Paraphrasing Reference Models and Transformations[*]

Andrés Vignaga

MaTE, Department of Computer Science, Universidad de Chile

avignaga@dcc.uchile.cl

Abstract

Textual descriptions in natural language of models are often used as a complement to graphical notations or source code for documentation purposes. In this work we discuss our experiments on paraphrasing two variants of the abstract notion of model, as defined by the Global Model Management (GMM) approach. We use model transformations for paraphrasing reference models and transformation models.

1 Introduction

Models are mostly expressed using graphical notations. However, graphical forms are usually complemented by a textual description in natural language. The main purpose of this is enhancing understandability for human readers. In fact, descriptions in natural language are not as amenable as models for mechanical treatment. Textual descriptions thus focus on documentation.

Although documentation is an important aspect of a product, these textual descriptions are usually produced by humans. That process is time consuming and the result may be of little value if it is not carried out properly. Maintaining documentation is also costly because the text and the model it describes must be synchronized. It is a common situation to find outdated descriptions or no descriptions at all.

In this work, we address the automatic generation of textual descriptions of models. We transform models to descriptions of them in natural language. A model transformation first produces a model of a simplified natural language containing the full text of the description. A simple model extractor then generates the actual text from that intermediate model.

The transformation that produces the intermediate textual model naturally relies on the language used for expressing the model to be described. This means that for each language we need a specific transformation for processing the corresponding models. Based on the classification of model types introduced by Global Model Management (GMM) [4], which is partially illustrated in Fig. 1, we focus on the ReferenceMetamodel and TransformationModel variants. Specifically, the languages that we address in this work are KM3 [3] for reference models and ATL [6] for transformation models.

The rest of this work is structured as follows. Section 2 provides an overview of the prototype we developed for this experiment. Section 3 describes the metamodels involved in the implementation. The paraphrasing of KM3 metamodels is discussed in Sect. 4, while the paraphrasing of ATL transformation models is discussed in Sect. 5. In Sect. 6 we illustrate the application of both prototypes with concrete examples. Section 6 refers to the implementation of the prototype. Section 7 concludes.

[*]This work was carried out in collaboration with the AtlanMod team (INRIA and EMN), and was funded by CONICYT Chile and Universidad de Chile.

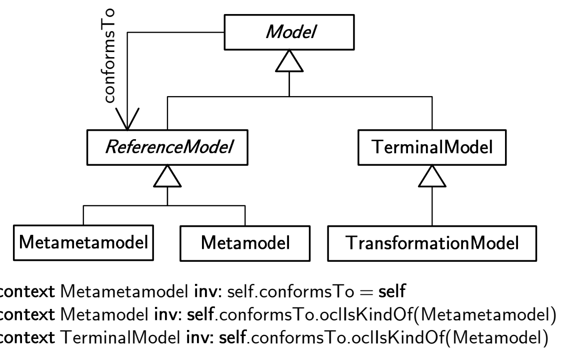
Model
conformsTo
ReferenceModel
TerminalModel
Metametamodel
Metamodel
TransformationModel
context Metametamodel inv: self.conformsTo = self
context Metamodel inv: self.conformsTo.oclIsKindOf(Metametamodel)
context TerminalModel inv: self.conformsTo.oclIsKindOf(Metamodel)

Figure 1: Partial Global Model Management extension of AM3Core metamodel with constraints

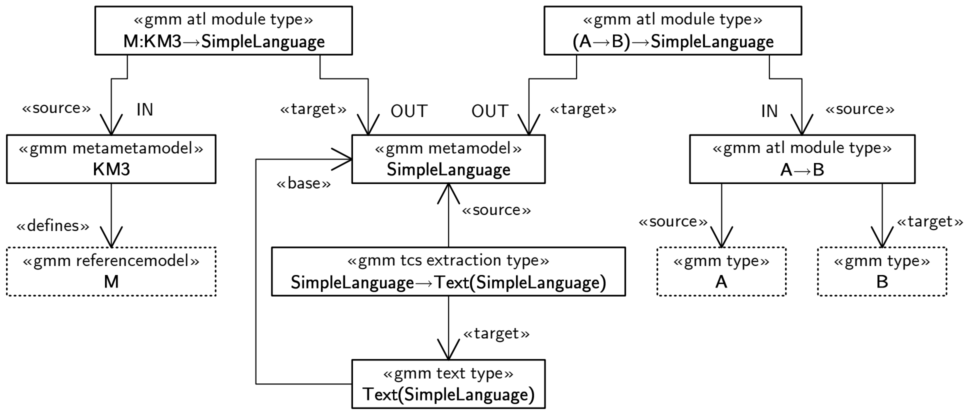
«gmm atl module type» M:KM3→SimpleLanguage
«gmm atl module type» (A→B)→SimpleLanguage
«source»
IN
«target»
OUT
OUT
«target»
IN
«source»
«gmm metametamodel» KM3
«gmm metamodel» SimpleLanguage
«gmm atl module type» A→B
«base»
«source»
«defines»
«source»
«target»
«gmm referencemodel» M
«gmm tcs extraction type» SimpleLanguage→Text(SimpleLanguage)
«gmm type» A
«gmm type» B
«target»
«gmm text type» Text(SimpleLanguage)

Figure 2: Types of entities involved in paraphrasing transformations

2 Overview

Our solution is composed of two model-to-text chains of transformations; the first is called *ParaphrasingReferenceModels* and the second is called *ParaphrasingTransformations*. Both chains are composed of a model-to-model transformation and a model-to-text transformation (i.e. an extractor). While the model-to-model transformations are specific for each chain, they share the second component. Figure 2 illustrates the types of the entities involved in both chains.

The first component of the *ParaphrasingReferenceModels* chain is a model-to-model ATL transformation called KM32SL. It transforms a KM3 reference model to a model of a simplification of the natural language. The second component of the chain is a model-to-text transformation (i.e., a TCS extractor [5]) called SLExtractor. It transforms the intermediate model produced by KM32SL to a text file. Figure 3 shows these entities, along with the entities involved in the application of *PraphrasingReferenceModels* to the KM3 metamodel itself. Such an application is further discussed in Sect. 6.1. The KM3 metamodel conforms to itself and is used as the source of KM32SL by appropriately binding parameter M to KM3. Transformation KM32SL produces a terminal model tmp which conforms to SimpleLanguage. This model is in turn the source for the extractor SLExtractor, which finally produces the KM3Text textual entity.

The types of the entities involved in *ParaphrasingTransformations* is similar and can be seen also in Fig. 2. Its first component is a HOT model-to-model ATL transformation

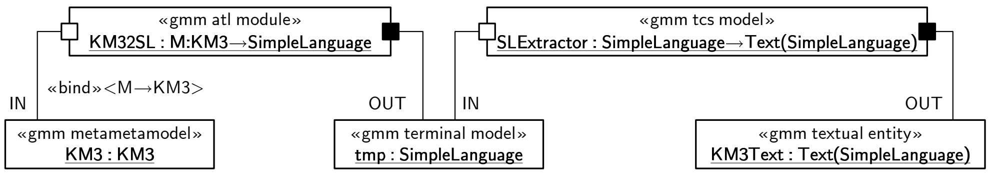

Figure 3: An application of *ParaphrasingReferenceModels*

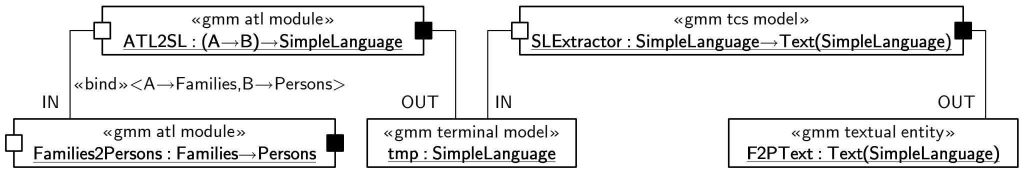
«gmm atl module»
ATL2SL : (A→B)→SimpleLanguage
«gmm tcs model»
SLExtractor : SimpleLanguage→Text(SimpleLanguage)
«bind»<A→Families,B→Persons>
IN
OUT
IN
OUT
«gmm atl module»
Families2Persons : Families→Persons
«gmm terminal model»
tmp : SimpleLanguage
«gmm textual entity»
F2PText : Text(SimpleLanguage)

Figure 4: An application of *ParaphrasingTransformations*

called ATL2SL. This transformation generates a model of our simplified natural language from another ATL transformation model. The second component of this chain is the same extractor described before. The application of this chain to the Famililes2Persons transformation is illustrated in Fig. 4 and is discussed in Sect. 6.2. In this case, the ATL transformation model Families2Persons is the source of ATL2SL by instantiating parameters A and B to Families and Persons respectively. Note that since the result does not depend on parameters A and B, their instantiation is only important for making sure that this is a valid application. The result of ATL2SL is the terminal model tmp which conforms to the SimpleLanguage metamodel as before. This model is then transformed to the F2PText textual entity using the SLExtractor extractor.

In the next sections both transformation chains are described in detail. An example of the application of both chains is also provided.

3 Metamodels

The sources for *ParaphrasingReferenceModels* and *ParaphrasingTransformations* are models which conform to KM3 and ATL metamodels respectively. The target of both chains is a plain text file. As an intermediate result, in both cases a model of a simplified natural language is produced. Therefore, three metamodels are involved in these chains: KM3, ATL, and SimpleLanguage. Note that the description in Sect. 3.3 is actually quite similar to that produced by *ParaphrasingReferenceModels* when applied to the metamodel of SimpleLanguage.

3.1 KM3

The KM3 metamodel used in the *ParaphrasingReferenceModels* transformation chain as the source of KM32SL is the standard one [2] and will not be discussed in detail in this document. However, as part of the example in Sect. 6.1, such a metamodel is expressed in KM3 syntax.

3.2 ATL

The ATL metamodel used in the *ParaphrasingTransformations* transformation chain as the source of ATL2SL is the standard one [2] and will not be discussed in this document.

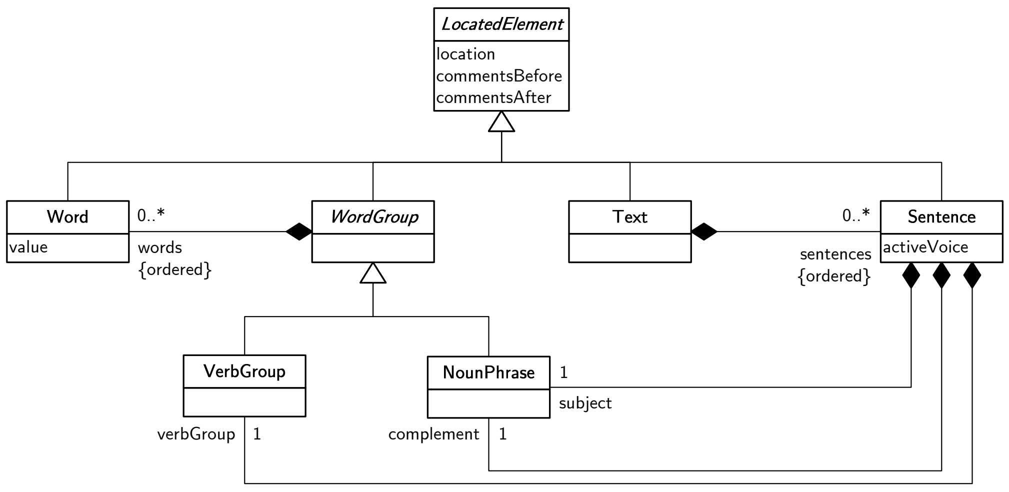

Figure 5: SimpleLanguage metamodel

3.3 SimpleLanguage

The metamodel of SimpleLanguage is illustrated in Fig. 5. A Text is composed of a sequence of Sentences, which is composed of a subject, a verb group and a complement. A Sentence may be expressed in active voice or in passive voice; the difference is the order in which its components should be considered for obtaining an actual English sentence. In turn, every component of a Sentence contains a sequence of Words. A Word has a string value which represents the word itself.

4 Paraphrasing Reference Models

This section describes the rationale and design of each step of *ParaphrasingReference-Models*.

4.1 From KM3 to SimpleLanguage

The first step in *ParaphrasingReferenceModels* transforms a KM3 reference model to a model of our simplified natural language. Transformation KM32SL is purely syntactical, in the sense that it does not try to guess the purpose or meaning of elements in the source model. This would go beyond the scope of this experiment.

4.1.1 Strategy

Transformation model MK32SL defines a fixed set of “template sentences”. Each template is instantiated for obtaining an actual sentence, using to that end specific information taken from elements in the source model. Each class of source element has associated a number of templates. Depending on the information held by concrete instances of a class, some, all or even none of the templates may be used for producing sentences in the target model. For example, an instance of Class may induce up to four different sentences: (a) one for describing its attributes, (b) one for describing its component classes, (c) one for describing its subclasses, and (d) one for describing its associated classes. Which of these four sentences will actually occur and what would be their specific words will depend on the elements within the particular reference model.

In what follows we express a mapping between elements in a KM3 reference model and their corresponding templates. We highlight the component parts of a sentence using colors. The part colored in blue is the subject of the sentence, the part in green is the verb group, and the part in red is the complement.

Metamodel

T1: "This metamodel contains packages **PName**$_1$, **PName**$_2$, ... and **PName**$_n$"

Where PName$_i$ are the names of the packages contained in the metamodel.

Package

T2: "Package **PName** contains the following classes"

Where PName is the name of the package. A sentence generated from this template is followed by a number of sentences generated from the classes contained in the package.

T3: "Additionally, package **PName** contains the following enumerations"

Where PName again is the name of the package. If a sentence is to be generated from this template, it will be followed by the sentences generated from the enumerations contained in the package. Additionally, if the package contains no classes and thus a sentence from this template is the first to be generated, the "Additionally" at the beginning of the sentence is removed. This applies to all similar cases from now on.

T4: "Finally, package **PName** contains data type **DTName**$_1$, **DTName**$_2$, ... and **DTName**$_n$"

Where PName is the name of the package, and DTName$_i$ are the names of the data types contained in the package.

Class

T5: "A[n] **CName** [is an abstract entity that] defines the **AName**$_1$, **AName**$_2$, ... and **AName**$_n$"

Where CName is the name of the class, and AName$_i$ are the names of the attributes owned by the class.

T6: "A[n] **CName** [is an abstract entity, and] is composed of a[n] [(set of | sequence of)] **TName**$_1$ [which play[s] the role of **RName**$_1$], a[n] [(set of | sequence of)] **TName**$_2$ [which plays the role of **RName**$_2$], ... and a[n] [(set of | sequence of)] **TName**$_n$ [which plays the role of **RName**$_n$]"

Where CName is the name of the class, TName$_i$ is the name of the opposite class and RName$_i$ is its corresponding role name. The "set of" and "sequence of" parts of this template depend on the *upper*, *isUnique* and *isOrdered* values of the corresponding feature of the class. The role name is omitted from the sentence when it matches the name of the opposite class.

T7: "A[n] **CName** [is an abstract entity, and] (can | must) be a[n] **SName**$_1$, **SName**$_2$, ... or **SName**$_n$"

Where CName is the name of the class and SName$_i$ are the names of its direct subclasses. If the class is concrete its instances can also be instances of any subclasses, but if the class is abstract then its instances must be also instances of one of the subclasses.

T8: "A[n] **CName** [is an abstract entity, and] (can have | has) [a[n]] **RName**$_1$ which correspond[s] to **OName**$_1$, and [a[n]] **RName**$_2$ which correspond[s] to **OName**$_2$, ... and [a[n]] **RName**$_n$ which correspond[s] to **OName**$_n$"

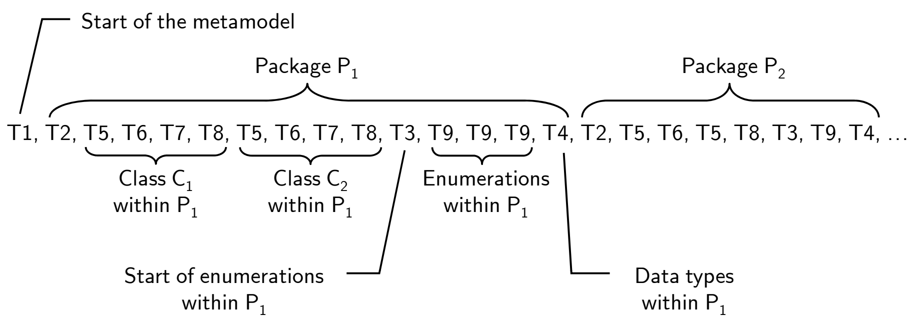
Start of the metamodel
Package P_1
Package P_2
T1, T2, T5, T6, T7, T8, T5, T6, T7, T8, T3, T9, T9, T9, T4, T2, T5, T6, T5, T8, T3, T9, T4, ...
Class C_1 within P_1
Class C_2 within P_1
Enumerations within P_1
Start of enumerations within P_1
Data types within P_1

Figure 6: Sequence of sentences in target model

Where CName is the name of the class, RName$_i$ is the opposite role name, and OName$_i$ is the name of the opposite class. The "can have" and "has" parts are resolved depending on the lower value of the corresponding feature.

Enumeration

T9: "Enumeration **EName** defines enumeration literals **LName**$_1$, **LName**$_2$, ... and **LName**$_n$"

Where EName is the name of the enumeration, and LName$_i$ are the names of its literals.

Note that not every concrete class maps to one or more templates. However, information included in instances of other concrete classes such as DataType, EnumLiteral, Attribute or Reference is used in one of the templates expressed above and thus not lost. For example, the names of data types are used in T4, and information of references is expressed in T6 and T8.

4.1.2 Design

In this section some noteworthy details about the realization of KM32SL are explained. The form of the output is first described. This sets the basis for better describing the design of the transformation.

The result essentially contains one single instance of Text which in turn contains a sequence of Sentence elements, each of them generated from one of the templates described above. The structure of such a sequence is illustrated in Fig. 6.

The first sentence must be generated from template T1 in every case. After that, a sequence of sequences of sentences comes. Each of these sequences describes the contents of a package, and is any non empty subsequence of {T2, T3, T4}. If T2 occurs, it must be followed by a sequence of non empty subsequences of {T5, T6, T7, T8}, each of them describing a class. If T3 occurs, it must be followed by at least one sentence generated from T9.

It is clear that the sequence described above is the result of flattening some of the elements contained in the source reference model. To that end, a series of flatten() helpers is defined on such elements. In Metamodel, flatten() just concatenates the results of flatten() called on each contained package.

The implementation of flatten() for Package uses the output elements (all of type Sentence) of the rules that transform Package elements. In fact, if the package contains classes, enumerations and data types, it will be transformed by a rule which produces three sentences; one corresponding to T2, one corresponding to T3 and other corresponding to T4. These sentences are accessed within flatten() via the resolveTemp() operation. In case the package to be transformed contains classes and data types only, then it will be transformed by a rule that produces just two sentences; the first corresponding to T2 and the second corresponding to T4. Therefore, there are as many rules that transform a Package as combinations of non empty subsequences of {T2, T3, T4}, which totals an amount of seven rules. For being complete, this implementation of flatten() must flatten all classes (if exist) and all enumerations (if exist), and place the results in the proper position among the sentences already discussed. Flattening enumerations is straightforward, since there are no dependencies among them. Therefore they are just linearized in any order.

Flattening classes is more complicated, since classes may depend on other classes, and thus their order is important for a sequential reading by a human. We decided that presenting classes by level in their generalization hierarchies, starting from the root to the leaves, is appropriate. In other words, classes in hierarchies are visited following a Depth First Search (DFS) approach, processing first a node (class) and then all its adjacent (subclasses). As there may be several parallel hierarchies, there may be several root classes. Helper getRootClasses() finds them, and the flatten() version corresponding to Class is called on each class in the result of that call.

The implementation of this latter version of flatten() just realizes the DFS approach; first the "self" class, and then the result of a recursive call to some of its subclasses. Two issues arise here; first why just "some" of the subclasses are included in the search, and second, which is the order of them? The first issue is simple; no leaf class without attributes and navigable associations are interesting to be described per se. The second issue concerns the sorting of "interesting" subclasses before making the recursive call. Given classes A and B, if any ancestor or descendant of B depends in any way (i.e., composes or has a navigable association to) on any descendant of A, then class A must be described before class B. This provides a comparison criterion for sorting sibling classes, which is implemented in the qSort() helper. Note that the root classes described before are sorted using this algorithm too; the order in the roots determines which hierarchy will be described first in the resulting text.

Analogously to the case of Package, the number and contents of sentences produced after a class depend on the form of the class (i.e. weather it owns attributes or not, and so on). Therefore, there are as many rules that transform a Class as combinations of non empty subsequences of {T5, T6, T7, T8}, which totals an amount of fifteen rules.

Finally, a series of helpers is defined for implementing the templates. Unless a component of a template is fixed, which makes a helper unnecessary, there are typically three helpers per template; one for producing the subject, one for producing the verb group, and another for producing the complement.

4.2 From SimpleLanguage to Text

The current implementation slExt of the extractor is quite straightforward. It simply outputs the sentences in the order they appear. Each sentence is formed by concatenating its components and ended with a period symbol. Each component is formed by concatenating its component words, and for each word its value is printed. The complete TCS code of the extractor is given in Fig. 7.

There is a final consideration though. Sentences are usually expressed in active voice, however in some specific cases they may be formed using a variant of passive voice, in which the verb group and the complement are swapped.

5 Paraphrasing Transformations

This section describes the rationale and design of each step of *ParaphrasingTransformations* and discusses current limitations and possible paths of evolution.

```
template Text main
        :       [sentences]
        ;
    template Sentence
        :       subject (activeVoice ?
                        verbGroup complement : complement verbGroup)
                "."
        ;
    template WordGroup abstract
        ;
    template NounPhrase
        :       words
        ;
    template VerbGroup
        :       words
        ;
    template Word
        :       value{as = identifier}
        ;
```

Figure 7: Class templates of extractor

5.1 From ATL to SimpleLanguage

The first step in *ParaphrasingTransformations* transforms an ATL transformation model to a model of our simplified natural language. Transformation ATL2SL is purely syntactical; it literally paraphrases the source transformation without acknowledging the purpose or semantics of pieces of code (e.g., helpers), which goes beyond the scope of this experiment.

5.1.1 Strategy

As before, transformation model ATL2SL defines a set of "template sentences". These templates are associated to classes of source elements. In what follows we provide a mapping between elements in an ATL transformation model and the corresponding templates. The same font color scheme is used again here for highlighting the component parts of a sentence.

Module

T1: "Transformation **TName** produces a[n] $\textbf{OutMM}_1$ model, a[n] $\textbf{OutMM}_2$ model, ... and a[n] $\textbf{OutMM}_n$ model from a[n] $\textbf{InMM}_1$ model, a[n] $\textbf{InMM}_2$ model, ... and a[n] $\textbf{InMM}_m$ model"

This kind of sentence occurs once in the text and describes the signature of the source transformation. TName corresponds to the source transformation name, InMM_i correspond to source reference model names, and OutMM_j correspond to target reference model names.

MatchedRule

T2: "For each **InPatternElem** instance, a[n] **DefOutPatternElem** instance has to be created"

This kind of sentence is expressed in passive voice and occurs once per rule in the text. It is used for describing the signature of a rule. InPatternElem is the type of the source pattern element, and DefOutPatternElem is the type of the default target pattern element.

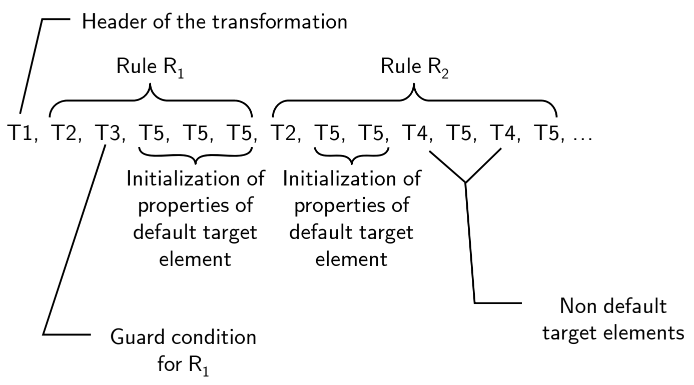
Header of the transformation
Rule R1
Rule R2
T1, T2, T3, T5, T5, T5, T2, T5, T5, T4, T5, T4, T5, ...
Initialization of properties of default target element
Initialization of properties of default target element
Guard condition for R1
Non default target elements

Figure 8: Sequence of sentences in target model

InPattern

T3: "The **InPatternElem** must be **FilterCondition**"

This kind of sentence occurs once per rule, and is used for expressing the guard condition (if applicable) of a rule. InPatternElem again is the type of the source pattern element, and FilterCondition expresses the predicate that constitutes the filter.

SimpleOutPatternElement

T4: "A **NonDefOutPatternElem** instance has to be created"

This kind of sentence occurs once for each non-default target pattern element of each rule. It is used for expressing the creation of target elements other than the default one. NonDefOutPatternElem is the type of the target pattern element.

Binding

T5: "The **OutPatternElemProp** is **BindingExp**"

This kind of sentence is used for expressing the way in which properties of target elements are initialized. If a sentence of this kind follows a sentence of kind T2, then OutPatternElemProp is the name of a property of DefOutPatternElem. If it follows a sentence of kind T4, OutPatternElemProp is the name of a property of the corresponding NonDefOutPatternElem. In either case, BindingExp expresses the initialization of the property.

5.1.2 Design

Not every source element will lead to a sentence in the resulting text. For this reason the transformation first selects the proper source elements that will be transformed. Moreover, selected source elements are flattened in a sequence conveniently so that the order of the elements matches the expected order of the corresponding sentences of the final text. The ordering that is currently implemented is shown in Fig. 8.

The selection (and ordering) process is achieved by a series of helpers named flatten(). If the elements to be selected, and/or their order, is to be altered then new flatten() helpers, or the modification of existing ones, would be required.

Only the type of elements mentioned in the mapping presented before will be contained in the resulting sequence. As a consequence, the transformation will define matched rules only for them. Additionally, note that there is a one-to-one correspondence between rules and template sentences. In particular, each template prescribes how the sentence needs to be formed within the rule. Finally, OCL expressions that represent a filter condition or a binding value are transformed by a series of helpers named process(), each returning a string. A specific process() helper is then required for every variant of OclExpression. The returned string represents the expression in text, and is built using local information of the expression and information of subexpressions as well. The transformation of the filter property of an InPattern and the value property of a Binding lead to FilterCondition and BindingExp, which are the complement of templates T3 and T5 respectively.

5.2 Limitations

In its current state, the implemented prototype is limited on the structure of transformations it can process. In the next section we describe concrete actions to overcome such limitations, and also possible directions for evolving the prototype.

1. Only matched rules are addressed.
2. Source patterns have one single element.
3. Filter conditions apply to that source element only.
4. Only simple target elements are addressed.
5. No local variables are supported.
6. No imperative block is supported.
7. Only a subset of OCL expressions can be processed.
8. Only a few operator and operation calls within OCL expressions can be processed.

Additionally, the prototype does not generate text from the helpers within the source transformation. In principle, the prototype assumes that the name of a helper suffices for expressing its intention, and a *syntactical* description may not provide useful information. However, more detailed experiments should be conducted for clarifying this point.

5.3 Evolution

With respect to the limitations expressed above the prototype may be evolved as follows:

1. Supporting more kinds of rules in a source transformation requires the definition of new rules (e.g., CalledRule2Sentence) and therefore the definition of new templates, and also the update of the flatten() algorithms.
2. Supporting more than one element in source patterns requires the update of rule MatchedRule2Sentence, and therefore the update of the structure of template T2.
3. Supporting a more general filter condition requires the update of InPattern2Sentence and therefore the update of template T3.
4. Supporting iterative target patterns elements requires the modification of rule MatchedRule2Sentence (for handling the case in which the default target pattern element is iterative), and the definition of a new rule (e.g., ForEachOutPatternElement2Sentence) and therefore the definition of a new template.

5. Supporting variable declarations requires the definition of a new rule (e.g., RuleVariableDeclaration2Sentence). As each variable is declared and initialized, they deserve a sentence for its own. Therefore the definition of a corresponding template is required.

6. Supporting imperative blocks requires the definition of new rules (e.g., ActionBlock2Sentence for opening the corresponding text fragment, and a rule for each variant of Statement) and the corresponding templates.

7. Supporting more OCL expressions requires the definition of new process() helpers.

8. Supporting calls to specific operators and operations requires the addition of more cases in the process() helpers associated to OperatorCallExp and OperationCallExp respectively.

As a further improvement, the first step of *ParaphrasingTransformations* may be split into two consecutive substeps. Instead of generating a SimpleLanguage model at once, a first substep may produce a model where only the information to be contained in sentences which is variable is maintained. The associated metamodel could define specific elements, each matching a template. For example, a metaclass for holding the information which will be used for producing a sentence from T1, another metaclass for template T2, and so on. Using such a model, a second substep would produce the expected SimpleLanguage model.

The benefits of this approach are two. First, extraction of the chunks of information from the source transformation is kept separated from the generation of the sentences. Second, dictionaries may be used during the second substep for producing variants of the textual representation of the same template. For example, when generating a sentence from T5 it is possible to choose from one of these concrete forms: "The **OutPatternElemProp** is **BindingExp**", "The **OutPatternElemProp** is initialized with **BindingExp**", or even "The **OutPatternElemProp** has to be set to **BindingExp**". This would make the reading of the resulting text much more pleasant for the human reader.

6 Examples

In this section we show the result of applying *ParaphrasingReferenceModels* and *ParaphrasingTransformations* to the KM3 metamodel and Families2Persons transformation as introduced in Sect. 2. In both cases we present the source models and show the textual result.

6.1 Example of Paraphrasing Reference Models

The following example illustrates the operation of *ParaphrasingReferenceModels*. In this case, the source metamodel is the KM3 metamodel itself. Figure 9 shows the KM3 definition of the KM3 metamodel [2]. The resulting text shown in Fig. 10 corresponds to the KM3Text textual entity, and is conveniently formatted with line breaks for the sake of readability. The prototype actually outputs sentences one right after the other.

6.2 Example of Paraphrasing Transformations

The following example illustrates the operation of *ParaphrasingTransformations*. In this case, the source transformation is the Families2Persons transformation model from the ATL Transformation Zoo [1]. Figure 11 shows its ATL source code. The resulting text, shown in Fig. 12, corresponds to the F2PText textual entity and is again conveniently formatted for the sake of readability.

```
package KM3 {
        abstract class LocatedElement {
                attribute location : String;
        }
        abstract class ModelElement extends LocatedElement {
                attribute name : String;
                reference "package" : Package oppositeOf contents;
        }
        class Classifier extends ModelElement { }
        class DataType extends Classifier { }
        class Enumeration extends Classifier {
                reference literals[*] ordered container : EnumLiteral oppositeOf enum;
        }
        class EnumLiteral extends ModelElement {
                reference enum : Enumeration oppositeOf literals;
        }
        class Class extends Classifier {
                attribute isAbstract : Boolean;
                reference supertypes[*] : Class;
                reference structuralFeatures[*] ordered container : StructuralFeature oppositeOf owner;
        }
        class TypedElement extends ModelElement {
                attribute lower : String;
                attribute upper : String;
                attribute isOrdered : Boolean;
                attribute isUnique : Boolean;
                reference type : Classifier;
        }
        class StructuralFeature extends TypedElement {
                reference owner : Class oppositeOf structuralFeatures;
        }
        class Attribute extends StructuralFeature { }
        class Reference extends StructuralFeature {
                attribute isContainer : Boolean;
                reference opposite[0-1] : Reference;
        }
        class Package extends ModelElement {
                reference contents[*] ordered container : ModelElement oppositeOf "package";
                reference metamodel : Metamodel oppositeOf contents;
        }
        class Metamodel extends LocatedElement {
                reference contents[*] ordered container : Package oppositeOf metamodel;
        }
}

package PrimitiveTypes {
        datatype Boolean;
        datatype Integer;
        datatype String;
}
```

Figure 9: KM3 definition of KM3 metamodel

7 Implementation

Our prototype was implemented using ATL and TCS following the ideas depicted in Figs. 3 and 4 and using the extractor of Fig. 7. An Eclipse workspace containing all source files is available at `http://mate.dcc.uchile.cl/research/tools/paraphrasing`. A megamodel containing all the assets involved in this work, which follows the structure illustrated in Figs. 2, 3 and 4, is also included in the workspace.

8 Conclusions

Textual descriptions of models play an important role in documentations. Such descriptions complement (semi) formal notations with information expressed in natural language, which is understandable by a wider range of human readers. For being effective, the generation of text from models should be carried out properly. The information required to that end is in possession of the authors of the models themselves. This process is often

This metamodel contains packages KM3 and PrimitiveTypes.

Package KM3 contains the following classes only. A LocatedElement is an abstract entity that defines the location attribute. A LocatedElement must be a ModelElement or a Metamodel. A Metamodel is composed of a sequence of Package elements which play the role of contents. A ModelElement is an abstract entity that defines the name attribute. A ModelElement must be a Classifier, an EnumLiteral, a TypedElement or a Package. A Package is composed of a sequence of ModelElement elements which play the role of contents.
A Classifier can be a DataType, an Enumeration or a Class. An Enumeration is composed of a sequence of EnumLiteral elements which play the role of literals. A Class defines the isAbstract attribute. A Class is composed of a sequence of StructuralFeature elements. A Class may have supertypes which correspond to Class elements.
A TypedElement defines the lower, upper, isOrdered and isUnique attributes. A TypedElement can be a StructuralFeature. A TypedElement has a type which corresponds to a Classifier element.
A StructuralFeature can be an Attribute or a Reference. A Reference defines the isContainer attribute. A Reference may have an opposite which corresponds to a Reference element.

Package PrimitiveTypes only contains data type Boolean, Integer and String.

Figure 10: Resulting text from applying *ParaphrasingReferenceModels* to the KM3 metamodel

neglected in practice due to its high cost. Therefore, the automatic generation of these descriptions can relieve modelers from the burden of both creating and maintaining them. In this work we presented a prototype of two model transformations which paraphrase two different types of models: KM3 reference models and ATL transformation models.

The prototype consists of two transformation chains realized as two separate ATL transformations composed with one TCS extractor. If models of another language are to be considered for paraphrasing then a new chain must be created. The extractor may be reused, but a new specific transformation must be provided for the new language.

The complexity of the main transformation of each chain, as expected, depends on the size and complexity of the metamodel of the model to be paraphrased. ATL's metamodel is larger and more complex of that of KM3. In fact, KM32SL was completely implemented, while with the same effort, the implementation of ATL2SL supports a limited number of cases only.

The complexity of those transformations also was affected by the sequential ordering of the generated sentences. As in declarative transformations source elements are processed in an arbitrary order, special care was required for arranging sentences in the appropriate order. This was achieved by flattening source elements into a sequence and rearranging them before processing, leading to complex algorithms as in the case of KM32SL. Furthermore, some source elements that may produce a sentence of their own are optional in a concrete source model (e.g., a rule may not have a guard condition). This caused an increase in the number of rules, since different combinations of similar patterns of source elements needed to be handled.

The value of a textual description of a model relies on the ability to express information which is not straightforwardly extracted from the model, or even information which is not present in it. This information involves the semantics of the model and is usually captured as the intent of some of its elements. That intent is hard to guess from the syntax of a model, and our implementation does not try to do it. For example, it

```
module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: familyName : String =
	if not self.familyFather.oclIsUndefined() then
		self.familyFather.lastName
	else
		if not self.familyMother.oclIsUndefined() then
			self.familyMother.lastName
		else
			if not self.familySon.oclIsUndefined() then
				self.familySon.lastName
			else
				self.familyDaughter.lastName
			endif
		endif
	endif;

helper context Families!Member def: isFemale() : Boolean =
	if not self.familyMother.oclIsUndefined() then
		true
	else
		if not self.familyDaughter.oclIsUndefined() then
			true
		else
			false
		endif
	endif;

rule Member2Male {
	from
		s : Families!Member (not s.isFemale())
	to
		t : Persons!Male (
			fullName <- s.firstName + ' ' + s.familyName
		)
}

rule Member2Female {
	from
		s : Families!Member (s.isFemale())
	to
		t : Persons!Female (
			fullName <- s.firstName + ' ' + s.familyName
		)
}
```

Figure 11: ATL source code of transformation Families2Persons

could be possible to describe the algorithm of the isFemale() helper in Fig. 8, but a text describing its purpose would be of more interest in the context of this experiment. In our prototypical implementation we rely on proper element names in such cases.

Our prototype needs to be improved for handling more cases of source transformation models. In Sects. 5.2 and 5.3 we provided a complete list of the current limitations and concrete actions for overcoming them. Additionally, the usability of our proposal can be enhanced if more semantic information about source models could be included in the resulting text. Possible directions for this include the use of annotations in source models, and also the recognition of patterns of source elements. Finally, splitting the specific transformation into two separate substeps, as discussed in Sect. 5.3, could improve the quality of the resulting text, since different forms of the same template may be more easily used. We think that this improves the quality of the text because it becomes less repetitive. Moreover, although each chain ends up containing three substeps instead of two, each step is conceptually simpler since the concern of extracting information from the source model is separated from the concern of constructing the sentences. The resulting text is currently plain and is presented as a sequence of sentences. Another way to improve it is adding some form of formatting as in the examples in Sect. 6. This could be done by modifying the SimpleLanguage metamodel, and thus all involved transformation, or by modifying the SLExtractor extractor.

Transformation Families2Persons produces a Persons model, from a Families model.

For each Member instance, a Male instance has to be created. The Member must be not female.

- The full name is the first name summed to ‘ ’ summed to the family name.

For each Member instance, a Female instance has to be created. The Member must be female.

- The full name is the first name summed to ‘ ’ summed to the family name.

Figure 12: Resulting text from applying *ParaphrasingTransformations* to Families2Persons

References

[1] ATL Transformations Zoo. Internet: `http://www.eclipse.org/m2m/atl/atlTransformations/`, 2009.

[2] Atlantic Zoo. Internet: `http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/`, 2009.

[3] J. Bézivin and F. Jouault. KM3: a DSL for Metamodel Specification. In *8th IFIP*, pages 171–185, 2006.

[4] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In U. Aßmann, M. Aksit, and A. Rensink, editors, *MDAFA*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2004.

[5] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE*, pages 249–254. ACM, 2006.

[6] F. Jouault and I. Kurtev. Transforming Models with ATL. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.