ICTA: International Conference on Information and Communication Technology and Applications

Information and Communication Technology and Applications

Third International Conference, ICTA 2020, Minna, Nigeria, November 24–27, 2020, Revised Selected Papers

- Editors
- (view affiliations)

- Sanjay Misra
- Bilkisu Muhammad-Bello

Conference proceedings **ICTA 2020**

Part of the Communications in Computer and Information Science book series (CCIS, volume 1350)

Table of contents

1. **Information Science and Technology**

    1. A Scoping Review of the Literature on the Current Mental Health Status of Developers
    Ghaida Albakri, Rahma Bouaziz
    Pages 485-496
    2. Gamifying Users' Learning Experience of Scrum
    Guillermo Rodriguez, Alfredo Teyseyre, Pablo Gonzalez, Sanjay Misra
    Pages 497-509
    3. Visualizing Multilevel Test-to-Code Relations
    Nadera Aljawabrah, Abdallah Qusef, Tamás Gergely, Adhyatmananda Pati
    Pages 510-519
    4. Privacy Preservation in Mobile-Based Learning Systems: Current Trends, Methodologies, Challenges, Opportunities and Future Direction
    Muhammad Kudu Muhammad, Ishaq Oyebisi Oyefolahan, Olayemi Mikail Olaniyi, Ojeniyi Joseph Adebayo

Pages 520-534
5. Drug Verification System Using Quick Response Code
Roseline Oluwaseun Ogundokun, Joseph Bamidele Awotunde, Sanjay Misra, Dennison Oluwatobi Umoru
Pages 535-545
6. A Novel Approach to News Archiving from Newswires
Bilkisu Larai Muhammad-Bello, Mudi Lukman, Mudi Salim
Pages 546-559
7. Mobile Application Software Usability Evaluation: Issues, Methods and Future Research Directions
Blessing Iganya Attah, John Kolo Alhassan, Ishaq Oyebisi Oyefolahan, Sulaimon Adebayo Bashir
Pages 560-573
8. Perception of Social Media Privacy Among Computer Science Students
Adebayo Omotosho, Peace Ayegba, Justice Emuoyibofarhe
Pages 574-587
9. An Efficient Holistic Schema Matching Approach
Aola Yousfi, Moulay Hafid El Yazidi, Ahmed Zellou
Pages 588-601
10. AnnoGram4MD: A Language for Annotating Grammars for High Quality Metamodel Derivation
Hamzat Olanrewaju Aliyu, Oumar Maïga
Pages 602-617
11. Design and Implementation of an IoT Based Baggage Tracking System
Olamilekan Shobayo, Ayobami Olajube, Obina Okoyeigbo, Jesse Ogbonna
Pages 618-631
12. Validation of Computational-Rabi's Driver Training Model for Prime Decision-Making
Rabi Mustapha, Muhammed Auwal Ahmed, Muhammad Aminu Ahmad
Pages 632-644
13. Efficient Approaches to Agile Cost Estimation in Software Industries: A Project-Based Case Study
Shariq Aziz Butt, Sanjay Misra, Diaz-Martinez Jorge Luis, De la Hoz-Franco Emiro
Pages 645-659
14. Efficient Traffic Control System Using Fuzzy Logic with Priority
Ayuba Peter, Babangida Zachariah, Luhutyit Peter Damuut, Sa'adatu Abdulkadir
Pages 660-674
15. An Enhanced WordNet Query Expansion Approach for Ontology Based Information Retrieval System
Enesi Femi Aminu, Ishaq Oyebisi Oyefolahan, Muhammad Bashir Abdullahi, Muhammadu Tajudeen Salaudeen
Pages 675-688
16. Design of a Robotic Wearable Shoes for Locomotion Assistance System
Bala Alhaji Salihu, Lukman Adewale Ajao, Sanusi Adeiza Audu, Blessing Olatunde Abisoye
Pages 689-702
17. Design of Cash Advance Payment System in a Developing Country: A Case Study of First Bank of Nigeria Mortgages Limited
Saka John, Jacob O. Mebawondu, Ajayi O. Olajide, Mebawondu O. Josephine
Pages 703-714
18. Users' Perception of the Telecommunication Technologies Used for Improving Service Delivery at Federal University Libraries in Anambra and Enugu State
Rebecca Chidimma Ojobor
Pages 715-726
19. A Step by Step Guide for Choosing Project Topics and Writing Research Papers in ICT Related Disciplines
Sanjay Misra
Pages 727-744
2. Back Matter

Pages 745-746
PDF

Previous
Page of 3

About these proceedings

Introduction

This book constitutes revised selected papers from the Third International Conference on Information and Communication Technology and Applications, ICTA 2020, held in Minna, Nigeria, in November 2020. Due to the COVID-19 pandemic the conference was held online.

The 67 full papers were carefully reviewed and selected from 234 submissions. The papers are organized in the topical sections on Artificial Intelligence, Big Data and Machine Learning; Information Security Privacy and Trust; Information Science and Technology.

Keywords

artificial intelligence communication channels (information theory) communication systems computer hardware computer networks computer security cryptography data mining data security machine learning network protocols signal processing software design software engineering telecommunication networks telecommunication systems wireless telecommunication systems

Editors and affiliations

- Sanjay Misra 1View author's OrcID profile
- Bilkisu Muhammad-Bello 2View author's OrcID profile

1.Covenant UniversityOtaNigeria
2.Federal University of Technology MinnaMinnaNigeria

Bibliographic information

- Book Title Information and Communication Technology and Applications
- Book Subtitle Third International Conference, ICTA 2020, Minna, Nigeria, November 24–27, 2020, Revised Selected Papers
- Editors Sanjay Misra
  Bilkisu Muhammad-Bello
- Series Title Communications in Computer and Information Science
- Series Abbreviated Title Communic.Comp.Inf.Science
- DOI https://doi.org/10.1007/978-3-030-69143-1

- Publisher Name Springer, Cham
- eBook Packages Computer Science Computer Science (RO)
- Softcover ISBN 978-3-030-69142-4
- eBook ISBN 978-3-030-69143-1
- Series ISSN 1865-0929
- Series E-ISSN 1865-0937
- Edition Number 1
- Number of Pages XXIV, 746
- Number of Illustrations 86 b/w illustrations, 240 illustrations in colour

- Topics Information Systems and Communication Service
  Computer Systems Organization and Communication Networks
  Artificial Intelligence
  Data Structures and Information Theory
  Computing Milieux
  Computer Appl. in Social and Behavioral Sciences

AnnoGram4MD: A Language for Annotating Grammars for High Quality Metamodel Derivation

Hamzat Olanrewaju Aliyu[1] and Oumar Maïga[2]

[1]School of Info. & Comm. Tech., Federal University of Technology, Minna, Nigeria
[2] Université des Sciences, Techniques et Technologies, Bamako, Mali
hamzat.aliyu@futminna.edu.ng, maigabababa78@yahoo.fr

Abstract. The quests for transfers of software artifacts between the model ware and grammar ware technical spaces have increased in recent decades. Particularly, the need to port grammar-based concepts into the model ware space has birthed efforts to synthesise Ecore-based metamodels from Extended Backus Naur Form (EBNF)-based grammars. However, automatic derivation of high-quality metamodels from grammars is still a challenge as existing solutions produce metamodels containing either superfluous classes or anonymous classifiers or both, making the results less useful. AnnoGram4MD addresses these issues by adding special annotations to the grammar as complementary information to guide the derivation algorithm towards producing high-quality metamodels. A comparison of AnnoGram4MD with existing solutions when applied to a sample grammar reduced the number of EClassifiers by 52% and without anonymous EClassifiers in the generated metamodel.

Keywords: Grammar to Metamodel, EBNF to MOF, Reverse Engineering,

1 Introduction

Metamodels and grammars define software languages in the modelware and grammarware technical spaces (TSs) respectively [1, 2]. Thus, "metamodel" and "grammar" occupy equivalent meta positions in the two orthogonal TSs. Researches in Model-Driven Engineering (MDE) [3, 4] have prompted the need to establish equivalences between the two TSs to facilitate the exchange of specified domain concepts between them. In fact, bridging the two TSs is considered a prerequisite to several MDE activities [5] especially in reverse engineering and/or model-driven software evolution [6] where codes are transformed to high-level models for use in the modelware TS. For example, a "semantic-preserving" grammar-metamodel translation is needed to reuse Z language-based specifications[7, 8] in the modelware TS. The three most significant deficiencies of the metamodels obtained using the existing grammar-metamodel translation techniques are:

- Superfluous Eclassifiers (Classes and Enums) and EReferences.
- Presence of poorly named or anonymous Eclassifiers and EReferences.
- Missing elements due to incomplete extraction of concepts from grammars.

To address these issues, we view the challenges from two perspectives:
Difference in the goals of grammars and metamodels: An EBNF grammar [9] specifies both the abstract and concrete syntaxes of a language while a metamodel defines only the former. Thus, the mechanism for deriving metamodels from grammars must be able to filter out those concrete syntax elements that will add noise to the output.
Difference in the details required to specify languages in the two TSs: Grammars use fewer details than metamodels to define clear language elements. e.g., "Exp := Exp1 + Exp1;" is understood once the token Exp1 is defined in the grammar. However, a metmodel must define the roles of each operand wrt the operator "+". Thus, a mechanism to infer a metamodel from the grammar must have a deterministic way to augment the limited information in the grammar to derive a complete metamodel.

This paper presents the Annotated Grammar for Meta-model Derivation (AnnoGram4MD), a language for annotating grammar specifications as directives for automated derivation of "high-quality" metamodels. AnnoGram4MD defines special annotations which guide the metamodel derivation algorithm to filter out “noisy” concrete syntax elements and add extra information where necessary.

Section 2 lays the foundation for subsequent sections and presents a running example to illustrate the approach. AnnoGram4MD's syntax and semantics are presented in Sections 3 and 4 respectively with applications to the running example at different stages. Section 4 discusses the related works before the conclusion in Section 5.

2 Background

2.1 Elements of a Software Language

We assume the reader has a basic knowledge of the elements of a software language specification; hence they are not discussed here due to space constraints. If necessary, interested readers may consults [10-12] for detailed descriptions of the elements.

2.2 Grammars

A (context-free) grammar specifies a language by defining all the keywords and concrete symbols to render its sentences in specified patterns[12]. They are used to define programming languages, and to formalise other string-processing applications. Mathematically, a grammar, G, can be defined as in Equation (1) [13]:

$$G = \langle V, T, P, S \rangle;\ V \cap T = \emptyset. \quad (1)$$

V, T and P are finite sets of variables, terminals and production rules respectively. While $t \in T$ is an irreducible element of the language, $v \in V$ describes an independent entity in the language vocabulary, which is defined recursively in terms of other variables and/or terminals by production rules. $S \in V$ is the root variable called the starting symbol; every other element (variable or terminal) must be reachable from S.

EBNF [9] uses the model in Equation (1) to formally specify grammars. An EBNF description is an unordered list of EBNF (production) rules, each having three parts: a left-hand side (LHS), a right-hand side (RHS), and the special character “:=” (read as "is defined as") separating the two sides. The LHS, a variable $v \in V$, is defined by the RHS which contains elements of $(V \setminus \{S\}) \cup T$ in four major configurations [9]:

- *Sequence*. An ordered list of zero or more variables and/or terminals from left-to-right. For example, the RHS of the rule $v := v_1 t_1 v_2$ is a sequence.
- *Selection*. Two or more independent definitions (choices) separated by the "stroke" character (|) from which exactly one is chosen. e.g., $v := v_1 | t_1 | v_2$.
- *Option*. An element appearing zero or one time at its specified location
- *Repetition*. One or more successive appearances of an element in a rule. *Optional repetition* implies zero or more successive appearances of an element in a rule.

Listing 1 is an excerpt from syntax of the Z specification language [14], which serves as a running example in this paper. The starting variable of the excerpt is *Predicate*.

Listing 1. A sample grammar

```
Predicate:= ∀ SchemaText • Predicate |∃ SchemaText • Predicate
|∃1 SchemaText • Predicate |Predicate1;
Predicate1:= Expression Rel Expression |SchemaRef |Predicate1 ∧ Predicate1
|Predicate1 ∨ Predicate1 |Predicate1 ⟹ Predicate1
|Predicate1 ⟺ Predicate1;
Expression:= Expression InFun Expression |Expression2;
Expression2 := SetExp |⟨/Expression,. . ., Expression/⟩ |⟦/ Expression,. . ., Ex-
pression /⟧ |(Expression, . . ., Expression);
SetExp:={/Expression,. . ., Expression /} |{SchemaText/ • Expression/};
Rel:=equal_to|not_equal|greater_than|less_than|greater_or_equal; | less_or_equal
| member_of | contains;
InFun:=plus|minus|mult|div|mod|union|intersection|difference;
```

2.3 Metamodels

"A metamodel is a model of a modelling language" [1]. It defines the abstract syntax (set of domain concepts and their legal relationships) of a modelling language. A metamodel may be described using a subset of the Unified Modeling Language (UML)'s class diagram [15].

2.4 Annotations

Annotations are special metadata that provide additional information about program/model elements during processing without altering their semantics [16]. They are used in programming languages like Java for documentations and to associate specific properties with program artefacts. In the modelware, annotations are used in UML to aid code syntheses. They are also used sparingly in grammarware to filter undesired grammar elements or add role information to grammar elements [17]. In AnnoGram4MD, we adopt the Java convention [16] to annotate EBNF grammar rules. This convention can be described in general using EBNF as:

`Annotation:= @annotationType[(param₁ = 'value₁',..., paramₙ = 'valueₙ')]`

3 AnnoGram4MD

This section presents the key elements of AnnoGram4MD language specification; i.e., the syntaxes, syntax mapping, semantics domain and semantics mapping.

3.1 Abstract and Concrete Syntaxes

The metamodel in Fig. 1 shows the abstract syntax of AnnoGram4MD. The language combines major grammar concepts with annotations embedded in production rules. Therefore, the abstract syntax has two parts with one describing grammar elements as summarized in Section 2.2 and the other describing the proposed abstract annotations.

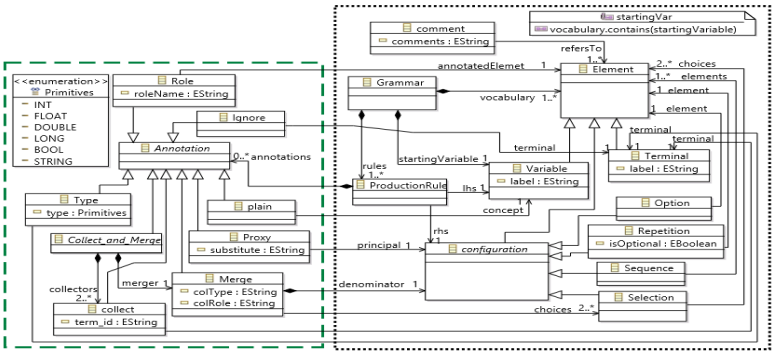


Fig. 1. Abstract syntax of AnnoGram4MD.

Grammar Concepts in AnnoGram4MD. This part is contained within the dotted box on the right side of Fig. 1. A grammar (Class Grammar) has a vocabulary which is a set of named Elements. An element may be a Variable, Terminal or any of the different configurations described previously in Section 2.2.

AnnoGram4MD Annotations. They are specified in the dashed box on the left side of Fig. 1. A productionRule (i.e., in the grammar concepts) may contain some annotations attached to it. The various kinds of annotation in AnnoGram4MD are:

Ignore Annotation. It is used to filter tokens that serve only as concrete syntax elements in the grammar and, as such, are to be excluded from the generated metamodel. Given that a rule defines a set as "$S \coloneqq \{a, b\}$;", metamodelling simply sees S as comprising a and b while all other symbols on the RHS are concrete syntax elements. We represent the annotation as "@ign" preceding a token. In the rule "SetExp" in Listing 1, the curly brackets "{" and "}" in the two choices and the bullet symbol "●" are not needed in the metamodel. These can be removed by annotating the rule as follows:

```
SetExp:=@ign{[Expression,...,Expression]@ign}
|@ign{SchemaText[@ign●Expression]@ign};
```

Role Annotation. It is used to precise the role of an element in a sequence configuration. The grammar only indicates the presence of each element at certain locations in a sequence without any clue its role. This becomes more important when a particular element appears at different positions in the sequence. While the role is intuitive in the grammar, it will be difficult to differentiate them when translated to a metamodel. Therefore, this annotation adds unique role names to variables and terminals where necessary. It is denoted by @role(`roleName'). We illustrate its application on one of the choices in the selection that produces the variable Expression in Listing 1:

```
Expression:=@role('lhs')Expression1 @role('op')InFun @role('rhs')Expression1;
```

In this example, the production rule describes an infix operator "InFun" with two operands each of type "Expression1". When translated into a metamodel, each element in this sequence will be a kind of structural features (e.g., attribute or reference) of class "Expression" which should have a "name" and a "type". Since only the types are explicit in the grammar, this annotation provides the "roleName" to complement the available information for a complete metamodel derivation.

*Plain Annotation.*It indicates variables whose productions are selections of only terminal choices; e.g., "Rel" and "InFun" in Listing 1. Such entities are expressed in metamodels as enumerations with each of the choices as a literal; more details of this will be provided in the semantics. The annotation is used by simply placing @plain before the variable. By applying it to the "Rel" and "InFun" variables, we have:

```
@plainRel:=equal|neq|greater|memberOf|geq|leq|contains;
@plainInFun:=plus|minus|mult|div|mod|union|intersection;
```

*Proxy Annotation.*We can see from Fig. 1 that the outer-most configuration presented by the RHS of a productionRule may embed some other configurations. e.g., embedding sequences within selections. While this enhances the compactness of the grammar, it also leads to shortage of information when translating the grammar to metamodel, thereby leading to the creation of anonymous Eclassifiers in the metamodel. To address this problem, we define a variable, "substitute", through a proxy annotation to replace the embedded configuration while itself (substitute) is defined by the "principal" configuration being replaced. It is applied by preceding a configuration to be replaced with @prox('substitute'). We apply it to the rule SetExp in Listing 1, thus:

```
SetExp:= @prox('EnumeratedSet')[[Expression,...,Expression]]
        |@prox('SetBuilder')[SchemaText[●Expression]];
```

After processing the annotations, the rule will be broken into simpler rules as follows:

```
SetExp := EnumeratedSet|SetBuilder;
EnumeratedSet := [[Expression,...,Expression]];
SetBuilder := [SchemaText[●Expression]];
```

Type Annotation. Recall from Section 2.2 that terminals are indivisible elements in a grammar which are usually expressed in metamodels as attributes of primitive types like integer, boolean etc. Given a grammar describing a domain, a domain expert may intuitively decipher the group to which a terminal belongs but this must be explicitly defined in a metamodel. We use the type annotations to add type information to terminals where necessary. It is denoted by @type('type') preceding a terminal. Given an hypothetical production rule, `Article := name value;` describing an article in a store where terminals "name" and "value" refer to the article's name and price respectively. While a reader can intuitively respectively assign types string and double to the terminals, a transformation algorithm must be told. We apply the annotation as:

```
Article := @type('string')name @type('double')value;
```

Collect-and-Merge Annotation. This is a combo annotation that provides directives for refactoring a particular pattern of definition by providing a simple general equivalent definition for a group of choices in a selection. It is used to tell the parser that a group of two or more choices in a selection configuration can actually be collected and merged into one choice as a common denominator in a metamodel. There are two requirements to be met by all choices in the group to merit being merged:

i. All choices in the group are described by a sequence of same elements in the same order; the only difference being the terminals at one particular position in each sequence

ii. The distinguishing terminals at the specific position must be semantically suitable to be grouped into one category in the domain being described.

For instance, given a production rule describing the mathematical expression as `Exp:=Exp1 + term|Exp1 - term|Exp1 * term|Exp1 div term;`, all the four choices in the selection are expressed as sequences of same elements in same order with the only difference being the middle terminals (+, -, * and div). Interestingly, these terminals can be semantically grouped into a category 'ArithOps', i.e., Arithmetic operators. If we generate a metamodel from this rule as is, it will give four superfluous sub-classes of Exp, each denoting the different operations. However, it is sufficient to merge the four choices into one by replacing the middle terminals with a variable 'ArithOps'. In that way, the rule can be replaced by the following two rules:

```
Exp := Exp1 ArithOps term;
ArithOps := +|-|*|div;
```

Similarly, the first three choices in the production of variable Predicate in Listing 1 typifies this situation; the differences between the three choices are the terminals ∀, ∃ and $\exists_1$ in the first position of each sequence. Semantically, the three terminals belong to the group of "Quantifier" and so can be collected and merged in a variable. Collect-and-Merge combines two annotations, @merge and @col, which are defined as:
@merge(colRole='role',colType='varName',denomintor='replacement')(choice$_1$|...|choice$_n$)
choice$_1$|...|choice$_n$ are the choices to be merged. The collect parts of the annotation, denoted by @col('term$_i$'), identify the unique terminals in their respective choices. The parameter "colType" of @merge specifies the name of the general variable to replace the unique terminals while "colRole" specifies its role name in the refactored grammar and "denominator" defines a general replacement for the merged of choices.

After processing the directives given by this group of annotations, `(choice`$_1$`|...|choice`$_n$`)` will be replaced by @role('role') varName replacement; in the original rule and a new rule varName:= term$_1$|...|term$_n$; is created. We illustrate this combo-annotation by applying it to the production rule of Predicate in Listing 1.

```
Predicate:= @merge(colRole='quantifier',colType ='Quantifier',
denominator='Quantifier SchemaText● Predicate')(@col('forAll')
∀SchemaText ● Predicate|@col('exists') ∃SchemaText ● Predi-
cate |@col('unique') ∃1SchemaText ● Predicate) | Predicate1;
```

The @merge part of the annotation covers the first three choices of the selection configuration while the @col parts provide the identities to distinguish the terminals. Therefore, after processing these directives, the rule will be refactored as follows:

```
Predicate := @role('quantifier') Quantifier SchemaText● Predicate
|Predicate1;    Quantifier := forAll | exists | unique;
```

Consequently, instead of having three superfluous classes - each describing one of the merged choices - in the resulting metamodel, we will have one class with an attribute, quantifier, of type Quantifier while Quantifier itself will generate an enumeration with literals "forAll", "exists" and "unique".

3.2 Case Study

Listing 2 presents the application of appropriate annotations to the entire grammar in Listing 1. Note the applications of multiple annotations on the same elements. The semantics will be provided in the next section.

Listing 2. AnnoGram4MD Annotated Grammar

```
Predicate:=@prox('QuantifiedPred')@merge(colRole = 'quantifier',
colType = 'Quantifier', denominator='Quantifier SchemaText @ign • Predi-
cate')(  @col('forAll')∀ SchemaText • Predicate |@col('exists') ∃ SchemaText
• Predicate |@col('unique') ∃1SchemaText • Predicate ) |Predicate1;
Predicate1:= @prox('SimplePred') @role('lhs')Expression @role('op') Rel
@role('rhs')Expression |SchemaRef
|@prox('ComplexPred')@merge(colRole='connective', colType = 'LogicalCon-
nective', denominator=' @role('lhs')Predicate1 LogicalConnective @role('rhs')
Predicate1)(Predicate1 @col('and') ∧ Predicate1
|Predicate1 @col('or') ∨ Predicate1 |Predicate1 @col('implies') ⟹ Predi-
cate1 |Predicate1 @col('equiv') ⟺ Predicate1);
Expression:= @prox('ArithExp') @role('lhs') Expression @role('op') InFun
@role('rhs')Expression |Expression2;
Expression2 := SetExp |@prox('Sequence') @ign⟨/Expression,. . ., Expression/
@ign⟩ |@prox('Bag') @ign⟦/ Expression,. . ., Expression /@ign⟧
|@prox('Tuple') @ign(Expression, . . ., Expression @ign);
SetExp:=@prox('EnumeratedSet') @ign{/Expression,. . ., Expression /@ign}
|@prox('SetBuilder') @ign{SchemaText/ @ign• Expression/ @ign};
@plainRel:=equal|neq|greater|memberOf|geq|leq|contains;
@plainInFun:=plus|minus|mult|div|mod|union|intersection;
```

3.3 Semantics Domain

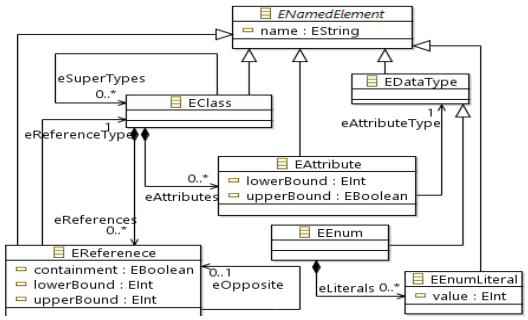


Fig. 2. Abridged Ecore Kernel

The semantics of AnnoGram4MD is based on the Ecore, an implementation of the Object Management Group (OMG)'s Essential Meta-Object Facility (EMOF) [18] for the Eclipse Modeling Framework (EMF) [19]. Ecore contains a part of the UML class that is sufficient to create abstractions of classes and class structures.

Fig. 2 is an abridged Ecore kernel that defines the semantics of AnnoGram4MD. ENamedElement is the base class of all uniquely named elements in a metamodel. EClass describes an independent entity which may have some attributes (eAttributes) and/or references (eReferences) as structural features and may inherit from other Eclasses by referencing them as its superTypes. Every attribute has a type that is defined by a data type or an Enumeration. The allowable limits of occurrence of an attribute or reference in a class are defined by lowerBound and upperBound.

3.4 Semantics Mapping

The semantics mapping occurs in two phases: normalization and derivation. The derivation phase generates metamodel from the normalized annotated grammar by mapping its elements to the corresponding elements of the Ecore.

Normalization. In AnnoGram4MD, a normalized grammar is one in which:

i. There are no concrete syntax-specific symbols except those for recognising configurations and the separators ';' that mark the ends of production rules.
ii. No selection has any other configuration embedded in any of its choices
iii. No sequence has a selection embedded within it
iv. No option or repetition has any other kind of configuration embedded in it
v. No selection contains choices that may be merged into a compact choice

Condition (i) can be satisfied by processing all @ign annotations in the grammar, conditions (ii)-(iv) by processing all @prox annotations while condition (v) is satisfied by processing the @merge-and-@col annotations. Algorithm 1 presents the normalization algorithm to process all occurrences of the aforementioned annotations in the grammar in the normalization phase. A normalization of the annotated grammar in Listing 2 produces a normalized annotated grammar as shown in Listing 3.

Algorithm 1 Normalization Algorithm

```
1: function normalize (AnnotatedGrammar G)
2: for all @ign in G do
3:    delete the annotated token
4: end for
5: for all @merge(colRole, colType, denominator)(choice_1
   |...|choice_n); @col_1('term'_1),...,@col_n('term'_n) in G do
6:    create colType := term_1|term_2|...|term_n
7:    substitute @role('colRole')denominator for
      (choice_1|...|choice_n) in G
8: end for
9: for all @prox('substitute') element in G do
10:   create substitute := element
11:   replace 'element' with 'substitute' in G
12: end for
13: return normalized G
14: end function
```

Listing 3. Normalized Annotated Grammar

```
Predicate := QuantifiedPred | Predicate1;
QuantifiedPred:=@role('quantifier') Quantifier SchemaText Predicate;
Quantifier:= forAll | exists |unique;
Predicate1 := SimplePred | SchemaRef | ComplexPred;
SimplePred := @role('lhs')Expression @role('op') Rel @role('rhs')Expression;
ComplexPred:= @role('lhs') Predicate1 @role('connective') LogicalConnective @role('rhs') Predicate1;
LogicalConnective := and | or | implies | equiv;
Expression := ArithExp | Number| Expression2;
ArithExp := @role('lhs')Expression @role('op')InFun @role('rhs')Expression;
Expression2 := SetExp| Sequence | Bag | Tuple;
Sequence:= [Expression,..., Expression];
Bag:= [Expression,..., Expression];
Tuple := [Expression,..., Expression];
SetExp := EnumeratedSet | SetBuilder;
EnumeratedSet:= [Expression,..., Expression];
SetBuilder := SchemaText [Expression];
@plainRel:=equal|neq|greater|memberOf|geq|leq|contains;
@plainInFun:=plus|minus|mult|div|mod|union|intersection;
```

Derivation. The derivation phase maps the elements of the normalized annotated grammar obtained in the previous phase to Ecore elements to generate metmodels. The derivation process follows two general rules:

Table 1. Mapping table for productions of class variables *Assume L to be the EClass generated for variables

Configuration	Element	Annotation	Ecore Element
	class variable C	@role(“rolename”)	A reference of L with name "rolename", cardinality '1..1' and type C. If no @role annotation, reference name same as C in lowercase.
Sequence	enum variable E	@role(“rolename”)	An attribute of L with name "rolename", cardinality '1..1' and type E. If no @role annotation, reference name as L in lowercase.
	terminal T	@role(“rolename”) @type(“typename”)	An attribute of L with name "rolename", cardinality '1..1' and type @type("typename"). If no @role annotation, reference name as T in lowercase.
Selection	$C_1\|C_2\|...\|C_n$	N/A	EClasses C_1, C_2, ..., C_n are subclasses of L
Option	Same as in Sequence but with cardinality '0..1'		
Repetition	Same as in Sequence but with cardinality '1..*'		
Optional repetition	Same as in Sequence but with cardinality '0..*'		

i. Every lhs variable with @plain annotation translates into an EEnum such that the choices at the rhs of the production become the eLiterals of the EEnum.

ii. Every lhs variable without a @plain annotation translates into an EClass.

Table 1 provides a summary of how the rhs elements are translated to build the EClass obtained from the lhs following from these rules, henceforth; we refer to variables with and without the @plain annotation as enum variables and class variables respectively. In Table 1, `L' represents the EClass generated from the class variable on the lhs of the production rule.

Algorithm 2 gives the details about the implementation of the rules in the table. Function $derive\ (Variable\ V, AnnotatedGrammar\ G)$ takes two parameters; V and G (where V is a variable in G) and returns a metamodel EClass or EEnum derived from the description of V in G. In the process of building the metamodel representation of a variable, the function recursively builds the corresponding metamodel elements of all variables and terminals that define its production rule. The result obtained by processing the normalized grammar in Listing 1.3 with the derivation algorithm is shown in Fig. 3(a). The R@role and R@type clauses in the derive function refer to the information attached to R by the @role and @type annotations respectively.

There are other smaller functions called within the function derive() which we cannot provide their detailed definitions due to space constraint. We will however provide brief descriptions. newClass(V) returns an EClass named 'V' if already created, it creates and returns a new class named 'V' if otherwise. newEnum(V) is similar to newClass(V) except that it returns an EEnum instead. addAttribute(C, word, T, mult) adds an attribute 'word' of type 'T' with cardinality 'mult' to EClass 'C'. addLiteral (E, word) adds an eLiteral named 'word' to an EEnum 'E'. connectSubClass(C1, C2) creates an inheritance relationship between classes C1 and C2 with the former as the superType of the later. Finally, compose(C1, C2, role, mult) creates a composition

association named 'role' with cardinality 'mult' between classes 'C1' and 'C2' with the former as the container. Algorithm 3 describes the main transformation function that cascades the normalization and derivation processes. It takes a raw annotated grammar and its starting variable as input to generate an equivalent metamodel.

Algorithm 2 Derivation Algorithm

```
1: function derive (Var V, AnnotatedGrammar G)
2: V := R ← getRule(V, G) {get the definition of V in G}
3: if R is a selection (R = R1|R2|...|Rn) then
4:   if V has @plain annotation then
5:     M ← newEnum(V) {create empty enum V}
6:     for all Ri ∈ R1, R2, ..., Rn do
7:       addLiteral(M, Ri)
8:     end for
9:   else {i.e., R1, ..., Rn are all variables}
10:    M ← newClass(V) {create empty class V.}
11:    for all Ri ∈ R1, R2, ..., Rn do
12:      connectSubClass(M, derive(Ri))
13:    end for
14:  end if
15:  return M
16: else
17:  M ← newClass(V) {create empty class named V}
18:  if R is a sequence (R = R1, R2, ..., Rn) then
19:    for all Ri ∈ R1, R2, ..., Rn do
20:      if Ri is a repetition (Ri = (R')+) then
21:        if R' is a terminal then
22:          addAttribute(M, R', R'@type,' mult')
23:        else if R' is an enum variable then
24:          addAttribute(M, Ri@role, R',' mult')
25:        else { i.e., R' is a class variable}
26:          compose(M, derive(R', G), Ri@role,' mult')
27:        end if
28:      else if Ri is an optional repetition (Ri = (R')*) then
29:        if R' is a terminal then
30:          addAttribute(M, R', R'@type,' optMult')
31:        else if R' is an enum variable then
32:          addAttribute(M, Ri@role, R',' optMult')
33:        else { i.e., R' is a class variable}
34:          compose(M, derive(R', G), Ri@role, 'optMult')
35:        end if
36:      else if Ri is an option (Ri = (R')?) then
37:        if R' is a terminal then
38:          addAttribute(M, R', R'@type,' opt')
39:        else if R' is an enum variable then
40:          addAttribute(M, Ri@role, R',' opt')
41:        else { i.e., R' is a class variable}
42:          compose(M, derive(R', G), Ri@role,' opt')
43:        end if
44:      else
45:        if R' is a terminal then
46:          addAttribute(M, R', R'@type,' single')
47:        else if R' is an enum variable then
48:          addAttribute(M, Ri@role, R',' single')
49:        else {i.e., R' is a class variable}
50:          compose(M, derive(R', G), Ri@role,' single')
51:        end if
52:      end if
53:    end for
54:  else if R is a repetition (R = (R')+) then
55:    if R' is a terminal then
56:      addAttribute(M, R', R'@type,' mult')
57:    else if R' is an enum variable then
58:      addAttribute(M, R'@role, R',' mult')
59:    else {i.e., R' is a class variable}
60:      compose(M, derive(R', G), R'@role,' mult')
61:    end if
62:  else if R is an optional repetition (R = (R')*) then
63:    if R' is a terminal then
64:      addAttribute(M, R', R'@type,' optMult')
65:    else if R' is an enum variable then
66:      addAttribute(M, R'@role, R',' optMult')
67:    else { i.e., R' is a class variable}
68:      compose(M, derive(R', G), R'@role,' optMult')
69:    end if
70:  else if R is an option (R = (R')?) then
71:    if R' is a terminal then
72:      addAttribute(M, R', R'@type,' opt')
73:    else if R' is an enum variable then
74:      addAttribute(M, R'@role, R',' opt')
75:    else {i.e., R' is a class variable}
76:      compose(M, derive(R', G), R'@role,' opt')
77:    end if
78:  else if R is a terminal then
79:    addAttribute(M, R, R@type,' single')
80:  else
81:    Invalid grammar
82:  end if
83:  return M
84: end if
85: endfunction
```

Algorithm 3 Main Transformation Algorithm

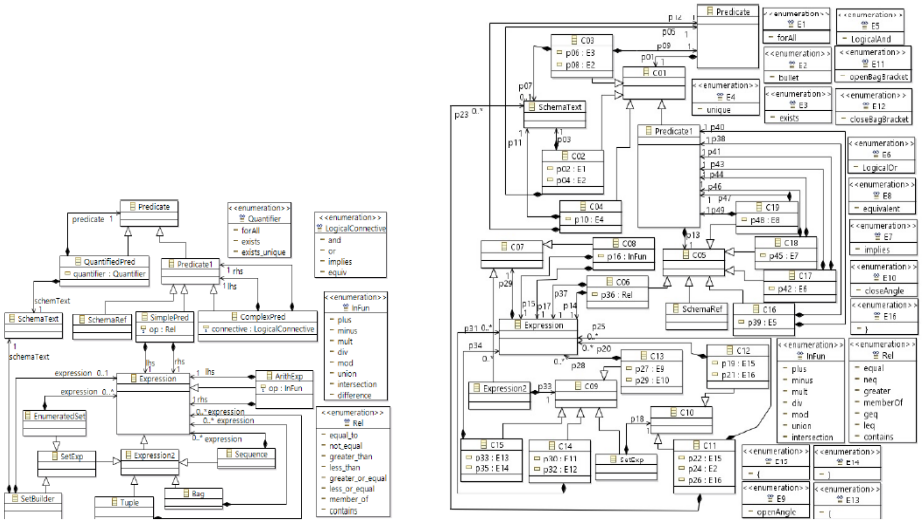
```
1: function transform (Var S AnnotatedGrammar G) {S is the starting variale of G}
2: Gn ← normalize(G)
3: Metamodel_G ← derive(S, Gn)
4: return Metamodel_G
5: endfunction
```

4 Related Works

Alanen and Porres [20] made one of the earliest proposals that gave impetus to further studies of this subject. The idea is to map each token in a sequence configuration to ordered unidirectional composite properties (references and attributes). Properties generated for successive tokens in a sequence are numbered in increasing alphabetical orders to document their order of occurrences in the source grammar. The rules for

generating the cardinalities of such properties are similar to that used in our approach. For every terminal symbol (including the concrete syntax-specific elements), an enumeration is created having the string value of the terminal as its only literal, then an attribute is added to the class with this enumeration as its type. When a selection configuration is encountered, a class is generated with an automatically generated nomenclature and it is sub classed by the classes generated for all choices in the selection. Similar automatically-named classes are generated for repetitions and options which are then connected to the corresponding classes under them according to the rule for sequence configuration. An attempt to use this approach has shown that it will produce a metamodel that is very difficult to (re)use. There are issues such superfluous classes, properties and enumeration. Another aspect that needs improvement in this approach is the area of nomenclature of metamodel elements which is also acknowledged by the authors; the use of alphabets in ascending order as names of metamodel elements will not help the user to understand the domain being modeled. Using this approach, the metamodel derived from the grammar in Listing 1. is shown in Fig. 3(b). When compared with the metamodel generated by AnnoGram4MD in Fig. 3(a), we observe that the latter generated a total of 20 classes and enums compared to 42 in the latter; that is about 52.4% reduction in the size of the output. Moreover, out of the 42 classifiers in Fig. 3(b), only 9 have comprehensible names while others bear some anonymously generated identities. This is also compounded by the generated references, which all have anonymous identities.

AnnoGramm4MD proffers solutions to these drawbacks as shown in the Fig. 3(a) with the metamodel elements bearing names extracted from the source grammar and the embedded annotations. Due to space constraint, we cannot do side-by-side comparisons of the metamodel generated by AnnoGramm4MD and every other work in the literature; nevertheless, we present as much of such comparisons as possible in the rest of this section.



(a) AnnoGram4MD's solution (b) Alanen and Porres' solution

Fig. 1. Derived metamodels from grammars

Wimmer and Kramler [21] have also proposed solutions to some of the drawbacks of [20]. They proposed a three-stage process to generate high-quality MOF-based metamodels from EBNF grammars. The first stage, parsing stage, generates what is referred to as a raw metamodel from a given EBNF grammar. It generates a stereotyped class for every kind of configuration encountered (e.g., sequence, repetition) that has the appropriate references to the metamodel elements generated from the grammar tokens they encapsulate.

Having recognized the possibility of the raw metamodel having exaggerated number of classes, the second - optimization- stage removes undesired elements while documenting the changes made in a "change model". The result obtained from the second stage is called the “condensed metamodel”.

The third - customization - stage adds annotations to the condensed metamodel to provide additional semantics that are not expressed in the original EBNF grammar. From the annotated condensed metamodel, a “customized” metamodel is automatically generated which is considered to be of high quality.

While this is an interesting approach especially considering the fact that it uses the native annotation techniques in the target MOF, we are of the opinion that unless a strict measure is taken, the user may have to redo the annotation in the event that the grammar (source) changes and existing annotations are overwritten during regeneration. Moreover, the user will most likely have to redo the optimization and customization processes as many times as changes are made in the grammar. AnnoGram4MD add annotations to the grammar itself and automates all other processes to avoid the possible situations of repeated work at the different stages when some changes are made in the grammar since they will all be automated. Another advantage of doing the annotation at the source is that it allows for building the metamodel directly from the knowledge of grammar; even someone with only the knowledge of grammars can play with the annotations and obtain a usable metamodel of the domain.

A more recent approach by Kunert [17] proposed another interesting multistage solution. The author is also of the opinion that it will be more convenient to add abstract concepts directly to the grammar as the user will have only one source file to contend with; an opinion that arguably gives more credence to the approach proposed in the current paper. Kunert [17] however did not propose a complete solution to this hypothetical problem. Though the paper also uses annotations in the grammar, its use is limited to identifying grammar specific concrete syntax elements that are not required in a metmodel; it used a special character `!' to annotate concrete syntax elements such as delimiters and identifiers in the grammar to prevent them from being transferred to the generated metamodel. The annotated grammar is then fed into a parent compiler that produces what is called the “simple metamodel" which is considered to be of low quality. The simple metamodel is processed further in a second stage by removing unwanted classes and providing additional annotations to provide information such as alternative class names to produce a “good metamodel". The paper identifies the need for annotating only grammars and automating all other processes, though it did not provide a complete solution to the problem. It, however, provides the motivation for further contributions such as AnnoGram4MD.

5 Conclusions

We have presented the AnnoGram4MD, a language that formally blends java-like annotations with grammar concepts to facilitate the addition of complementary information to EBNF-based grammars for automated synthesis of equivalent domain metamodel of high quality. This has become necessary particularly to facilitate the reuse of grammar-based formal specification languages in the MDE environments. This paper documents the syntax and semantics of the language as well as a case study to illustrate its usability in a step-by-step application of the different techniques. AnnoGram4MD offers the means to underscore grammar elements that are not desired in metamodels as well as add domain-specific information not captured in the grammar concepts but required to build useful metamodels while the metamodel derivation process can be completely automated.

It is however important to state here that our current solution does not directly provide support for adding information that could be used to derive constraints for static semantics as is sometimes required to complement a metamodel. But it supports the derivation of usable Ecore-based metamodels. Moreover, unlike most of the existing solutions, there is currently no support for bidirectional transformation between grammars and metamodel though our grammar to metamodel track claims some important advantages compared to many other proposals. We believe the documentation provided in this paper can serve as guide towards the implementation of supporting tools for the language.

References

1. Favre, J.-M. *Foundations of meta-pyramids: Languages vs. metamodels--episode ii: Story of thotus the baboon1*. in *Dagstuhl Seminar Proceedings*. 2005. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
2. Favre, J.-M. *Towards a basic theory to model model driven engineering*. in *3rd Workshop in Software Model Engineering*. 2004.
3. Schmidt, D.C., *Model-driven engineering.* Computer-IEEE Computer Society-, 2006. **39**(2): p. 25.
4. Brambilla, M., J. Cabot, and M. Wimmer, *Model-driven software engineering in practice.* Synthesis lectures on software engineering, 2017. **3**(1): p. 1-207.
5. Bergmayr, A. and M. Wimmer. *Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques*. in *First International Workshop on Model-driven Engineering By Example*. 2013.
6. Izquierdo, J.L.C. and J.G. Molina, *Extracting models from source code in software modernization.* Software & Systems Modeling, 2014. **13**(2): p. 713-734.
7. Smith, G., *The Object-Z specification language*. Vol. 1. 2012: Springer Science & Business Media.
8. Spivey, J.M. and J. Abrial, *The Z notation*. 1992: Prentice Hall Hemel Hempstead.
9. Feynman, R., *EBNF: A Notation to Describe Syntax*. Режим доступа: http://www. ics. uci. edu/~ pattis/misc/ebnf2. pdf. 2016. 1-19.

10. Kolovos, D.S., et al. *Bridging the Epsilon Wizard Language and the Eclipse Graphical Modeling Framework*. in *Modeling Symposium, Eclipse Summit Europe, Ludwigsburg, Germany*. 2007.
11. Kleppe, A. *A language description is more than a metamodel*. in *Fourth international workshop on software language engineering*. 2007. megaplanet. org.
12. Kleppe, A., *Software language engineering: creating domain-specific languages using metamodels*. 2008: Pearson Education.
13. Hopcroft, J.E., R. Motwani, and J.D. Ullman, *Introduction to automata theory, languages, and computation.* Acm Sigact News, 2001. **32**(1): p. 60-65.
14. Spivey, J.M., *Understanding Z: a specification language and its formal semantics*. Vol. 3. 1988: Cambridge University Press.
15. Booch, G., *The unified modeling language user guide*. 2005: Pearson Education.
16. Cazzola, W. and E. Vacchi, *@ Java: Bringing a richer annotation model to Java.* Computer Languages, Systems & Structures, 2014. **40**(1): p. 2-18.
17. Kunert, A., *Semi-automatic generation of metamodels and models from grammars and programs.* Electronic Notes in Theoretical Computer Science, 2008. **211**: p. 111-119.
18. OMG, *OMG Meta Object Facility (MOF) Core Specification*. 2019, Object Management Group.
19. Steinberg, D., et al., *EMF: eclipse modeling framework*. 2008: Pearson Education.
20. Alanen, M. and I. Porres, *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels.* 2003.
21. Wimmer, M. and G. Kramler. *Bridging grammarware and modelware*. in *International Conference on Model Driven Engineering Languages and Systems*. 2005. Springer.