping(identifier, targetCand);
13         }
14       }
15     }
16     if (result.isEmpty()) {
17       result.addError("Vertex " + name + " not defined");
18     }
19   }
20 }
```

Listing 1.4. `TransitionTargetReferenceResolver` implementation

als for transitions starting with "st". Fuzzy resolution usually differs only slightly from exact resolution. In the example (Listing 1.4) only `equals()` (Line 11) was exchanged for `startsWith()` (Line 7). If a developer does not care about fuzzy resolution and code completion, the flag can simply be ignored. The default resolvers, however, do implement fuzzy resolution. Thus, code completion is provided out of the box for all cases were the default resolution is not overridden. For de-resolving the default implementation can be kept here. If defined anyway, the code would consist of a statement that reads the `name` attribute of a given `State` and returns it.

Tailoring Token Resolving EMFText also supports to tailor the mapping of tokens (i.e., strings) to attribute values in the model. For instance, an input string valued `yes` (`no`) can be converted to the boolean value `true` (`false`). As this is a bi-directional mapping (from tokens to attribute values and vice versa), we call the two complementary processes resolving and de-resolving. For each token type, a `TokenResolver` is generated in which additional conversion rules can be implemented. Note that this is only needed for special mappings. Common conversions (e.g., for numbers) are already available.

5 Related Work

As discussed in [14], both generic and custom textual syntax can be beneficial for different application scenarios. Consequently, a variety of different TS tools and approaches exist. Some provide TS based on metamodelling languages (e.g., [14]), while others are explicitly designed for the specification of custom syntaxes. Goldschmidt et al. [15] presented a comprehensive overview of existing approaches and classified them within a multi-faceted classification schema.

In contrast to existing approaches, which either focus on one type of syntax or the other, our approach and the presented implementation (EMFText), was designed with the combined focus on refinement, derivation and evolution. Thus, it combines the strengths of both types of syntaxes. Presenting the differences between EMFText and all other approaches in detail is beyond the scope of this paper. Nonetheless, we will compare EMFText according to the classification in [15] with determined tools that share most of EMFText's concepts and architectural decisions and are thus most related to our work.

Rose et al. [14] presented an implementation of the generic syntax HUTN. The problems adressed by this implementation (rapidly changing metamodels) can be countered with EMFText as well. The tooling generated from a HUTN syntax specification that can be automatically derived by EMFText is very similar to the one provided by [14]. However, we allow users to refine this syntax where appropriate, which is not possible with a pure HUTN implementation. Furthermore, we can automatically generate syntax for new meta classes, keeping the customised syntax for existing ones.

The definition of customised syntax is supported by variety of tools. To our knowledge none of these tools provides support to either derive, refine or evolve syntax definitions. However, there is more differences we want to mention here.

Similar to EMFText, TCS [16] has a tight coupling between the mapping definition and the metamodel, which facilitates the definition of a syntax. It provides a generic editor for registered syntaxes, including syntax colouring, error marking, and navigation to underlying model elements. However, the latter comes at the price of a mandatory base class for all metamodel elements handled by the TCS editor. This is a limitation compared to EMFText's implementation which stores information regarding column and line numbering externally.

Sintaks [17] is built upon the Kermeta metamodelling facility and uses a concrete syntax metamodel to generate transformations between text and models. In contrast to EMFText the mapping between syntax rules and metamodel elements is explicit. The concrete syntax has to be specified via an EMF tree-based editor, which is not as convenient as using our specification language CS.

TEF [18] uses an interpretative approach (in contrast to the generative approach used in EMFText). Additionally, each rule needs to be associated explicitly with classes and properties of the metamodel using metamodel bindings. This differs from the implicit mappings used in EMFText which facilitates much shorter mapping descriptions. TEF supports background parsing and a Model View Controller (MVC) update strategy.

Monticore [19] supports an integrated specification of concrete and abstract syntax. It provides similar editing features as EMFText. Because of its integrated approach, Monticore is well suited to define a metamodel and an editor for a given textual language. This is a clear difference to our approach which targets to easily derive and refine concrete textual syntax for existing metamodels.

XText [20] is tightly integrated with EMF and is based on ANTLR. As Monticore, XText provides an integrated specification language and similar editing features as EMFText. However, neither it can be used to define more than one syntax per metamodel nor it allows to compose syntax for existing metamodels.

6 Conclusion

In this paper we presented a novel approach for defining textual concrete syntax for modelling languages based on derivation and refinement. We introduced the conceptual foundation for that approach and presented its implementation in the tool EMFText. Finally, we showed usage and applicability of approach and tool on different examples. EMFText allowed us to quickly define several text syntax for two modelling languages. The possibility to start with a generated default syntax but having all the customisation options turned out to be very helpful—in particular for first-time users.

In the future, we plan to extend our text editor and investigate further usages of the concrete text syntax. The tight integration with EMF and Eclipse can be extended to connect to other functionality which profits from textual representations, like diff and merge in version control. We also plan to improve EMFText to support a more declarative specification language for name analysis. The language will provide adequate support to declare different namespaces and scoping rules.

Currently, our tool implementation relies on ANTLR to generate parsers. However this imposes some restrictions: As a recursive descent LL(*) parser generator it does not support general left recursive grammars [4]. Furthermore ANTLR parsers use an extra scanner component to convert input character streams into token streams which may cause conflicts when composing syntax definitions with intersecting token definitions. To improve EMFText in these directions we plan to support alternative backends such that parser generators can be exchanged accordingly. The effort of transforming left recursive grammars into right recursive ones can then be avoided by switching to a tool that can handle left recursive grammars, for example a LR based parser generator [4]. Moreover, a scannerless parsing approach can be applied to prevent conflicts between token definitions [21].

Furthermore, we will define and analyse TS for more languages to find possible improvements to our approach. One direction here is to utilise EMFText for *type-safe* code generation—using model-to-model transformations and EMFText's ability to print models into text. Another, yet related, idea is to extend metamodels to obtain type safe template languages. We have done work into this direction in [22]. The question here is how a TS can be automatically extended along with a metamodel.

Acknowledgement

This research has been co-funded by the European Commission within the 6th Framework Programme project Modelplex #034081, by the German Research Foundation within the project HyperAdapt and by the German Ministry of Education and Research within the projects feasiPLe and SuReal.

References

1. TU Dresden: Software Technology Group: EMFText. http://emftext.org (2008)
2. Object Management Group: Human Usable Textual Notation (HUTN) Specification. Final Adopted Specification ptc/02-12-01 (2002)
3. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd Edition. Pearson Education (2008)
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques, and Tools. Addison Wesley (1986)
5. Meyer, B.: Introduction to the Theory of Programming Languages. Prentice Hall (1990)
6. Kleppe, A.: Software Language Engineering. Pearson Education (2009)
7. The Eclipse Foundation: Eclipse Platform. http://www.eclipse.org (2008)
8. Parr, T.: ANTLR — ANother Tool for Language Recognition — parser generator. http://www.antlr.org (oct 2008)
9. TU Dresden: Software Technology Group: EMFText: Concrete Syntax Zoo. http://emftext.org/zoo (2008)
10. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
11. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA (1990)
12. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping Features to Models. In: Companion Proc. of ICSE'08, New York, NY, USA, ACM (May 2008)
13. The Eclipse Foundation: EMF-based implementation of UML2 metamodel. http://www.eclipse.org/modeling/mdt/?project=uml2 (2008)
14. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.: Constructing Models with the Human-Usable Textual Notation. In: Proc. of the MoDELS 2008, Toulouse, France (2008) 249–263
15. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: Proc. of ECMDA-FA. Volume 5095 of LNCS., Springer (2008)
16. Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proc. of GPCE'06, New York, NY, USA, ACM (October 2006)
17. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In: Proc. of the MoDELS 2006, Genova, Italy (October 2006)
18. Scheidgen, M.: Textual Modelling Framework. http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/
19. Krahn, H., Rumpe, B., Völkel, S.: Efficient Editor Generation for Compositional DSLs in Eclipse. In: Proc. of DSM'07, Montreal, Quebec, Canada, Technical Report TR-38, Jyväskylä University, Finland (2007)
20. Efftinge, S., Völter, M.: oAW xText: a framework for textual DSLs. In: Workshop on Modeling Symposium at Eclipse Summit. (2006)
21. Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized lr parsers. In: Compiler Construction (CCŠ02), Springer-Verlag (2002) 143–158
22. Henriksson, J., Heidenreich, F., Johannes, J., Zschaler, S., Aßmann, U.: Extending Grammars and Metamodels for Reuse: The Reuseware Approach. IET Software **2**(3) (2008) 165–184