Classification of Concrete Textual Syntax Mapping Approaches

Thomas Goldschmidt[1], Steffen Becker[1], and Axel Uhl[2]

[1] FZI Research Center for Information Technologies
Karlsruhe, Germany
{goldschmidt,sbecker}@fzi.de
[2] SAP AG
Walldorf, Germany
axel.uhl@sap.com

Abstract. Textual concrete syntaxes for models are beneficial for many reasons. They foster usability and productivity because of their fast editing style, their usage of error markers, autocompletion and quick fixes. Furthermore, they can easily be integrated into existing tools such as diff/merge or information interchange through e-mail, wikis or blogs. Several frameworks and tools from different communities for creating concrete textual syntaxes for models emerged during recent years. However, these approaches failed to provide a solution in general. Open issues are incremental parsing and model updating as well as partial and federated views. To determine the capabilities of existing approaches, we provide a classification schema, apply it to these approaches, and identify their deficiencies.

1 Introduction

With the advent of model-driven development techniques, graphical modelling languages became more and more popular. However, there are also use cases where a concrete textual syntax (CTS) is more appropriate to edit models. For example, this applies to mathematical, expression-like languages such as query, or constraint languages (e.g. Object Constraint Language (OCL)[1]). Another example where textual syntaxes are preferred over graphical ones are model transformation languages such as QVT. Even in graphical modelling there are parts that can only be expressed and displayed textually in a convenient way, e.g., an operation signature in UML.

Advantages of such textual syntaxes are their clear structure (reading from left to right, from top to bottom, indentation as substructures) and their focus on straight ahead typing. Tools that can handle textual artefacts are widely spread and very mature. Especially software developers are used to having their development artefacts being developed as text. Helpers such as code highlighting, autocompletion, and error annotations elevate the capabilities of textual editors significantly. Diff/merge operations, the construction of patches, etc. are already well understood for text in contrast to graphically-noted models. Furthermore, submitting a part of text representing the model to a discussion forum or writing a mail containing snippets written in the concrete syntax is easy in a textual syntax because, due to its platform and tool independency, everyone can view and edit this text. Further advantages of a textual versus a graphical concrete syntax for users as well as for tool developers are mentioned in [2].

Basically, a CTS approach has to map constructs of a metamodel to the definition of a textual syntax, i.e., a grammar. Tools that translate between the textual representation as well as the abstract representation should be (automatically) derived from the mapping definition. Additionally, an editor could be provided that facilitates features such as syntax highlighting, autocompletion or error markers specific for the particular syntax.

However, for a CTS approach to prove applicable in an enterprise and in a large scale environment a lot of requirements have to be met [3,4]. Important requirements are for instance the incremental updating of existing models and the support for UUID-based repositories or the definition of partial and/or combined views.

A great variety of approaches and tools that provide concrete textual syntax mappings for models emerged recently or have been enhanced to support it. Originating from different communities, from academia as well as industry, their set of features is also very diverse. Some approaches facilitate the translation from text to model by parsing text from time to time in the background while others use a model-view-controller (MVC) pattern to keep the model in sync with the text. Being able to store format information in addition to the actual model, some approaches preserve the original format of the text over subsequent translation runs. However, there are still requirements that are not or insufficiently fulfilled by existing approaches.

The contributions of this paper are (1) a classification schema for CTS approaches, (2) the application of this schema to existing CTS approaches as well as the identification of their deficiencies and (3) the discussion of several important features that are not yet addressed. The presented classification schema is used to describe the necessary as well as extended features of a CTS mapping framework. Ten different approaches were examined for their support of these features. The discussion of the yet unsupported features focuses on the applicability of a CTS approach in an enterprise with a multitude of modelling languages, metamodels and tools and distributed, parallel development.

This paper is structured as follows. An overview on the foundations of a CTS approach is given in Sect. 2. Section 3 presents the classification schema that includes the features of a CTS approach. The actual classification is presented in Sect. 4. Section 5 discusses the findings and requirements for modelling in the enterprise. Related work concerning the classification of CTS approaches is treated in Sect. 6. Conclusions are drawn in Sect. 7.

2 Foundations of a Concrete Textual Syntax Mapping

Several basic components are needed to provide a comprehensive tooling for a CTS approach. To be able to relate constructs from a metamodel to elements of a CTS, a mapping between the metamodel and the definition of this syntax is needed. The definition of a textual syntax is provided by a grammar. To translate the textual syntax to its model representation, a *lexer*, a *parser* as well as a component that is responsible for the *semantical analysis* (type checking, resolving of references, etc.) are needed. Even for an approach that directly edits the model without having an explicit *parser* component for the grammar, a similar component is needed that decides how the text is translated into model elements. For reasons of convenience we will call all kinds and combinations of components that implement the translation form text to model a

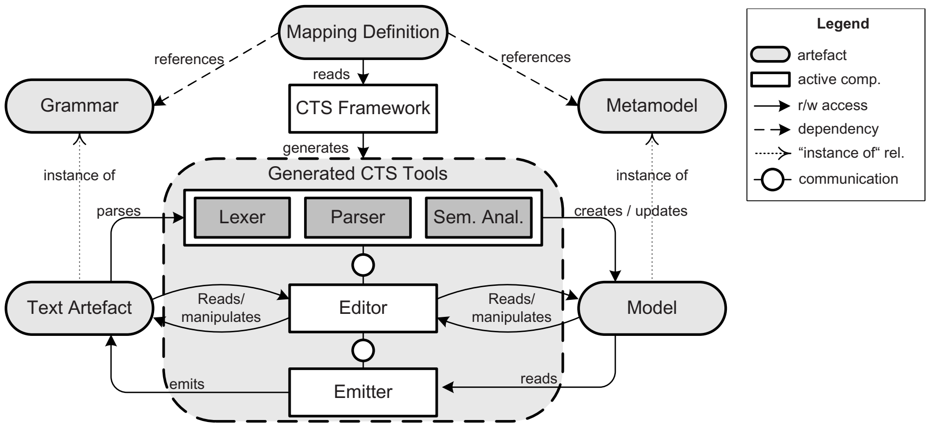

Fig. 1. General structure of a CTS framework

parser. The backward transformation, from model to text, is provided by an *emitter*. Both components can be generated using the above-mentioned mapping definition.

An overview of these components is depicted in Fig. 1. This figure shows that the CTS framework uses the information that is provided in the mapping definition to generate the parser, emitter and editor components. For example, the mapping could define that a UML class `c` is represented in the concrete syntax using the following template: `class <c.name> { <call to templates for contents of c> }`. The framework could then generate a parser that recognises this structure and instantiates a UML class when parsing this pattern and setting the name property accordingly. Furthermore, an emitter can use this template to translate an existing UML class into its textual representation.

The grammar⇔metamodel mapping can also be used to generate an editor for the language represented by the metamodel. This editor can then use the generated parser and emitter to modify the text and the model. This editor is then also responsible for keeping the text and the model in sync, e.g., by calling the parser everytime the text has changed. Based on the mapping definition several features of the editor can be generated, such as syntax highlighing, autocompletion or error reporting. Refactoring actions can also be provided with this editor. Having the model as well as the text in its direct access such an editor could, e.g, provide a rename action which updates the name property of an element on the model and then uses the emitter to update all occurrences of this name in the text.

3 Classification Schema

To be able to compare and classify different approaches that exist for the creation of a concrete textual syntax for a metamodel a systematic list of possible features of such an approach is needed. We chose to use feature diagrams [5] to provide an overview on the available features. Figure 2 depicts a feature diagram of the features considered

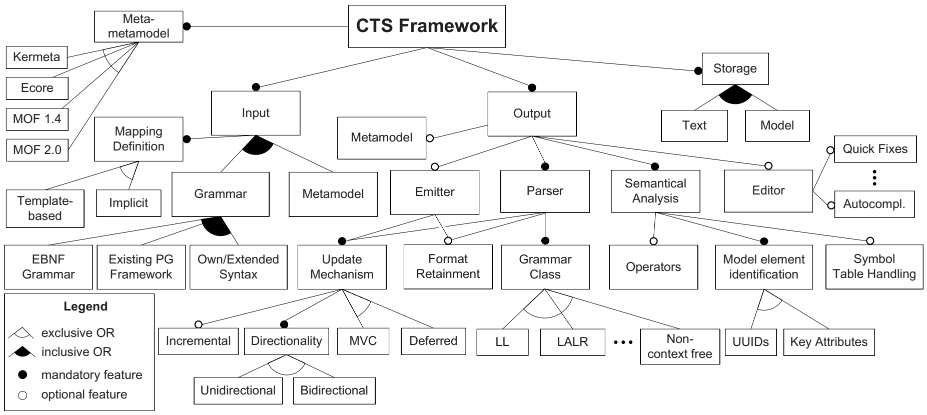

Fig. 2. Feature Diagram of all considered features

in this survey. The features shown in this figure are discussed in detail in the following subsections. How and if these features are provided by the actual approaches is shown in Tab. 1.

3.1 Supported Meta-metamodels (M_3)

Current approaches are based on different meta-metamodels: Ecore [6], different versions of MOF 1.4 [7] or 2.0 [8] or the Kermeta meta-metamodel [9] used by Sintaks [10]. Based on the capabilities of the meta-metamodels also the supported features of the textual syntax approaches vary. For example, MOF 1.4 uses UUIDs (in this case called MOFID) to identify model elements where Ecore uses designated key attributes. Because of these different approaches also the implementation based on one of these meta-metamodels needs to support the respective identification mechanism (see Sect. 5 for a detailed discussion on this problem).

3.2 Input and Output

Depending on the use case for a textual language and its editor, different artefacts may already exist or need to be created. Possible combinations are:

1. Existing language specification, e.g. with a formal grammar, no metamodel exists. This is a typical use case when existing languages, i.e. Domain Specific Languages (DSL), should be integrated into a model driven development environment.
2. Existing metamodel, no specification for concrete syntax. For the development of a new concrete syntax based on an existing metamodel this is an important use case.
3. Both, concrete syntax definition and metamodel exist. For this use case the mapping definition needs to be flexible enough to bridge larger gaps between concrete and abstract syntax (e.g., OCL).

For frameworks which use a grammar as input it should also be distinguished if it is possible to use standard (E)BNF grammars or if a proprietary definition for the CTS constructs is needed. For approaches that specify the concrete syntax based on an existing metamodel a template-based approach that defines how each metamodel element is represented as text may be used. The components needed to translate back and forth between text and model—namely parser and emitter—are considered an output of the CTS framework. This is closely connected to the bidirectionality support of an approach because it is clear that if it only supports one direction, one of these components is not needed. For example, an approach only supporting the translation from the textual syntax to the model representation would not generate an emitter.

In Tab. 1 the following abbreviations denote the input and output parts of the CTS frameworks: E=Emitter, G=EBNF grammar, G_{pg}=Reuse of an existing parser generator grammar definition, G_s=Proprietary grammar definition, M_2=Metamodel, P= Parser, T=Templates for the concrete syntax.

3.3 Update Mechanism

There are two main possibilities how changes of the text can be reflected in the model. First, a Model View Controller (MVC) like approach may be used. Using an MVC-based editor, all changes to the textual representation are directly reflected in the model and vice versa. This means that there are only atomic commands that transform the model from one consistent state to another. Hence, it is at every point in time consistent. Second, a deferred update approach may be used. The parser is called from time to time or when the text is saved. However, intermediate states of the text may then be out of sync with the model because it may not always be possible to parse the text without syntactical errors. Such an approach is for instance used in the background parsing implementation of the Eclipse JDT project.

These approaches are identified in Tab. 1 by: mvc=Model View Controller, bg= Background parsing.

3.4 Incrementality

If the translation between text and model is done incrementally, only the necessary elements are changed rather than the whole text or model. For example, model elements are kept, if possible, when the text is re-parsed. Vice versa, changes to the model would only cause the necessary parts of the text to be updated. Especially when dealing with models in which model elements are identified by a UUID an incremental update approach becomes more desirable. Here, incrementality is important to keep the UUIDs of the model elements stable so that references from other models outside the current scope do not break. Therefore it is important not to re-create model elements every time a model is updated from its CTS. A detailed discussion on the issues that arise when using a CTS approach on top of a UUID-based repository is performed in Sect. 5.

Even in a non UUID-based environment incrementality becomes important as soon as the textual representation reaches a certain size. Lexing, parsing, semantical analysis and instantiating model elements for the whole text causes a significant performance overhead.

In Tab. 1 the following abbreviations are used to distinguish these possibilities: y=Full support for incremental parsing/updating, n=No support for incremental updates, $y([p|e])$=Support only for incremental parsing(p) or emitting(e).

3.5 Format Retainment

If an emitter is used to translate models to their textual representation, users would expect that the format information of the text, such as whitespaces, is preserved. Furthermore, elements that are only present in the concrete syntax and not in the metamodel, as for example comments, also need to be retained. Especially when the textual representation is not explicitly stored but rather derived from the actual model (c.f., Sect. 3.12) format information has to be stored in addition to the model.

Possible values for this feature in Tab. 1 are the following: y=Format is retained upon re-emitting, n=No format retainment support.

3.6 Directionality

Bidirectional transformations between the abstract model representation and its CTS means that it is also possible to update existing textual representations if the model has changed. An initial emitter that produces a default text for a model can easily be produced using the information from the grammar or mapping definition. For a more sophisticated emitter, knowledge about formatting rules and format retainment is needed.

For updating existing representations, it would be expected that the user-defined format is retained. Imagine a textual editor that is used to create queries on business objects. Now an attribute in the business object model is renamed. This means that all references in the queries need to be updated. Hence, the queries need to be re-emitted from the model. For this case it would be desirable that the queries' format looks exactly the same as before that change rather than having the default format.

There are some difficult cases that should be considered: Imagine a series of inline ”//” comments that the user aligned nicely. When the length of the identifier changes, it will be tricky to know what the user wanted with the formatting: aligned comments or a specific number of spaces/tabs between the last character of the statement and the ”//” marker. Hence, perhaps there needs to be the possibility to specify the behaviour of such formatting rules within the mapping definition.

The following values are possible for this feature in Tab. 1: y=Completely bidirectional transformation, n=No bidirectionality supported, i=Creation of textual representation only initially.

3.7 Grammars Class

The parser component of an approach needs to have a grammar defined to be able to handle the textual input of the concrete syntax. Possible grammar classes are those of general-purpose programming languages such as LL or LALR [11]. However, it might be possible that even non context-free grammars may be used as input. Another possibility where no grammar in a usual form is needed would be a pseudo text editing approach. In such an approach no text file is edited but all modifications are directly applied to the model using an MVC approach (c.f., Sect. 3.3).

The following grammar classes are considered for Tab. 1: LL(1/k/*), SLR, LR, LALR, ncf=Non-context free, dir.=Direct editing.

3.8 Semantical Analysis

After the parser has analysed the structure of the text document links between the resulting elements need to be created. For example, a method call expression in an OCL constraint that was just parsed needs to be linked to the corresponding operation model element. To represent these links, two different concepts may be used by the model repository, either by their UUID or by designated key attributes (c.f., Sect. 5). As the choice of one of these mechanisms has a great impact on the implementation of the CTS approach (also see Sect. 5) this feature is also listed in this classification schema.

The following abbreviations are used in Tab. 1: UUID=Identification via UUID, Key-Attr.=Identification by designated key attributes.

3.9 Operators

Especially in mathematical expressions the use infix operators is widely spread. During the semantical analysis the priorities, arities and associativities of such expressions habe to be resolved. To be able to automatically translate a textual representation of such an expression into its abstract model this information needs to be present in the mapping definition. If such an automated support is present this allows the gap between the metamodel and the grammar to be much bigger. For approaches that generate a metamodel from the mapping definition this feature is mostly implicitly supported since the operator precedence is then directly encoded in the generated metamodel.

In Tab. 1 a y means that explicit support for operators is built into the framework, p means partial support exists and n means that a manual translation is needed.

3.10 Symbol Table

A symbol table is needed to handle the resolving of references within the textual syntax. As there is, mostly, only the containment hierarchy explicitely present within a CTS a symbol table is needed during the parsing process to resolve other references that are stated using e.g., named references. The support for custom namespace contexts (such as blocks in Java) can also be an important feature of the employed symbol table.

Possible values for this feature in Tab. 1 are the following: y=full support including custom contexts, p=partial support without additional contexts, n=no built-in symbol table.

3.11 Features of the Generated Editor

Most approaches also generate a comfortable editor for the concrete syntax. Functionality that is based on the abstract syntax (such as autocompletion or error reporting) can be provided based on the model. If the tool also supports bidirectional transformation, refactoring support (such as renaming, etc.) may be easily implemented using the model. Other possible features are syntax highlighting or quick fixes.

Table 2 shows an overview on the features of the generated editors of each framework. The following features are considered: autocompletion, syntax highlighting, refactoring, error markers and quick fixes.

3.12 Storage Mechanism

Having two kinds of representation, i.e. concrete textual syntax or abstract model, there are several possibilities to store the model. First, the model may be stored just using the concrete syntax. Second, only the abstract model is stored and the textual representation is derived on the fly whenever the textual editor opens the model. Then formatting information needs to be stored additionally to the model (c.f., Sect. 3.5). Third, both representations could be stored independent from each other. However, this means in most cases that they are not kept in sync with each other. Fourth, a hybrid approach may be implemented that stores the format information and merges them with the model when it is loaded into the editor. This additional format storage may then again be represented as an annotation model to the actual model or as some kind of textual template.

These different possibilities are identified in Tab. 1 using the following abbreviations: text=The textual representation is stored, mod.=The model is stored, both=Both representations are stored, hyb.=Hybrid storage approach.

4 Classification of Existing Approaches

According to the classification schema presented in Sect. 3, we evaluated several approaches that present a possibility to create a model based CTS. Table 1 lists the supported features of each approach. Table 2 shows the features of a potentially generated editor. All evaluations were based on the cited works and prototypes that were available at the time of writing. Future work proposals of these sources were not considered.

Bridging Grammarware and Modelware: Grammar-based approaches are used to automatically generate a metamodel for the CTS. Those metamodels are closely related to the grammar elements for which they were created. This inherently causes the metamodel to be relatively large. Reduction rules can be used to reduce the amount of metamodel elements that are produced for elements in the grammar.

For example, a trivial mapping would generate a class c_{nt_k} for each non-terminal nt_k in the grammar as well as one class c_{alt_i} for each alternative alt_i of nt_k. Furthermore, an association ref_{alt_i} would be generated that connects the c_{alt_i} to their c_{nt_k}. The c_{alt_i} then reference the corresponding c_{nt_j} for the referenced non-terminals nt_j of alt_i. One reasonable reduction rule for this scenario is: if nt_k references only alt_i with only one referenced non-terminal each, the ref_{alt_i} as well as c_{alt_i} could be omitted, reducing the whole structure to a direct generalisation between the c_{nt_j} and c_{nt_k}.

Some of these reduction rules can be applied automatically during the metamodel generation, while others need additional information given as annotations to the grammars.

Wimmer and Kramler [12] present such an approach. A multi-phase automatic generation that facilitates reduction rules as well as manual annotations reduces this amount to make the resulting metamodel more usable. The reduction steps that are applied during these phases then also implicitly define the mapping between the mapping definition. The main area where such an approach is useful is the Architecture Driven Modernisation (ADM)[16] where existing legacy code is analysed for migration, documentation or gathering of metrics.

Frodo: Frodo [13] was developed with the goal to provide a unified solution for the creation of a DSL. This approach presents an end-to-end solution for textual DSLs, providing support for the creation of a CTS as well as back-end support for the target DSL. It also makes initial attempts to derive a debugging support from the mapping specification. Frodo supports several sources for the definition of the CTS. Either a grammar metamodel may be specified or a specific grammar for a supported parser generator (currently ANTLR) could be used. An implicit mapping from the grammar to the DSL metamodel is automatically created. This is done by matching the names of classes and attributes to elements in the grammar rules. Additional mapping rules, such as those needed for the resolving of references between model elements can be specified on the grammar metamodel.

Grammar Based Code Transformation for MDA: The approach elaborated in [14], similar to **Bridging Grammarware and Modelware**, also relies on reduction and annotations to the grammar. However, this approach additionally facilitates the storage of format information as a decorator model attached to the actual model.

Sintaks (TCSSL): Fondement [20] presents an bidirectional approach that generates a parser (based on the ANTLR parser generator) and an emitter (using the JET template engine). The mapping definition is created using the MOF concrete syntax. For complex mappings which need several passes, e.g., for resolving references or performing type checking, a multiple pass analysis can be integrated into the mapping. The main idea of this approach is to have an n-pass architecture for the transformation from code to model. Intermediate models are hereby treated as models decorated with refinements. Model transformations are then used to subsequently transform these models and finally create the abstract model that then conforms to the target metamodel.

TCS: A similar technique is presented in [10]. This approach also provides a generic editor for the syntaxes handled by TCS. For each syntax that is defined using TCS, there is also a definition that can be registered in the editor. Within this definition, it can, for example, be specified how the syntax highlighting should be done. This editor uses text-to-model trace-links that are created during parsing to allow hyperlinks and hovers for references within the text. However, currently these links are implemented as attributes (column and line number of the corresponding text) on a mandatory base class (LocatedElement) for all metamodel elements handled by a TCS editor. If the metamodel classes do not extend these class, the trace functionality is disabled. In later versions of TCS, this issue might be resolved. The mapping definition of a TCS syntax is tightly coupled to the metamodel, which means that for each element in the metamodel there is one rule describing its textual notation. This tight coupling makes the definition of a syntax relatively easy. However, it is therefore not possible to define additional

Table 1. Comparison of related approaches.

Name(s)	Ref.	Input	Output	Bid.	Updates	Inc.	Grammars	Format	M_3	Ident.	Oper.	Symb. Tab.	Storage
Bridging Grammarware and Modelware	[12]	G	M_2, P, E	y	bg	n	LL(k)	n	Ecore	KeyAttr.	*n/a*	p	both
Frodo	[13]	G or G_{PG}, M_2, T[a]	P, E	y	bg	n	LL(*)	n	Ecore	KeyAttr.	n	p	both
Grammar Based Code Transformation for MDA	[14]	G_{PG}[b]	M_2, P, E	y	bg	n	LALR(1)	y	MOF 1.4	KeyAttr.	*n/a*	p	both
Gymnast	[15]	G	M_2, P, E	y[c]	bg	n	LL(k)/LL(*)	y	Ecore	KeyAttr.	*n/a*	n	both
HUTN	[23]	M_2	G, P, E	y	bg/mvc[d]	n	*n/a*[d]	n	MOF 1.4	KeyAttr.	n	p	*n/a*[d]
JetBrains MPS	[17]	G_s	M_2, P, E	y	mvc	y?[e]	?[f]	y	prop.	?[f]	y	y	mod.
MontiCore	[19,18]	$I(M_2, G)$[g]	P, E	y	bg	n	LL(k)	n	Ecore	KeyAttr.	n	y	text
TCS	[10]	M_2, T	G, P, E	y	bg	n	LL(*)	n	Ecore	KeyAttr.	y	y	both
Sintaks(TCSSL)	[20]	M_2, T	G, P, E	y	bg	n	LL(*)	n	Kermeta	KeyAttr.	p	p	both
TEF	[21]	M_2, T	G, P	n	bg	n	SLR,LR,LALR	n	Ecore	KeyAttr.	y	y	text[h]
xText	[22]	G_s	M_2, P	n	bg	n	LL(*)	n	Ecore	KeyAttr.	y	p	text

Legend:
Input/Output: E=Emitter, G=Grammar, G_s=Own Grammar Definition, G_{pg}=Reuse of parser generator framework grammar definition, M_2=Metamodel, P=Parser, T=Templates for the CS
Updates: bg=Background parsing, mvc.=Model View Controller based
Storage: mod.=Model

[a] Frodo allows the import of standard EBNF or ANTLR grammars that can then be annotated with mapping rules.
[b] Currently there are implementations for SableCC and ANTLR.
[c] Navigation only, no transformation from model to text.
[d] This depends on the actual implementation of the HUTN standard.
[e] Seems to be supported somehow due to MVC approach.
[f] This feature could not be evaluated.
[g] The metamodel as well as the concrete syntax are defined within the same file.
[h] It is possible to directly access the underlying model via a special implementation of the Eclipse DocumentProvider interface.

rules in the mapping that are e.g., needed to resolve left recursions [11] within a LL grammar.

MontiCore: Another approach to integrate a textual concrete syntax with a corresponding metamodel is presented by Krahn et al. in [18]. This approach facilitates an integrated definition where the abstract syntax (the metamodel) is also defined within a grammar like definition of the concrete syntax. For simple languages and especially for languages where only one form of presentation, i.e. the textual syntax, is used, this approach seems to be promising. However, if a metamodel may have several presentation forms, or if only parts of the metamodel are represented as text, the tight integration of concrete and abstract syntax this approach promotes does not seem to be applicable. MontiCore allows the composition and inheritance of different languages and provides a comfortable support for generating editors from these composite specifications [19].

HUTN: The Human-Usable Text Notation (HUTN) approach [23], now specified as a standard by the Object Management Group (OMG) can be used to generate a standard textual language for a given metamodel. It focuses on an easy-to-understand syntax as well as the completeness of the language (it is able to represent all possible metamodel instances). Furthermore all languages, though each language is different, conform to a single style and structure and it is not possible to define an own syntax for a metamodel. In HUTN a grammar for a metamodel, including parser and emitter is generated.

There were currently only two implementations for HUTN. An early implementation was developed by the DSTC, named TokTok. However, this implementation is not available anymore. Another implementation was developed by Muller and Hassenforder [24], who examined the applicability of HUTN as a bridge between models and concrete syntaxes. They identified several flaws in the specification that make it difficult to use.

TEF: The first version of the Textual Editing Framework (TEF) presented in [21] was based on an MVC updating approach. Inherent problems with this concept (see Sect. 5) led to the choice of background parsing as the final method for updating the model. An interesting feature of TEF that is not directly mentioned in the classification schema is the possibility to define multiple syntactic constructs for the same metamodel element. Vice-versa, it is also possible to use the same notation for different elements by providing a semantic function that selects the correct function based on the context.

JetBrains MPS: JetBrains developed a framework called Meta Programming System (MPS) [17,25] that allows to define languages that consist of syntactical elements that look like dynamically-arranged tabulars or forms. This means, that the elements of the language are predefined boxes which can be filled with a value. MPS follows the MVC updating approach that allows for direct editing of the underlying model. This allows the editor to easily provide features such as syntax highlighting or code completion. However, it is not possible to write code that does not conform to the language. Hence, a copy, paste, adapt process is not possible in this approach.

xText: Developed as part of the openArchitectureWare (oAW) framework, xText [22] allows the definition of a CTS within the oAW context. xText generates an intermediate metamodel for the concrete syntax from the mapping specification. For this reason the framework provides an EBNF-like definition language which facilitates features like

Table 2. Editor capabilities

Name(s)	Reference(s)	Autocomp.	Err. mark.	Refactoring supp.	Quick fixes
Frodo	[13]	n	?[a]	n	n
Gymnast	[15]	y	y	y	n
IntelliJ MPS	[17]	y	y[b]	y	y
MontiCore	[19]	y	y	y	n
TCS	[10]	n	y	n	n
TEF	[21]	y	y	y	n
xText	[22]	y	y	y	n

[a]This feature could not be evaluated.
[b]No syntactic errors possible because of resolute MVC concept.

the possibility to specify identity properties or abstract classes. A translation into an instance of the intended target metamodel needs to be done by developing a model to model transformation from this intermediate language into the target abstract syntax. Having such an intermediate metamodel complicates the bidirectional mapping as an additional transformation for the backwards transformation is needed.

Gymnast: Garcia and Sentossa present an approach called Gymnast in [15], which similar to xText generates an intermediate language on which the editor is based. Refactorings, occurrence markings, etc., are provided by the generated editor based. As this work's main focus is on the generation of the editor and its functionality, a mapping to an existing target metamodel has to be developed in addition to the generated tools.

Other Approaches: A recently emerged project on eclipse.org that also aims to tackle the development of CTSs is the **Eclipse IMP** [26]. However, the current focus of the project lies on the easy development of an editor and not on the integration with a model repository. Still, it was stated in the project's declaration of intent that an integration with EMF is projected. Furthermore, there is an eclipse.org project called **Textual Modelling Framework (TMF)**[27] that currently regroups TCS and xText under a common architecture. A different approach is followed by Intentional Software's **Intentional Programming** [28]. Here, it seems to be possible to directly edit the model using a text editor. Multiple syntaxes, also textual ones, can be defined and handled even in the same editor. However, being a proprietary approach, the integration with existing standards-based repositories may be problematic.

5 Discussion

Current CTS implementations span a continuum between text and model affine-frameworks. On the text-end-side there are tools that stem from conventional programming languages. The idea to create development tools that are based on a model rather than on plain text was already developed before model-driven development became prominent. The IDE of Smalltalk systems follows a similar paradigm. Code artefacts are also Smalltalk objects that are edited using a specialised editor. Especially considering refactoring, this is a great advantage. Another example where boundaries between a textual and a model view on code are starting to blur is the Eclipse Java Development Tools (JDT) project. Even though the Java source files are still stored as plain text files, JDT uses indices and meta-data in the background to be able to provide comprehensive refactoring, navigation and error reporting capabilities. Many tools that were considered here (such as Gymnast or xText) focus on this end of the spectrum. On the other end of this spectrum, there are tools such as the MPS framework treating text as a sequence of frames or compartments containing text blocks. While text-based issues such as diff or merge are solved on text artefacts, solutions for these problems are still immature concerning models. Vice versa, issues that can easily be or are already solved on models, such as the usage of UUIDs for references or partial and combined views on models are challenges that none of the currently available CTS approaches is able to handle.

As it can be seen in Tab. 1 support for incremental model updates is currently not widely available. Only MPS provides some support for this. Furthermore, none of the approaches under evaluation support the UUID identification mechanism for model elements. Even HUTN, which is an OMG standard explicitly specifies that key attributes need to be specified in order to realise model element identification. That lack of these features complicates the application of these approaches in certain environments. The next sections discuss some of the yet untreated issues.

5.1 Universally Unique Identifiers

There are two different possibilities how model repositories can handle links between model elements: either by defining designated key attributes for each model element (as, e.g., in EMF/Ecore) or by assigning each element a Universally Unique Identifier (UUID) that remains stable across the lifetime of the element (e.g, the MOFID in MOF 1.4) [3,4]. Very important issues arise when trying to put a parser-based approach on top of a repository that uses UUIDs to identify model elements. In large scale environments with a high number of model partitions and numerous connections between those partitions, such repositories become very important. In distributed development where developers of one artefact do not always know all referrers from other model partitions to a specific model element it is crucial that elements have stable IDs. Further advantages of the UUID-based approach can be found in [3].

In textual syntaxes identification through UUIDs becomes problematic for several reasons:

– **Storage Mechanism:** If a model artefact is stored using the concrete syntax only, there needs to be the possibility to store these IDs somewhere in the text. However, during development a developer should not see these IDs as they contradict the crisp textual view and make it confusing. On the other hand, if the artefact is solely stored as model, other problems concerning the update from text to model arise (see below).

– **Model Updates:** Updating existing models is an inherent problem of textual notations. In graphical or forms-based modelling only a comparably small set of changes may occur that can easily be wrapped into command structures that are executed in a transactional way, transforming the model from one consistent state to another. In a textual editor this is very difficult, especially when IDs are not present

in the textual representation. A small change, e.g., adding an opening bracket in the text may alter the whole structure of the text, making it difficult to identify which elements in the model are meant to be kept and which are not.
- **Creation and Deletion of Model Elements:** In a graphical editor there are explicit commands to create and delete model elements. Within a textual editor this is difficult mostly because the creation or deletion of model elements is done implicitly. For example, if the name of an element is changed in the textual syntax it may either mean that the old element should be deleted and a new one should be created, or a simple rename of the existing model element may have been intended.

A solution that lets transformations produce stable UUIDs for new model elements was proposed in [4]. Such an approach may, for example, use the ID of the source element and some transformation ID to compute the ID of the target element. However, if text is used as source there is no stable ID for a source element, just properties derived directly from the textual representation, such as a name attribute. Hence, no stable ID for a target element can be computed and the only way to keep the identity is to rely on incremental updates of the model (c.f. Sect. 5.2).

5.2 Update Mechanism

Many of the aforementioned problems may be solved by employing an MVC-based update mechanism. However, there are still other problematic constructs in this concept. Consider for example an expressions such as “(a+b)*c”: At the time of typing the expression in the parentheses, it can not be known that the model that actually should be created would have the “*” expression as root node and the parenthesised expression only as a subnode. A discussion of the advantages and disadvantages of the MVC and the background parsing approach can be found in [21].

Closely related to the update issue is the general problem of incremental updates. In compiler construction literature this problem was already discussed. For example, Wagner [29] developed a methodology that allows incremental lexing as well as parsing. Furthermore, Reps et. al. [30] present an incremental process that allows incremental updates to the attributed trees that result from the semantical analysis. However, such techniques were not adopted by any of the CTS frameworks under evaluation.

5.3 Partial and Combined Views

One advantage of graphical modelling is that it is easily possible to define partial views on models. This means that it is possible to create diagrams that highlight only a specific aspect of the model while hiding other parts. for example, in UML one diagram may be used to display an inheritance hierarchy of classes only while another diagram is used to show the associations between these classes. Defining models with a CTS should also include the possibility to do this. However, this imposes that there are two different modes for deleting elements. One, which deletes only the text and another which deletes the model element from the text *and* the model. Using only standard text editing techniques (typing characters and using backspace or delete to remove them) there is no possibility to distinguish between both commands.

6 Related Work

An analysis of existing approaches for the creation of a CTS was conducted in [13]. However, neither a systematic schema was used to classify these approaches, nor was a comprehensive analysis of the approaches' features provided. Garcia [15] presented an overview of existing approaches and highlighted some of their peculiarities. Nevertheless, an extensive analysis was considered "a lengthy, lengthy, affair" and therefore omitted. Several of the papers presenting the CTS approaches contain a short section on related work, but they also only give hints.

7 Conclusions and Future Work

In this paper we presented a classification schema for CTS mappings and their frameworks. We applied this classification to several academic and industrial approaches. The presented classification is not exhaustive. However, the classification schema can be used to evaluate, classify and compare further approaches. Furthermore, we identified issues that currently complicate the use of these frameworks in a large-scale enterprise MDSD environment. Issues like those highlighted in our discussion need to be solved in order to apply a CTS framework in a larger model-driven environment.

The contribution of this paper is useful for the practitioner, as he can select an adequate approach for his particular requirements based on the presented classification. Furthermore, it is also interesting for academia, as several unresolved issues concerning a CTS mapping are highlighted.

We plan to develop a CTS approach that explicitly supports UUID-based repositories with all consequences, such as the need for incremental parsing and updating. The envisioned approach should also support partial and combined views.

References

1. Object Management Group: Object Constraint Language (OCL) 2.0. Doc. No 05-06-06
2. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Textbased modeling. In: Proc. of the 4th Int. Workshop on Software Language Engineering (ateM 2007) (2007)
3. Uhl, A.: Model-driven development in the enterprise (2007), `https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/7237`
4. Uhl, A.: Model-driven development in the enterprise. IEEE Software 25(1), 46–49 (2008)
5. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. re 0, 139–148 (2006)
6. Eclipse Foundation: Eclipse modeling project last visited: 24.01.2008, `http://www.eclipse.org/modeling/`
7. Object Management Group: Meta Object Facility (MOF) 1.4. Doc. No 02-04-03
8. Object Management Group: MOF 2.0 core final adopted specification. Doc. No ptc/03-10-04
9. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Proc. of MODELS/UML 2005 (2005)
10. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE 2006, pp. 249–254 (2006)
11. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)

12. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844. Springer, Heidelberg (2006)
13. Karlsch, M.: A model-driven framework for domain specific languages. Master's thesis, University of Potsdam, Hasso Plattner Insitute (2007)
14. Goldschmidt, T.: Grammar based code transformation for the model driven architecture. Master's thesis, Hochschule Furtwangen University, Furtwangen, Germany (August 2006)
15. Garcia, M., Sentosa, P.: Generation of Eclipse-based IDEs for Custom DSLs. Technical report, Software Systems Institute (STS), TU Hamburg-Harburg, Germany (2007)
16. Object Management Group: Architecture Driven Modernization (ADM), `http://www.omg.org/adm/`
17. JetBrains: MPS. last visited: 26.03.2008, `http://www.jetbrains.net/confluence/display/MPS/`
18. Krahn, H., Rumpe, B., Völkel, S.: Integrated definition of abstract and concrete syntax for textual languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, Springer, Heidelberg (2007)
19. Krahn, H., Rumpe, B., Völkel, S.: Efficient editor generation for compositional dsls in eclipse. In: Proc. 7th OOPSLA Workshop on Domain-Specific Modeling (DSM 2007) (2007)
20. Fondement, F.: Concrete syntax definition for modeling languages. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (2007)
21. Scheidgen, M.: Textual editing framework (2007), `http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html`
22. Efftinge, S.: Xtext reference documentation (2006), `http://www.eclipse.org/gmt/oaw/doc/4.1/r80_xtextReference.pdf`
23. Object Management Group: Human-Usable Textual Notation (HUTN) Specification. Doc. No formal/04-08-01 (2004)
24. Muller, P.A., Hassenforder, M.: HUTN as a bridge between modelware and grammarware - an experience report. In: 4th Workshop in Software Model Engineering WiSME 2005 (2005)
25. Dimitriev, S.: Language oriented programming: The next programming paradigm. onBoard Magazine 2 (2005)
26. Fuhrer, R.M., Charles, P., Sutton, S., Vinju, J., de Moor, O.: Eclipse IDE Meta-tooling Platform (The Eclipse IMP) (2007), `http://www.eclipse.org/proposals/imp/`
27. Eclipse Foundation: Textual modeling framework. last visited: 24.01.2008, `http://www.eclipse.org/proposals/tmf/`
28. Simonyi, C.: Intentional software (2007), `http://www.intentsoft.com/`
29. Wagner, T.A.: Practical Algorithms for Incremental Software Development Environments. PhD thesis, University of California at Berkeley (1998)
30. Reps, T., Teitelbaum, T., Demers, A.: Incremental context-dependent analysis for language-based editors. ACM TOPLAS 5(3), 449–477 (1983)