Zhu, Hong
On The Theoretical Foundation of Meta-Modelling in Graphically Extended BNF and First Order Logic.

Zhu, Hong (2010) On The Theoretical Foundation of Meta-Modelling in Graphically Extended BNF and First Order Logic. In: *Prof. of the 3rd International Symposium on Theoretical Aspects of Software Engineering (TASE 2010),* , IEEE CS Press. pp. 95-104 .
Doi: 10.1109/TASE.2010.11

2010 Fourth International Symposium on Theoretical Aspects of Software Engineering

On The Theoretical Foundation of Meta-Modelling in Graphically Extended BNF and First Order Logic

Hong Zhu
Department of Computing and Electronics, Oxford Brookes University,
Oxford OX33 1HX, UK. Email: hzhu@brookes.ac.uk

***Abstract*—Meta-modeling plays an important role in model driven software development methodology. In our previous work, a graphic extension of BNF (GEBNF) was proposed to define the abstract syntax of graphic modeling languages. From a GEBNF syntax definition, a first order predicate logic language can be induced so that meta-modeling can be performed formally by specifying a predicate on the domain of syntactically valid models. In this paper, we investigate the theoretical foundation of this meta-modeling approach. We first formally define the semantics of GEBNF syntax definitions as algebras that contain no junk and satisfy constraints derived from GEBNF syntax rules. The semantics of the induced logic is then formally defined by regarding such algebras as models. We then formally prove that well-formed syntax definitions together with syntax morphisms form a category, where syntax morphisms represent the translations between modeling languages. The models (i.e. algebras) in a modeling language and the homomorphisms between them also form a category. Finally, we prove that the functors from GEBNF syntax definitions to the categories of models and to sentences in the induced first order logic form an institution. Therefore, GEBNF and its induced logics form a valid formal specification language for models.**

***Keywords*-Modeling languages; Meta-modeling; Syntax and semantics, Formal specification; Institutions; Predicate logic; Graphic modeling; Algebra.**

I. Introduction

Meta-modeling is to model models, i.e. to define a set of models that have certain structural and/or behavioral features. It play three key roles, or a combination of them, in model-driven software development methodologies.

Firstly, meta-modeling defines a modeling language by specifying the syntax, usually at the abstract syntax level, and the semantics, usually in the form of a set of basic concepts underlying the models and their interrelationships. For example, the meta-model for UML defines the abstract syntax of UML modeling language in a class diagram by defining a set of concepts represented as meta-classes and the relationships between them in association, inheritance and aggregation relations between meta-classes [1].

Secondly, it imposes restrictions on an existing modeling language so that only a subset of the syntactically valid models are considered as its instances. For example, specifying design patterns is widely considered as a meta-modeling problem. Each design pattern can be defined by a meta-model so that only those design models that are instances of the meta-model conform to the design pattern [2], [3].

Finally, meta-modeling also extends an existing meta-model by introducing new concepts and defining how the new concepts are related to the existing ones. For example, platform specific models can be defined through introducing platform specific model elements. Aspect-oriented modeling can be defined by a meta-model that extends UML meta-model with basic concepts of aspect-orientation, such as crosscut points, etc. [4].

Due to the importance of meta-modeling, growing research efforts on meta-modeling have been made in the past few years. In our previous work, we proposed a formal meta-modeling approach, which includes a meta-notation called GEBNF, which stands for Graphic Extension of BNF, for the definition of abstract syntax of modeling languages, and a theory and technique that induce first order predicate logic languages (FOL) from GEBNF syntax definitions [5]. In this FOL, constraints on models can be specified and inferred formally and supported by automatic or interactive inference engines. A non-trivial subset of UML, including class diagrams and sequence diagrams, has been defined in GEBNF, and all the design patterns in the Gang-of-Four book [6] have been formally specified as meta-models using the induced FOL [7], [9]. An design pattern recognition tool LAMBDES-DP based on the formal specification has been developed successfully by employing the theorem prover SPASS [10]. Reasoning about meta-models, such as proving a design pattern A is a sub-pattern of B and composition of patterns, has also been explored [8]. In this paper, we further advance the approach by laying a solid theoretical foundation through a formal semantics of GEBNF meta-notation and its induced logic.

The paper is organized as follows. Section II introduces the GEBNF meta-modeling approach. Section III investigates how syntactic constraints imposed by GEBNF meta-notation can be represented as predicates in the induced FOL. Section IV formally defines the semantics of GEBNF and its induced FOL by applying the model theory of mathematical logic. Section V studies the theoretical properties of GEBNF and its induced logic systems in the framework of institution theory. Finally, section VI concludes the paper with a discussion of related works and future work.

978-0-7695-4148-8/10 $26.00 © 2010 IEEE
DOI 10.1109/TASE.2010.11

II. Overview of GEBNF

In this section, we introduce the meta-notation GEBNF and the FOL induced from GEBNF syntax definitions.

A. The Meta-Notation

Similar to the syntax definitions of programming languages in BNF, a syntax definition of a modeling language in GEBNF consists of a set of syntax rules that contains non-terminal symbols and terminal symbols. The extensions that GEBNF brings to BNF are twofold. The first is *field names*, which enable a set of function symbols to be deduced from a syntax definition to form a signature of a FOL. The second is the facility for *referential occurrences* of non-terminal symbols so that non-linear structures like graphs can be defined.

Definition 1 (GEBNF meta-notation): In GEBNF, the abstract syntax of a modeling language is defined as a tuple $\langle R, N, T, S\rangle$, where N is a finite set of non-terminal symbols, and T is a finite set of terminal symbols. Each terminal symbol represents a set of atomic elements that may occur in a model. $R \in N$ is the root symbol and S is a finite set of syntax rules in the form of

$$Y ::= f_1 : X_1, f_2 : X_2, \cdots, f_n : X_n,$$

where $Y \in N$, f_1, f_2 , $\cdots$, f_n are called *field names*, and X_1, X_2 , $\cdots$, X_n are the fields. Each field can be an expression, which is inductively defined as follows.

- C is an expression, if C is a literal constant of a terminal symbol, such as string or number.
- Y is an expression, if $Y \in N \cup T$.
- $\underline{Y}@Z.f$ is an expression, if $Y, Z \in N$, and f is a field name in the definition of Z, and Y is the type of f field in Z's definition. The non-terminal symbol Y in the expressions $\underline{Y}@Z.f$ is a *referential occurrence*.
- Y^*, Y^+ and $[Y]$ are expressions, if $Y \in N \cup T$.
- $Y_1 \mid Y_2 \mid \cdots \mid Y_n$ is an expression, if $Y_1, Y_2, \cdots, Y_n \in N \cup T$. □

The meaning of the meta-notation is explained in Table I. Informally, each terminal and non-terminal symbol denotes a type of elements. Terminal symbols denote the basic atomic element; like $String$ denotes the set of strings. Non-terminal symbols denote the constructs of the modeling language. The elements of the root symbol are the models of the language.

If a non-terminal symbol Y is defined as

$$Y ::= f_1 : X_1, \cdots, f_n : X_n,$$

then, Y denotes a type of elements that each consists of n elements of type X_1, $\cdots$, X_n, respectively. In other words, each element of type Y is constructed from n elements of type $X_1, \cdots, X_n$, respectively. The k'th element in the tuple can be accessed through the field name f_k, for $1 \leq k \leq n$, and we write $a.f_k$ for the k'th element of a, if a is an element of type Y.

Note that the original GEBNF notation for referential occurrences of non-terminal symbols is in the form of $\underline{Y}$ It does not specify which occurrence of the non-terminal symbol it refers to. This may cause ambiguity. Thus, in this paper, we revised the notation. The original notation $\underline{Y}$ can be considered as a short form of $\underline{Y}@Z.f$ when there is no risk of ambiguity, i.e. when there is only one non-referential occurrence of Y in the syntax definition. Moreover, one can also write $\underline{Y}@Z$ as a short form of $\underline{Y}@Z.f$ when there is only one non-referential occurrence of Y in the definition of Z.

Example 1 (Directed graphs): The following is a definition of the abstract syntax of directed graphs in GEBNF. We will use it throughout the paper to illustrate the notions and notations introduced in this paper.

$$\begin{array}{rcl} Graph & ::= & nodes : Node^+,\ edges : Edge^* \\ Node & ::= & name : String,\ weight : [Real] \\ Edge & ::= & from : \underline{Node@Graph.nodes}, \\ & & to : \underline{Node@Graph.notes},\ weight : Real \end{array}$$

where $Graph$ is the root symbol. $Graph, Node$ and $Edge$ are non-terminal symbols, and $String$ and $Real$ are terminal symbols.

The syntax rules state that a graph consists of a non-empty set of nodes and a set of edges. Each node has a name, which is a string of characters, and may have an optional weight, which is a real number. Each edge is from one node to another, which refer to the nodes in the same graph. And, each edge has a weight, which is a real number. □

B. Well-Formed Syntax Definitions

If a symbol $X \in T \cup N$ occurs on the right-hand side of the definition of non-terminal symbol Y, we say that X is *directly reachable* from Y through a field name. For example, $Node$ and $Edge$ are directly reachable from $Graph$. We define the *reachable* relation as the transitive closure of the directly reachable relation.

If there is a non-terminal symbol that is not reachable from the root symbol R, its elements do not play any role in the construction of any model. Such cases should not occur in a well defined syntax. Similarly, we do not want a non-terminal symbol to be used but not defined, or to be defined more than once. Thus, we have the following notion of well-formed syntax.

Definition 2 (Well-Formed Syntax Definition): A syntax definition $\langle R, N, T, S\rangle$ in GEBNF is *well-formed* if it satisfies the following two conditions.

1) *Completeness*. For each non-terminal symbol $X \in N$ there is one and only one syntax rule $s \in S$ that defines X, i.e. of which X is the left-hand-side.
2) *Reachability*. For each non-terminal symbol $X \in N$ X is reachable from the root R. □

Table I
MEANINGS OF THE GEBNF NOTATION

Notation	Meaning	Example	Explanation
$X*$	A set of elements of type X.	$Model ::= diags : Diagram^*$	A model consists of a number N of diagrams, where $N \geq 0$.
X^+	A non-empty set of elements of type X.	$Model ::= diags : Diagram^+$	A model consists of a number N of diagrams, where $N \geq 1$.
$[X]$	An optional element of type X.	$StickFig ::= actor : [Actor]$	A $StickFig$ has an optional element of type $Actor$.
$\underline{X}@Z.f$	A reference to an existing element of type X in field f of an element of type Z.	$Assoc ::= end : \underline{ClassNode}$ $@ClassDiag.classes$	An association has an end that refers to an existing class node in the field $classes$ of $ClassDiag$.
$X_1 \vert \cdots \vert X_n$	An element of type X_1 or, $\cdots$, or X_n.	$Node ::= Actor \vert UseCase$	A node is either an actor or a use case.

Obviously, the syntax of directed graphs given above is well-formed.

C. Induced First-Order Language

Consider the syntax definition of directed graphs given in Example 1. The first syntax rule introduces two field names $nodes$ and $edges$. They can be regarded as two functions that mapping from a graph to two types of elements in the graph, i.e. its non-empty set of nodes and the set of edges, respectively. That is, if g is a graph, then $g.nodes$ is the set of nodes in g.

In general, every field $f : X$ in the definition of a symbol Y introduces a function $f : Y \to X$. Function application is written $a.f$ for function f and argument a of type Y. Given a well-formed syntax, a set of function symbols and their types can be derived as follows.

First, we define the types of expressions and symbols.

1) For all $s \in T \cup N$, s is a type, which is called a *basic type*.
2) $\mathcal{P}(\tau)$ is a type, called *the power type* of τ, if τ is a type.
3) $\tau_1 + \cdots + \tau_n$ is a type, called the *disjoint union* of $\tau_1, \cdots \tau_n$ for $n > 1$, if $\tau_1 \cdots \tau_n$ are types. We also write $\sum_{i=1}^{n} \tau_i$ to denote $\tau_1 + \cdots \tau_n$.
4) $\tau_1 \to \tau_2$ is a type, called a *function type* from τ_1 to τ_2, if τ_1 and τ_2 are types.

Definition 3 (Induced functions): A syntax rule "$A ::= \cdots, f : B, \cdots$" introduces a function symbol f of type $A \to T(B)$, where $T(B)$ is defined as follows.

- $T(C) = C$, if $C \in T \cup N$;
- $T([C]) = T(C)$;
- $T(\underline{C}@Z.f) = T(C)$;
- $T(C^*) = \mathcal{P}(T(C))$;
- $T(C^+) = \mathcal{P}(T(C))$;
- $T(C_1 \vert \cdots \vert C_n) = \sum_{i=1}^{n} (T(C_i))$. □

Example 2 (Induced Functions): For example, the functions induced from the GEBNF syntax definition of directed graphs and their types are given in Table II. □

Given a well-defined GEBNF syntax $\mathbf{G} = \langle R, N, T, S \rangle$ of a modeling language $\mathcal{L}$, we write $Fun(\mathbf{G})$ to denote the set of function symbols derived from the syntax rules. From $Fun(\mathbf{G})$, a FOL can be defined as usual using variables, relations and operators on sets, relations and operators on basic data types denoted by terminal symbols, equality and logic connectives *or* $\vee$, *and* $\wedge$, *not* $\neg$, *implication* $\to$ and *equivalent* $\equiv$, and quantifiers *for all* $\forall$ and *exists* $\exists$ [11].

Table II
EXAMPLE: INDUCED FUNCTIONS OF DIRECTED GRAPHS

Function	Type
nodes	$Graph \to \mathcal{P}(Node)$
edges	$Graph \to \mathcal{P}(Edge)$
name	$Node \to String$
weight	$Node \to Real$
from	$Edge \to Node$
to	$Edge \to Node$
weight	$Edge \to Real$

In particular, assume that for each terminal symbol $s \in T$, there is a set Op_s of operator symbols and a set R_s of relational symbols defined on s. These operation and relation symbols can be used in the predicates on models. The FOL induced from a GEBNF syntax definition can be defined inductively as follows.

Let $V = \bigcup_{s \in N \cup T} V_s$ be a set of variables, where $x \in V_s$ are variables of type s.

1) Each literal constant c of type $s \in T$ is an expression of type s.
2) Each element v in V_s, i.e. variable of type s, is an expression of type s, where $s \in T \cup N$.
3) $e.f$ is an expression of type τ', if f is a function symbol of type $\tau \to \tau'$, e is an expression of type τ.
4) $\{e(x) \vert Pred(x)\}$ is an expression of type $\mathcal{P}(\tau_e)$, if x is a variable of type τ_x, $e(x)$ is an expression of type τ_e and $Pred(x)$ is a predicate on type τ_x.
5) $e_1 \cup e_2$, $e_1 \cap e_2$, and $e_1 - e_2$ are expressions of type $\mathcal{P}(\tau)$, if e_1 and e_2 are expressions of type $\mathcal{P}(\tau)$.
6) $e \in E$ is a predicate on type τ, if e is an expression of type τ and E is an expression of type $\mathcal{P}(\tau)$.
7) $e_1 = e_2$ and $e_1 \neq e_2$ are predicates on type τ, if e_1 and e_2 are expressions of type τ.
8) $R(e_1, \cdots e_n)$ is a predicate on type τ, if $e_1, \cdots e_n$ are expressions of type τ, and R is any n-ary relation

symbol on type τ.

9) $e_1 \subset e_2$ and $e_1 \subseteq e_2$ are predicates on type $\mathcal{P}(\tau)$, if e_1 and e_2 are expressions of type $\mathcal{P}(\tau)$.
10) $p \wedge q$, $p \vee q$, $p \equiv q$, $p \Rightarrow q$ and $\neg p$ are predicates, if p and q are predicates.
11) $\forall x \in D \cdot p$ and $\exists x \in D \cdot p$ are predicates, if D is an expression of type $\mathcal{P}(\tau)$ or a non-terminal symbol s, x is a variable of type τ or of type s, and p is a predicate.

Further functions and relations can be defined as usual in the FOL. For the sake of readability, we will also use infix and prefix forms for defined functions and relations. Thus, we may also write the application of function f to argument x with the more conventional prefix notation $f(x)$.

Example 3 (Definition of a function): For example, the set of nodes in a graph g that have no weight associated with them can be formally defined using the functions induced from the syntax definition as follows.

$$UnweightedNodes(g : Graph) \triangleq \{n | n \in g.nodes \wedge n.weight = \bot\} \qquad \square$$

D. Meta-Modeling

Given the abstract syntax of a modeling language defined in GEBNF, meta-modeling in the framework of the modeling language can be performed by defining a predicate p such that the required subset of models are those satisfy the predicate. In the sequel, we define a *meta-model* to be an ordered pair $(\mathbf{G}, p)$, where $\mathbf{G}$ is a GEBNF syntax and p is a predicate in the FOL induced from $\mathbf{G}$.

Example 4 (Meta-modeling): Consider the directed graphs defined in Example 1. The set of strongly connected graphs can be defined as the set of models that satisfy the following predicate.

$$\forall x, y \in g.nodes \cdot (x \ reaches \ y),$$

where g is a variable of type $Graph$, and the predicate $(x \ reaches \ y)$ is defined as follows.

$$(x \ reaches \ y) \Leftrightarrow \exists e \in g.edges \cdot (x = e.from \wedge y = e.to) \vee \exists z \in g.nodes \cdot ((x \ reaches \ z) \wedge (z \ reaches \ y))$$

The set of acyclic graphs can be defined as the set of models that satisfy the following predicate.

$$\forall x, y \in g.nodes \cdot ((x \ reaches \ y) \Rightarrow x \neq y)$$

The set of connected graphs can be defined as follows.

$$\forall x \neq y \in g.nodes \cdot ((x \ reaches \ y) \vee (y \ reaches \ x)).$$

Finally, a tree can be defined as satisfying the following condition.

$$\exists x \in g.nodes \cdot (\forall y \in g.nodes \cdot (x \ reaches \ y)) \wedge \forall e, e' \in g.edges \cdot (e.to = e'.to \Rightarrow e = e') \qquad \square$$

In the same way, design patterns have been specified by first defining the abstract syntax of UML class diagrams and sequence diagrams in GEBNF, and then specifying the conditions that their instances must satisfy [7], [9].

III. Axiomatization of Syntax Constraints

In this section, we discuss how to use the induced FOL to characterize the syntax restrictions that GEBNF imposes on models.

A. Optional Elements

Assume that a non-terminal symbol A is defined in the following form.

$$A ::= \cdots, f : [B], \cdots.$$

Then, an occurrence of an element of type B in an element of type A is optional. The function f has the type $A \to B$, which is the same as the function g in the following syntax rule, where B is not optional.

$$A ::= \cdots, g : B, \cdots.$$

The difference is that f is a partial function while g is a total function. Therefore, for each non-optional function symbol g, we require it satisfying the following condition.

$$\forall x \in A \cdot (x.g \neq \bot),$$

where $\bot$ means undefined.

Example 5 (Partial functions): In Example 1, according to the second syntax rule, a node n may be associated with no weight. Thus, the function $weight$ of type $Node \to Real$ is a *partial* function. When a node n has no weight, $n.weight$ is undefined and we write $n.weight = \bot$. The type of a function does not distinguish total functions from partial functions. Instead, we assume that all function symbols are partial unless explicitly stated by an axiom about the function. An example of total function is $name : Node \to String$. It, therefore, must satisfy the following condition.

$$\forall x \in Node \cdot (x.name \neq \bot). \qquad \square$$

B. Non-Empty Repetitions

Assume that a non-terminal symbol A is defined in one of the following forms.

$$A ::= \cdots, f : B^*, \cdots \quad (1)$$
$$A ::= \cdots, g : B^+, \cdots \quad (2)$$

An element of type A may contain a set (in case of (1)) or non-empty set (in case of (2)) of elements of type B. The functions f and g induced from the above syntax rules are of the same type, i.e. $A \to \mathcal{P}(B)$. But, the image of the former can be an empty set while that of the latter cannot. Thus,

for each of the non-empty repetition structure, we require the function g satisfying the following condition.

$$\forall x \in A \cdot (x.g \neq \emptyset).$$

Example 6 (Non-empty Repetition): In Example 1, the set of nodes in a directed graph is defined as a non-empty repetition while the set of edges is defined as repetition that allows the empty occurrence. Therefore, the function $nodes$ must satisfy the following axiom, but $edges$ do not.

$$\forall g \in Graph \cdot (g.nodes \neq \emptyset). \qquad \square$$

C. Referential and Creative Elements

Assume that a non-terminal symbol A is defined in the following form.

$$A ::= \cdots, f : \underline{B}@C.g, \cdots$$

The field f of an element of type A will contain a reference to an element of type B in the field g of an element of type C. Thus, it is called a *referential occurrence*. The function f has the same type $A \rightarrow B$ as the function f' in the following syntax rule, where the element of type B is a *creative occurrence*.

$$A ::= \cdots, f' : B, \cdots$$

However, the function f has different properties from f'. Thus, its semantics in terms of the structure of the models is different. For example, if the syntax definition of $Edge$ in Example 1 is replaced by the following rule (i.e. when the reference modifier on $Node$ is removed from the original rule),

$$Edge ::= from : Node, to : Node, weight : Real,$$

each edge will introduce two new nodes, i.e. for all edges $e \neq e' \in Edges$, we have that $e.from \neq e'.from$. Moreover, for all edges e, we have that the node $e.from$ must be different from the node $e.to$, i.e. $e.from \neq e.to$. In contrast, the original definition requires that for all $e \in g.edges$, we have $e.from \in g.nodes$ and $e.to \in g.nodes$. This allows $e.from = e.to$, $e.from = e'.from$ and $e.to = e'.to$ to be true for some edges e and e'.

In general, the function symbols induced from creative occurrences of a same non-terminal symbol must have disjoint images. Formally, let f and g be two functions induced from two creative occurrences of non-terminal symbol X in two syntax rules in the following form,

$$Y ::= \cdots, f : E(X), \cdots$$
$$Z ::= \cdots, g : E'(X), \cdots$$

where $E(X)$ and $E'(X)$ be any of the expressions X, $[X]$ and $(X|X_1|\cdots|X_n)$, we require functions f and g satisfy the following condition.

$$\forall a \in Y \cdot \forall b \in Z \cdot (a.f \neq b.g).$$

Let $E(X)$ and $E'(X)$ be any of the expressions X^* and X^+. If the syntax rules are in the form of

$$Y ::= \cdots, f : E(X), \cdots$$
$$Z ::= \cdots, g : E'(X), \cdots,$$

we require that the functions f and g satisfy the following conditions.

$$\forall a \in Y \cdot \forall b \in Z \cdot (\forall x \in a.f \cdot (x \notin b.g)),$$
$$\forall a \in Y \cdot \forall b \in Z \cdot (\forall x \in b.g \cdot (x \notin a.f),$$

or simply,

$$\forall a \in Y \cdot \forall b \in Z \cdot (b.g \cap a.f = \emptyset).$$

Similarly, let $E(X)$ be any of the expressions X, $[X]$ and $(X|X_1|\cdots|X_n)$, and $E'(X)$ be any of the expressions X^* and X^+. If the syntax rules are in the form of

$$Y ::= \cdots, f : E(X), \cdots$$
$$Z ::= \cdots, g : E'(X), \cdots,$$

we require that the functions f and g satisfy the following property.

$$\forall a \in Y \cdot \forall b \in Z \cdot (a.f \notin b.g).$$

The semantics of referential occurrences can also be formally defined as constraints on models as follows.

Let $E(X)$ be in one of the forms X, $[X]$ and $(X|X_1|\cdots|X_n)$. For syntax rules in the following form:

$$Y ::= \cdots, g : E(X), \cdots,$$
$$Z ::= \cdots, f : \underline{X}@Y.g, \cdots,$$

we require functions f and g satisfy the following property.

$$\forall a \in Z \cdot \forall b \in Y \cdot (a.f = b.g)$$

Let $E(X)$ be one of the forms X^* and X^+. If the syntax rules are in the following form:

$$Y ::= \cdots, g : E(X), \cdots,$$
$$Z ::= \cdots, f : \underline{X}@Y.g, \cdots,$$

we require functions f and g satisfy the following property.

$$\forall a \in Z \cdot \forall b \in Y \cdot (a.f \in b.g)$$

Let $E(X)$ and $E'(X)$ be any of the forms X^* and X^+. If the syntax rules are in the following form:

$$Y ::= \cdots, g : E(X), \cdots,$$
$$Z ::= \cdots, f : \underline{E'(X)}@Y.g, \cdots,$$

we require functions f and g satisfy the following property.

$$\forall a \in Z \cdot \forall b \in Y \cdot (a.f \subseteq b.g)$$

It is worth noting that the above constraints are in the FOL induced from syntax definitions.

Example 7 (Referential Occurrences): In Example 1, there are two referential occurrences of non-terminal symbols. Thus, the functions to and $from$ must satisfy the following conditions.

$$\forall g \in Graph \cdot \forall e \in g.edges \cdot (e.from \in g.nodes)$$
$$\forall g \in Graph \cdot \forall e \in g.edges \cdot (e.to \in g.nodes) \quad \square$$

Let $\mathbf{G}$ be any well-formed syntax definition in GEBNF. In the sequel, we write $Axiom(\mathbf{G})$ to denote the set of constraints derived from $\mathbf{G}$ according to the above rules.

IV. Algebraic Semantics

This section formally defines the semantics of GEBNF.

A. Models as Mathematical Structures

Let $\mathbf{G} = \langle R, N, T, S\rangle$ be a GEBNF syntax definition and $\Sigma_G = (N \cup T, F_G)$, where $F_G = Fun(\mathbf{G})$ is the set of function symbols induced from $\mathbf{G}$. Σ_G is called the *signature* induced from $\mathbf{G}$.

Definition 4 (Σ_G-Algebras): A Σ_G-algebra $\mathcal{A}$ is a mathematical structure that consists of a family $\{A_x | x \in N \cup T\}$ of sets and a set of functions $\{f_\varphi | \varphi \in F_G\}$, where if φ is of type $X \to Y$, then f_φ is a function from set A_X to the set $[\![Y]\!]$, where

$$[\![Y]\!] = \begin{cases} A_Y, & if \quad Y \in N \cup T \\ \mathcal{P}(A_Z), & if \quad Y = \mathcal{P}(Z) \\ \sum_{i=1}^{n} A_{C_i}, & if \quad Y = \sum_{i=1}^{n} C_i. \end{cases}$$

$\square$

In particular, for each terminal symbol $s \in T$, for example, *String*, and the set Op_s of operator symbols and set R_s of relational symbols defined on s, there is a mathematical structure $\langle A_s, \{op_\varphi | \varphi \in Op_s\} \cup \{r_\rho | \rho \in R_s\}\rangle$ such that

1) there is a non-empty set A_s of elements, which are elements of type s;
2) for each operator symbol φ in the set Op_s, there is a corresponding operation op_φ defined on A_s;
3) for each n-ary relational symbol ρ, there is a corresponding n-ary relation r_ρ defined on A_s.

Obviously, not all Σ_G-algebras are syntactically valid models. Thus, we have the following notion of 'no junk'.

Definition 5 (Algebra without Junk): We say that a Σ_G-algebra $\mathcal{A}$ contains *no junk*, if

1) $|A_R| = 1$, and
2) for all $s \in N$ and all $e \in A_s$, we can define a function $f : R \to \mathcal{P}(s)$ in FOL such that for some $m \in A_R$ we have $e \in f(m)$. $\square$

Informally, we consider a Σ_G-algebra $\mathcal{A}$ as a model in the modeling language. No junk means there is only one root element. This is similar to the condition that a parse tree of a program must have one and only one root. Moreover, because each non-terminal symbol corresponds to a type of model elements, every element in a model must be accessible from the root. This is similar to the condition that every element in a program must be on the parsi tree of the program and thus is accessible from the root of the tree.

In the sequel, we will only consider Σ_G-algebras that contain no junk.

B. Satisfaction of Constraints

For a Σ_G-algebra to be a syntactically valid model, it must also satisfy the axioms derived from the GEBNF syntax. The following defines what is meant by an algebra satisfies a predicate.

An assignment in an Σ-algebra $\mathcal{A}$ is a mapping α from the set V of variables to the elements of the algebra.

Definition 6 (Evaluation of Expressions and Predicates): The evaluation of an expression e or predicate p under an assignment α, written $[\![e]\!]_\alpha$, is defined as follows.

- $[\![c]\!] = c$, if c is a constant of basic type $\tau \in T$;
- $[\![v]\!]_\alpha = \alpha(v) \in A_\tau$, if v is a variable of type τ;
- $[\![e.f]\!]_\alpha = f_A([\![e]\!]_\alpha)$;
- $[\![\{e(x) | Pred(x)\}]\!]_\alpha = \{[\![e(x)]\!]_\alpha | [\![Pred(x)]\!]_\alpha\}$;
- $[\![e_1 \cup e_2]\!]_\alpha = [\![e_1]\!]_\alpha \cup [\![e_2]\!]_\alpha$;
- $[\![e_1 \cap e_2]\!]_\alpha = [\![e_1]\!]_\alpha \cap [\![e_2]\!]_\alpha$;
- $[\![e_1 - e_2]\!]_\alpha = [\![e_1]\!]_\alpha - [\![e_2]\!]_\alpha$;
- $[\![e \in E]\!]_\alpha = [\![e]\!]_\alpha \in [\![E]\!]_\alpha$;
- $[\![e_1 = e_2]\!]_\alpha = ([\![e_1]\!]_\alpha = [\![e_2]\!]_\alpha)$
- $[\![e_1 \neq e_2]\!]_\alpha = ([\![e_1]\!]_\alpha \neq [\![e_2]\!]_\alpha)$;
- $[\![R(e_1, \cdots e_n)]\!]_\alpha = R_A([\![e_1]\!]_\alpha, \cdots, [\![e_n]\!]_\alpha)$;
- $[\![e_1 \subset e_2]\!]_\alpha = [\![e_1]\!]_\alpha \subset [\![e_2]\!]_\alpha$;
- $[\![e_1 \subseteq e_2]\!]_\alpha = [\![e_1]\!]_\alpha \subseteq [\![e_2]\!]_\alpha$;
- $[\![p \wedge q]\!]_\alpha = [\![p]\!]_\alpha \wedge [\![q]\!]_\alpha$;
- $[\![p \vee q]\!]_\alpha = [\![p]\!]_\alpha \vee [\![q]\!]_\alpha$;
- $[\![p \equiv q]\!]_\alpha = ([\![p]\!]_\alpha \equiv [\![q]\!]_\alpha)$;
- $[\![p \Rightarrow q]\!]_\alpha = ([\![p]\!]_\alpha \Rightarrow [\![q]\!]_\alpha)$;
- $[\![\neg p]\!]_\alpha = \neg [\![p]\!]_\alpha$;
- $[\![\forall x \in D \cdot (p)]\!]_\alpha = True$, if for all e in $[\![D]\!]_\alpha$, $[\![p]\!]_{\alpha[x/e]}$ is true;
- $[\![\exists x \in D \cdot (p)]\!]_\alpha = True$, if there exists e in $[\![D]\!]_\alpha$ such that $[\![p]\!]_{\alpha[x/e]}$ is true.

where $[\![D]\!]_\alpha = A_\tau$, if $D \in N \cup T$, and $\alpha[x/e](x) = e$. $\square$

Let α be an assignment in Σ_G-algebra $\mathcal{A}$ and p be a predicate in the FOL induced from $\mathbf{G}$.

Definition 7 (Satisfaction Relation): We say that p is true in $\mathcal{A}$ under assignment α and write $\mathcal{A} \models_\alpha p$, if $[\![p]\!]_\alpha = true$. We say that p is true in $\mathcal{A}$ and write $\mathcal{A} \models p$, if for all assignments α in $\mathcal{A}$ we have that $\mathcal{A} \models_\alpha p$. $\square$

We can now define what is a syntactically valid models and the semantics of meta-models.

Definition 8 (Syntactically Valid Models): A Σ_G-algebra $\mathcal{A}$ (with no junk) is a syntactically valid model with respect to $\mathbf{G}$, if for all $p \in Axiom(\mathbf{G})$, we have that $\mathcal{A} \models p$.

Let $MM = (\mathbf{G}, p)$ be a meta-model that consists of a GEBNF syntax definition $\mathbf{G}$ and a predicate p in the FOL induced from $\mathbf{G}$. The semantics of the meta-model MM is

a subset of syntactically valid models of $\mathbf{G}$ that satisfy the predicate p. □

Note that, the definition of satisfaction relation is the standard treatment of first order logic in the model theory of mathematical logics. Thus, it is sound, complete and compact [11].

V. Institution of Meta-models

Meta-modeling often involves multiple meta-models, where each meta-model defines a FOL. Translation between such logics plays a fundamental role in model transformation and reasoning about models. The syntax and semantics of such translations are captured by the theory of institutions [12] and entailment systems. This section we apply these theories to GEBNF.

A. The Category of GEBNF Syntax Definitions

Let's first introduce a few mathematical notions and notations.

A *category* $\mathbb{C}$ consists of a class C_{obj} of *objects* and a class C_m of *morphisms* or *arrows* between objects together with the following three operations:

- $dom : C_m \to C_{obj}$;
- $codom : C_m \to C_{obj}$;
- $id : C_{obj} \to C_m$,

where for all morphisms f, $dom(f) = A$ is called the *domain* of the morphism f; $codom(f) = B$ the *codomain*, and we say that the morphism f is from object $A = dom(f)$ to object $B = codom(f)$, written $f : A \to B$. For each object A, $id(A)$ is the *identity morphism* that its domain and codomain are A. $id(A)$ is also written as id_A.

Moreover, there is a partial operation $\circ$ of *composition* of morphisms. The composition of morphisms f and g, written $f \circ g$, is defined if $dom(f) = codom(g)$. The result of composition $f \circ g$ is a morphism from $dom(g)$ to $codom(f)$. The composition operation has the following properties. For all morphisms f, g, h,

$$
\begin{array}{ll}
(f \circ g) \circ h = f \circ (g \circ h) & \\
id_A \circ f = f, & if \quad codom(f) = A \\
g \circ id_A = g, & if \quad dom(g) = A.
\end{array}
$$

Given a category $\mathbb{C}$, we will also write $|\mathbb{C}|$ and $||\mathbb{C}||$ to denote C_{obj} and C_m, respectively, in the sequel.

We now define the morphisms between GEBNF syntax definitions and prove that they form a category.

Let $\mathbf{G} = \langle R_G, N_G, T_G, S_G \rangle$, $\mathbf{H} = \langle R_H, N_H, T_H, S_H \rangle$ be two GEBNF syntax definitions, $Fun(G)$ and $Fun(H)$ be the function symbols induced from $\mathbf{G}$ and $\mathbf{H}$, respectively.

Definition 9 (Syntax Morphisms): A syntax morphism μ from $\mathbf{G}$ to $\mathbf{H}$, written $\mu : \mathbf{G} \to \mathbf{H}$, is a pair (m, f) of mappings $m : N_G \to N_H$ and $f : Fun(G) \to Fun(H)$ that satisfy the following two conditions:

1) *Root preservation*: $m(R_G) = R_H$;
2) *Type preservation*: for all $op \in Fun(G)$, $(op : A \to B) \Rightarrow (f(op) : m(A) \to m(B))$, where we naturally extend the mapping m to type expressions. □

The composition of two syntax morphisms is the composition of the mappings correspondingly. Formally, we have the following definition.

Definition 10 (Composition of Syntax Morphisms): Assume that $\mu = (m, f) : \mathbf{G} \to \mathbf{H}$ and $\nu = (n, g) : \mathbf{H} \to \mathbf{J}$ be syntax morphisms. The composition of μ to ν, written $\mu \circ \nu$, is defined as $(m \circ n, f \circ g)$. □

We can prove that the above definition is sound.

Lemma 1 (Soundness of Syntax Morphism Compositions): For all syntax morphisms $\mu : \mathbf{G} \to \mathbf{H}$, $\nu : \mathbf{H} \to \mathbf{J}$, and $\omega : \mathbf{J} \to \mathbf{K}$, we have that:

1) $\mu \circ \nu$ is a syntax morphism from $\mathbf{G}$ to $\mathbf{J}$;
2) $(\mu \circ \nu) \circ \omega = \mu \circ (\nu \circ \omega)$.

Proof.

1) The statement can be proved by showing that the composition satisfies the root and type preservation conditions. Details are omitted for the sake of space.
2) The statement follows the associative property of the composition of mappings. □

We now define the identity syntax morphism Id_G on $\mathbf{G}$. Let id_X be the identity mapping on set X.

Definition 11 (Identity Syntax Morphisms): For all $\mathbf{G} = \langle R, N, T, S \rangle$, Id_G is defined as the pair of mappings $(id_N, id_{Fun(G)})$. □

The following lemma proves that the definition of Id_G is sound, i.e., they are indeed syntax morphisms and have the identity property. Its proof is omitted for the sake of space.

Lemma 2 (Soundness of Identity Syntax Morphisms): For all GEBNF syntax definitions $\mathbf{G}$ and $\mathbf{H}$, we have that

1) Id_G is a syntax morphism.
2) For all syntax morphism $\mu : \mathbf{G} \to \mathbf{H}$, we have that $Id_G \circ \mu = \mu$ and $\mu \circ Id_H = \mu$. □

From Lemma 1 and 2, we can easily prove that the set of GEBNF syntax definitions and the syntax morphisms defined above form a category.

Theorem 1 (Category of GEBNF Syntax): Let Obj be the set of well-formed GEBNF syntax definitions, Mor be the set of syntax morphisms on Obj. (Obj, Mor) is a category. It is denoted by $\mathbb{GEB}$ in the sequel.

Proof. The theorem directly follows Lemma 1 and 2. □

B. Translation of Sentences through Syntax Morphisms

Given a syntax morphism from one GEBNF defined modeling language to another, we can define a translation between the FOLs induced from them. Such a translation can be formalized as a functor between categories. The notion of functor is defined as follows.

Let $\mathbb{C}, \mathbb{D}$ be two categories. A *functor* F from $\mathbb{C}$ to $\mathbb{D}$ consists of two mappings: an object mapping $F_{obj} : C_{obj} \to$

D_{obj}, and a morphism mapping $F_m : C_m \to D_m$ that have the following properties.

First, for all morphisms $f : A \to B$ of category $\mathbb{C}$, we have that $F_m(f) : F_{obj}(A) \to F_{obj}(B)$ in category $\mathbb{D}$.

Second, for all morphisms f and g in $\mathbb{C}$, we have that

$$F_m(f \circ g) = F_m(f) \circ F_m(g).$$

Finally, for all objects A in category $\mathbb{C}$, we have that $F_m(id_A) = id_{F_{obj}(A)}$.

The following defines a functor from the category of GEBNF syntax definitions to the category of the sets of sentences in the FOL induced from GEBNF syntax definitions.

Definition 12 (Functor Sen from Syntax to Sentences): Given a well-formed GEBNF syntax definition $\mathbf{G}$, $Sen_{obj}(G)$ is the set of predicates on the root of $\mathbf{G}$.

Given a syntax morphism $\mu = (m, f)$ from $\mathbf{G}$ to $\mathbf{H}$, we define a mapping $Sen_m(\mu)$ from $Sen_{obj}(G)$ to $Sen_{obj}(H)$ as follows. For each predicate p in $Sen_{obj}(G)$,

1) Each variable v of type τ in predicate p is replaced by a variable v' of type $m(\tau)$.
2) Each $op \in Fun(G)$ in predicate p is replaced by the function symbol $f(op)$.

The predicate p' obtained is the image of p under $Sen_m(\mu)$. □

The following Theorem proves that the pair of mappings is a functor indeed.

Theorem 2 (Soundness of the Definition of Functor Sen): The pair (Sen_{obj}, Sen_m) is a functor from category $\mathbb{GEB}$ of GEBNF syntax definitions to the category $\mathbb{SET}$ of sentences in the corresponding induced FOLs.

Proof.

For the sake of space, here we only give a skeleton of the proof. Details are omitted.

First, we prove that for all predicate p in $Sen_{obj}(G)$, $Sen_m(\mu)(p)$ is a predicate in $Sen_{obj}(H)$. Thus, $Sen_m(\mu)$ is a mapping from $Sen_{obj}(G)$ to $Sen_{obj}(H)$. This can be proved by induction on the structure of the predicate p.

Second, we prove that $Sen_m(\mu \circ \nu) = Sen_m(\mu) \circ Sen_m(\nu)$. This follows directly the definition of syntax morphisms.

Finally, we prove that for all GEBNF syntax definition $\mathbf{G}$, $Sen_m(Id_G)$ is also the identity mapping on $Sen_{obj}(G)$. This directly follows the definition of Id_G. □

C. Translation of Models

The translation of the models in one modeling language to another can also be defined as a functor.

We first observe that the models in any given modeling language defined by a GEBNF syntax definition is a category, where the morphisms are the homomorphisms between the models (i.e. the algebras).

Let $\mathbf{G}$ be any given GEBNF syntax definition. We denote the set of Σ_G-algebras without junk by $Mod(G)$. The following defines the homomorphisms between models.

Definition 13 (Homomorphisms): Let $\mathcal{A}$ and $\mathcal{B}$ be Σ_G-algebras, a homomorphism φ from $\mathcal{A}$ to $\mathcal{B}$ is a mapping $\varphi : A \to B$ such that

$$\forall s \in N \cup T \cdot \forall x \in A_s \cdot (\varphi(x) \in B_s)),$$

and,

$$\forall x \in A_\tau \cdot \forall f \in F_G \cdot (f_B(\varphi(x)) = \varphi(f_A(x))),$$

where functions $f(x)$ are naturally extended to functions on sets such that $f(X) = \{f(x) | x \in X\}$. □

Lemma 3 (Category of Models): For any given well-formed GEBNF syntax definition $\mathbf{G}$, the set of Σ_G-algebras without junk as the set of objects and homomorphisms between them as the set of morphisms form a category, where for each Σ_G-algebra without junk $\mathcal{A}$, Id_A is the identity mapping on A. The category is denoted by $\mathbb{MOD}_G$ in the sequel.

Proof. The statement can be proved by showing the conditions of a category are satisfied. In particular, the associativity of morphism composition follows the associativity of the composition of homomorphisms. The unit property of Id_A follows the unit property of homomorphisms. □

Now, we define a category whose objects are the categories $\mathbb{MOD}_G$ for $\mathbf{G}$ varies in the set of GEBNF syntax definitions, and the morphisms are functors U_μ between these categories of models, where μ is any syntax morphism between GEBNF syntax definitions.

For each syntax morphism $\mu = (m, f)$ from $\mathbf{G}$ to $\mathbf{H}$, the mapping U_μ from category $\mathbb{MOD}_H$ to category $\mathbb{MOD}_G$ is defined as follows.

Let $\mathcal{B} \in |\mathbb{MOD}_H|$. We define an Σ_G-algebra $\mathcal{A}$ as follows:

1) For each $s \in N_G$, $A_s = B_{m(s)}$;
2) For each function symbol $op \in Fun(G)$, the function $\varphi_{op} \in \mathcal{A}$ is the function $\varphi_{f(op)}$ in $\mathcal{B}$.

We can prove that $\mathcal{A}$ defined as such is a Σ_G-algebra and contains no junk, thus it is in $|\mathbb{MOD}_G|$. Moreover, through U_μ, the homomorphisms between models in $||\mathbb{MOD}_H||$ are also naturally induced into the homomorphisms between such defined models in $\mathbb{MOD}_G$. Therefore, we have the following lemma.

Lemma 4 (Functor between Categories of Models): For each syntax morphism $\mu = (m, f)$ from $\mathbf{G}$ to $\mathbf{H}$, the mapping U_μ from objects of category $\mathbb{MOD}_H$ to the objects of category $\mathbb{MOD}_G$ and its naturally induced mapping on homomorphisms is a functor from $\mathbb{MOD}_H$ to $\mathbb{MOD}_G$. □

Furthermore, we have the following theorem.

Theorem 3 (Category of Modeling Languages): Let $Obj = \{\mathbb{MOD}_G | G \in |\mathbb{GEB}|\}$ and $Mor = \{U_\mu | \mu \in ||\mathbb{GEB}||\}$. (Obj, Mor) is a category. In the sequel, it is denoted by $\mathbb{CAT}$.

Proof. It is easy to prove that the definition satisfies the conditions of a category. Details are omitted for the sake of space. □

Now, we define the model translation as a functor.

Definition 14 (Model Translation): We define mappings $MOD_{obj} : |\mathbb{GEB}| \to |\mathbb{CAT}^{op}|$ and $MOD_m : ||\mathbb{GEB}|| \to ||\mathbb{CAT}^{op}||$ as follows.

$$MOD_{obj}(G) = Mod(G);$$
$$MOD_m(u) = U_{\mu}^{op}$$

where for an arrow $\mu : a \to b$, μ^{op} is the inverse arrow of μ. □

Then, we have the following theorem. Here, again for the sake of space, we omit the proof.

Theorem 4 (Functor of Model Translation): MOD is a functor from $\mathbb{GEB}$ to $\mathbb{CAT}^{op}$. □

D. Institution of GEBNF induced First Order Logics

We are now ready to prove that GEBNF and its induced first order logics form an institution. First let's review the notion of institution [12].

An institution is a tuple $(Sig, Mod, Sen, \models)$, where

1) Sig is a category whose objects are called signatures.
2) $Sen : Sig \to Set$ is a functor that for each signature it gives a set of sentences over that signature.
3) $Mod : Sig \to Cat^{op}$ is a functor that for each signature Σ it gives a category $Mod(\Sigma)$ whose objects are called Σ-models and whose arrows are called Σ-homomorphisms.
4) $\models$ is a signature indexed family of relations $(\models_\Sigma)_{\Sigma \in |Sig|}$, where for each $\Sigma \in |Sig|$, $\models_\Sigma \subseteq |Mod(\Sigma)| \times Sen(\Sigma)$. It is called Σ-satisfaction. It must satisfy the condition that for any $(\phi : \Sigma \to \Sigma') \in ||Sig||$, any $M' \in |Mod(\Sigma')|$ and any $e \in Sen(\Sigma)$,

$$M' \models_{\Sigma'} Sen(\phi)(e) \Leftrightarrow Mod(\phi)(M') \models_\Sigma e.$$

Theorem 5 (GEBNF Institution): Let $\mathbb{GEB}$ be the category of well-formed GEBNF syntax definitions as proved in Theorem 1; MOD be defined in Definition 14; Sen be defined in Definition 12; and $\models$ be the satisfaction relation defined in Definition 7. The tuple $(\mathbb{GEB}, MOD, Sen, \models)$ is an institution.

Proof.

The condition 1) of institution is true by Theorem 1. Condition 2) is true by Theorem 2. Condition 3) is true by Theorem 4. Condition 4) can be proved by induction on the structure of the sentence e. It is tedious but straightforward. Details are thus omitted for the sake of space. □

Note that, condition (4) means that the truth of a sentence is invariant under the translation of sentence and the models.

VI. Conclusion

In the past few years, many research efforts on meta-modeling have been reported in the literature. Existing meta-modeling languages can be classified into the general purpose and special purpose ones.

MOF and its representation in UML class diagrams is a general purpose meta-modeling language. It is a part of four layer UML language definition architecture. In such a meta-model, the basic concepts of a modeling language is represented as the meta-classes. The relationships between the concepts are represented as meta-relations between the meta-classes. Restrictions on the syntax and usage of models are specified using multiplicities and other properties (such as derived property, default values, etc.) associated to meta-classes and meta-relations. There are two long lasting issues associated to the UML meta-modeling approach. First, the semantics of meta-models is informally defined. There are few research efforts to formalize the semantics of UML meta-model, except [17]. Second, graphic notions are weak in expressiveness. In particular, properties that involve multiple entities are hard to express in graphic notation. This is partially overcome by defining and employing the Object Constraint Language OCL associated to elements in the meta-models. OCL is in fact also a first order predicate logic language induced from meta-model, but it is represented in a syntax more close to object oriented programming languages. Attempts to formalize the semantics of OCL have been reported in [18], [19], [20], [21], [22], [23], [24], [25], [26] etc. However, it is still unsatisfactory in the formal definition of OCL's semantics and understanding of its logic properties [27], [28].

Many special purpose meta-modeling languages have been proposed, mostly for defining design patterns. Typical examples are LePUS [13], [29], RBML [2], DPML [15], [16], and PDL [30]. They all use graphic notation to represent meta-models. In general, graphic meta-modeling approach suffers from several drawbacks. First, graphic meta-models are difficult to understand. This is partly solved in RBML, DPML and PDL by introducing new graphic notations for meta-models, but at the price of complexity in their semantics, which have not been formally defined. Second, graphic meta-models are ambiguous as in all graphic modeling languages such as UML. LePUS is the only exception that it has a formal specification of its semantics in first order logic. Third, graphical meta-models are not expressive enough. In particular, they are unable to state what is NOT allowed to be in a model.

In this paper, we have advanced the GEBNF approach to meta-modeling by laying its theoretical foundation on the basis of mathematical logic and theory of institutions. The main contributions are:

- We have formally defined semantics of a GEBNF syntax definitions as the algebras without junk and

satisfying a set of constraints written in the induced FOL. These constraints are derived from the syntax rules in GEBNF. We proved that these algebras and homomorphisms between them form a category.

- We have formally proved that GEBNF syntax definitions and syntax morphisms form a category, where a syntax morphism represents translations between modeling languages. Thus, this lays a solid foundation for model transformations and extension mechanisms of meta-modeling.
- We have formally proved that the category of GEBNF syntax definitions, the categories of models in any given modeling languages defined by GEBNF and the satisfaction relation form an institution. Therefore, GEBNF syntax definitions and the induced FOLs form a valid specification language of models.

For future work, we are considering developing software tools to support meta-modeling in GEBNF. Further application of theory to facilitate a meta-model extension mechanism should also be interesting. It is also interesting to found out if the approach taken by this paper is applicable to meta-models in UML class diagrams and OCL.

REFERENCES

[1] OMG, "Unified modeling language: Superstructure, version 2.0, formal/05-07-04," 2004.

[2] R. B. France, D.-K. Kim, S. Ghosh, and E. Song, "A UML-based pattern specification technique," *IEEE Trans. Softw. Eng.*, vol. 30, no. 3, pp. 193–206, 2004.

[3] M. Elaasar, L. C. Briand, and Y. Labiche, "A metamodeling approach to pattern specification," in *Proc. of MoDELS'06*, LNCS 4199. Springer, Oct. 2006, pp. 484–498.

[4] C. Mraidha, S. Gerard, F. Terrier, and J. Benzakki, "A two-aspect approach for a clearer behavior model," in *Proc. of ISORC'03*, May 2003, pp. 213–220.

[5] H. Zhu and L. Shan, "Well-formedness, consistency and completeness of graphic models," in *Proc. of UKSIM'06*, Apr. 2006, pp. 47–53.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] I. Bayley and H. Zhu, "Specifying behavioural features of design patterns in first order logic," in *Proc. of COMPSAC'08*, 2008, pp. 203–210.

[8] ——, "On the composition of design patterns," in *Proc. of QSIC'08*, Aug. 2008, pp. 27–36.

[9] ——, "Formal specification of the variants and behavioural features of design patterns," *Journal of Systems and Software*, vol. 83, no. 2, pp. 209–221, Feb. 2010.

[10] H. Zhu, I. Bayley, L. Shan, and R. Amphlett, "Tool support for design pattern recognition at model level," in *Proc. of COMPSAC'09*, July 2009, pp. 228–233.

[11] I. Chiswell and W. Hodges, *Mathematical Logic*. Oxford University Press, 2007.

[12] J. A. Goguen and R. M. Burstall, "Institutions: Abstract model theory for specification and programming," *J. ACM*, vol. 39, no. 1, pp. 95–146, 1992.

[13] A. H. Eden, "Formal specification of object-oriented design," in *Int'l Conf. on Multidisciplinary Design in Engineering*, November 2001.

[14] ——, "A theory of object-oriented design," *Information Systems Frontiers*, vol. 4, no. 4, pp. 379–391, 2002.

[15] D. Maplesden, J. Hosking, and J. Grundy, "A visual language for design pattern modelling and instantiation," in *Proc. of HCC'01*, 2001, pp. 338–339.

[16] ——, "Design pattern modelling and instantiation using DPML," in *Proc. of TOOLS Pacific'02*. 2002, pp. 3–11.

[17] L. Shan and H. Zhu, "Semantics of metamodels in UML," in *Proc. of TASE'09*. July 2009, pp. 55–62.

[18] M. V. Cengarle and A. Knapp, "A formal semantics for OCL 1.4," in *Proc. of UML'01*, LNCS 2185. Springer, Oct. 2001, pp. 118–133.

[19] M. Richters and M. Gogolla, "OCL: Syntax, semantics, and tools," in *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, T. Clarke and J. Warmer, Eds. LNCS 2263, Springer. 2002, pp. 42–68.

[20] R. Hennicker, H. Hussmann, and M. Bidoit, "On the precise meaning of OCl constraints," in *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, T. Clarke and J. Warmer, Eds. LNCS 2263, Springer. 2002, pp. 69–84.

[21] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills, "The amsterdam manifesto on OCL," in *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, T. Clarke and J. Warmer, Eds. LNCS 2263, Springer. 2002, pp. 115–149.

[22] A. Kleppe and J. Warmer, "The semantics of the OCL action clause," in *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, T. Clarke and J. Warmer, Eds. LNCS 2263, Springer. 2002, pp. 213–227.

[23] A. D. Brucker and B. Wolff, "A proposal for a formal ocl semantics in isabelle/hol," in *Proc. of TPHOLs'02*, V. A. Carreno, C. A. Munoz, and S. Tahar, Eds., LNCS 2410, Springer. Aug. 2002, pp. 147–175.

[24] S. Flake, "Towards the completion of the formal semantics of ocl 2.0," in *Proc. of ACSC '04*. 2004, pp. 73–82.

[25] R. Hennicker, A. Knapp, and H. Baumeister, "Semantics of ocl operation specifications," *Electronic Notes in Theoretical Computer Science*, vol. 102, pp. 111–132, Nov. 2004.

[26] A. Boronat and J. Meseguer, "Algebraic semantics of OCL-constrained metamodel specifications," in *Objects, Components, Models and Patterns– Proc. of TOOLS EUROPE'09*, M.Oriol and B.Meyer, Eds., Springer, 2009, pp. 96–115.

[27] ——, "Algebraic semantics of EMOF/OCL metamodels," Department of Computer Science, University of Illinois at Urbana-Champaign, USA, Tech. Rep. UIUCDCS-R-2007-2904, October 2007, URL: http://hdl.handle.net/2142/11398; Accessed on 5 Feb. 2010.

[28] A. D. Brucker, J. Doser, and B. Wolff, "Semantic issues of OCL: Past, present, and future," in *Proc. of the 6th OCL Workshop at the UML/MoDELS'06*, 2006, pp. 213–228.

[29] E. Gasparis, J. Nicholson, and A. H. Eden, "LePUS3: An object-oriented design description language," in *Proc. of Diagrams'08*, LNCS 5223, Springer. Sept. 2008, pp. 364–367.

[30] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien, "Instantiating and detecting design patterns: Putting bits and pieces together," in *Proc. of ASE'01*. 2001, pp. 166–173.