Front. Comput. Sci., 20xx, x(x)
DOI 10.1007/s11704-009-0000-0

RESEARCH ARTICLE

An Institution Theory of Formal Meta-Modelling in Graphically Extended BNF

Hong ZHU

Department of Computing and Electronics, Oxford Brookes University, Oxford, UK

Abstract Meta-modelling plays an important role in model driven software development. GEBNF is a graphic extension of BNF. It is proposed to define the abstract syntax of graphic modelling languages. From a GEBNF syntax definition, a formal predicate logic language can be induced so that meta-modelling 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-modelling approach. We formally define the semantics of GEBNF and its induced predicate logic languages, then apply Goguen and Burstall's institution theory to prove that they form a sound and valid formal specification language for meta-modelling.

Keywords Meta-modelling, Modelling languages, Abstract syntax, Semantics, Graphic extension of BNF, Formal logic, Institution.

1 Introduction

In the past years, we have seen a rapid growth of research on model-driven software development, in which models are created and processed as the main artefacts of software engineering. By raising the level of abstraction in software development, models facilitate a wider range of automation covering all phases and aspects of software development including requirements analysis, architectural and detailed design, code generation, integration, testing, maintenance, reverse engineering and evolution, and so on. Automated software tools and development environments have been developed to support model construction, model analysis, model transformation, and model-based software testing. However, despite of the great effort in the research on modelling languages and model-based software development tools, the correctness of modelling tools remains an open question. It is crucial to formally specify software modelling languages and tools since it is the basis of the verification, validation and testing of their correctness.

Formal specification of software systems has been a significant challenge to both communities of formal methods and software engineering for at least three decades [2]. The advent of model-driven methodology raises the stakes because modelling languages and tools are software systems one level higher than application software. They are languages to model software systems and tools to process software systems. In UML's terminology, they are at *meta-model* layer [3].

A meta-model is a model of models. Meta-modelling is to define a set of models that have certain structural and/or behavioural features by means of modelling. It is the approach adopted by OMG's model-driven architecture [4] and popular among researchers and practitioners in model-driven software engineering. It plays three key roles, and often a combination of them, in model-driven software development methodologies.

First, meta-models have been used to define modelling languages by specifying both the syntax and semantics. Currently, the syntax of a modelling language is usually defined at the abstract syntax level, while the semantics is usually specified in the form of an ontology, which presents a set of basic concepts and their interrelationships underlying the models. For example, the meta-model for UML defines the abstract syntax of UML modelling language in a class diagram that contains a set of concepts represented as meta-classes and a set of relationships between them represented as association, in-

This paper is an extended and revised version of the conference paper [1] presented at the *4th IEEE Symposium on Theoretical Aspects of Software Engineering* (TASE 2010).

heritance and aggregation relations between these meta-classes [5]. Many other languages can also be defined in this way, such as CWM, SPEM, XMI, etc. [3]. The transformations of a model into other types of software artefacts can be regarded as translation between different modelling languages.

Second, meta-models have been used to impose restrictions on an existing modelling language so that only a subset of the syntactically valid models are considered as its valid instances. For example, specifying design patterns is widely considered as a meta-modelling problem. Each design pattern can be defined as a meta-model so that only its instances are designs that conform to the pattern [6–8]. Checking if a model has certain structural and/or behavioural properties is therefore equivalent to check its conformance to a particular meta-model.

Finally, meta-models have also been used to extend existing meta-models 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 model elements that are specific to certain software development platforms. In [9], a meta-model was proposed for aspect-oriented modelling by extending the UML meta-model with basic concepts of aspect-orientation, such as cross-cut points, etc. Vertical development activities such as transformation of platform independent models to platform specific models and then to implementations can be regarded as mappings from one modelling language to another with certain consistency constraints.

Due to the importance of meta-modelling, growing research efforts on meta-modelling have been made in the past few years. In our previous work [10], we have proposed a formal meta-modelling approach, which includes

- a meta-notation called GEBNF, which stands for *Graphic Extension of BNF*, for the definition of abstract syntax of modelling languages, and
- a technique that induces formal predicate logic languages (FPL) from GEBNF syntax definitions.

In our approach, meta-modelling is performed by defining the abstract syntax of a modelling language in GEBNF and formally specifying the constraints on models in the formal logic language induced from GEBNF. Formal reasoning about meta-models can be supported by automatic or interactive inference engines. Transformation of models can be specified as mappings and relations between GEBNF syntax definitions together with translations between the predicate logic formulas. In particular, we have demonstrated the following uses of our approach in the quality assurance of model-driven software development tools.

- *Definition of graphic modelling languages*: A non-trivial subset of UML, including class diagrams and sequence diagrams, has been defined in GEBNF [8, 11]. Case studies have also been conducted successfully to specify the abstract syntax of the graphical software architecture description language ExSAVN [12] and agent-oriented software modelling language CAMLE.
- *Formal specification of models' structural and behavioural properties*: All the design patterns in the Gang-of-Four book [13] have been formalised by specifying the structural and behavioural properties of UML design models in the induced FPL [8,11]. A set of consistency constraints on UML models have also been formally specified in the FPL.
- *Automated checking of models' properties*: A formal specification of model's properties can be directly used in automated modelling tools as an input. For example, an automated design pattern recognition tool called LAMBDES-DP has been developed successfully by employing the theorem prover SPASS [14]. The formal specifications of design patterns are included in the tool as a repository. Reasoning about meta-models, such as proving a design pattern is a sub-pattern of another and the composition of patterns, has also been explored [15].
- *Formal specification of and reasoning about model transformations*: A set of pattern composition operators have been formally defined [16] and their algebraic properties proved on bases of FPL [17].

In this paper, we further advance the approach by laying a solid theoretical foundation via formally defining the semantics of GEBNF meta-notation and proving that GEBNF syntax definitions and their induced formal logics form an institution of formal specification for meta-modelling [18].

The paper is organized as follows. Section 2 gives an introduction to the GEBNF meta-modelling approach. Section 3 investigates how syntactic constraints imposed by GEBNF meta-notation can be represented as predicates in the induced FPL. Section 4 formally defines the semantics of GEBNF and its induced FPL by applying the model theory of mathematical logics. Section 5 studies the theoretical properties of GEBNF and its induced formal logic systems in the framework of institution theory. Finally, Section 6 concludes the paper with a discussion of related works and future work.

2 Overview of GEBNF

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

2.1 The Meta-Notation

Similar to the syntax definitions of programming languages in BNF, a syntax definition of a modelling language in GEBNF consists of a set of syntax rules that contain non-terminal symbols and terminal symbols. GEBNF extends BNF by bringing in two facilities. The first is called *labelled fields*. It requires each field in a syntax construction is labelled by a unique name. Therefore, these labels form a set of function symbols in the signature of a FPL. The second is the facility for *referential occurrences* of non-terminal symbols in the definition of a syntax construction so that non-linear structures like graphs can be defined.

In GEBNF, the abstract syntax of a modelling language is a 4-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, such as *String*, 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. Each syntax rule can be in one of the following two forms.

$$Y ::= X_1|X_2|\cdots|X_n \qquad (1)$$

$$Y ::= f_1 : E_1, f_2 : E_2, \cdots, f_n : E_n \qquad (2)$$

where $Y \in N$, X_1, X_2, $\cdots$, $X_n \in T \cup N$, f_1, f_2 , $\cdots$, f_n are *field names*, and E_1, E_2, $\cdots$, E_n are *syntax expressions*, which are inductively defined as follows.

- C is a basic syntax expression, if C is a literal instance of a terminal symbol, such as a string or a number.
- X is a basic syntax expression, if $X \in N \cup T$.
- $\underline{X@Z.f}$ is a basic syntax expression, if $X, Z \in N$, and f is a field name in the definition of Z, and X is the type of f field in Z's definition. The non-terminal symbol X is called a *referential occurrence*.
- E^*, E^+ and $[E]$ are syntax expressions, if E is a basic syntax expression.

Informally, each terminal and non-terminal symbol denotes a type of elements that may occur in a model. Each terminal symbol denotes a set of predefined basic elements. For example, the terminal symbol *String* denotes the set of strings of characters. Non-terminal symbols denote the constructs of the modelling language. The elements of the root symbol are the models of the language.

If a non-terminal symbol Y is defined in the following form,

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

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. And, if a is an element of type Y, we write $a.f_k$ for the k'th element of a.

For the sake of convenience, we also write $\underline{X@Z}$ and $\underline{X}$ as abbreviation of $\underline{X@Z.f}$ when there is no risk of confusion.

If a non-terminal symbol Y is defined in the form of

$$Y ::= X_1|X_2|\cdots|X_n,$$

it means that an element of type Y can be an element of type X_i, where $1 \le i \le n$.

The meaning of the meta-notation is informally explained in Table 1.

Example 1 *(Directed Graphs)*

The following is a definition of the abstract syntax of directed graphs in GEBNF. In the sequel, it will be referred to as **DG** and used throughout the paper to illustrate the notions and notations.

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

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

The first syntax rule states that a graph consists of a non-empty set of nodes and a set of edges. The second rule states that each node has a name, which is a string of characters, and it may have an optional weight, which is a real number. Finally, the third rule states that each edge refers to two nodes in the graph; one is referred to as the '*from*' node and the another as the '*to*' node. And, each edge has a weight, which is a real number. □

2.2 Well-Formed Syntax Definitions

If a non-terminal symbol $X \in N$ occurs on the right-hand-side of the definition of a non-terminal symbol Y, we say that X is *directly reachable* from Y. For example, *Node* and *Edge* are directly reachable from *Graph* through field names *nodes* and *edges*, respectively.

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 definitions.

Definition 1 *(Well-Formed Syntax Definition)*
A GEBNF syntax definition $\mathbf{G} = \langle R, N, T, S\rangle$ *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., X is on the left-hand-side of s.*

Table 1 Meanings of GEBNF Notation

Notation	Meaning	Example
$X*$	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.	$Assoc ::= end : \underline{Node@ClassDiag.classes}$: An association has an *end* that refers to an existing node in the field of *classes* of $ClassDiag$.

2. Reachability. *For each non-terminal symbol $X \in N$, X is reachable from the root R.* □

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

2.3 Induced Predicate Logic 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 mapping from a graph to two types of elements in the graph: 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 non-terminal symbol X, we will also use IsX to check if an element x is of type X. This is useful only if X occurs in a definition in the form of "$Y ::= ...|X|...$". Thus, the type of IsX is $Y \to Bool$.

In general, 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.

Definition 2 *(Types)*
Let $G = \langle R, N, T, S\rangle$ be a GEBNF syntax definition. The set of types of G, denoted by $Type(G)$, is defined inductively as follows.

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. *$\tau_1 \to \tau_2$ 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 "$A ::= B_1|B_2|\cdots|B_n$" introduces a set of function symbols IsB_i $(i = 1, \cdots, n)$ of type $A \to Bool$.

A syntax rule "$A ::= f_1 : B_1, \cdots, f_n : B_n$" introduces a set of function symbols f_i $(i = 1, \cdots, n)$ of type $A \to \Gamma(B_i)$, where $\Gamma(B_i)$ is defined as follows.

- $\Gamma(B) = B$, *if $B \in T \cup N$;*
- $\Gamma(B) = C$, *if $B = [C]$ and $C \in T \cup N$;*
- $\Gamma(B) = \Gamma(C)$, *if $B = \underline{C@Z.f}$;*
- $\Gamma(B) = \mathcal{P}(\Gamma(C))$, *if $B = C^*$ or $B = C^+$.* □

Table 2 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	$Edge \to Real$

Example 2 *(Induced Functions)*

The functions induced from the GEBNF syntax definition of directed graphs are given in Table 2. □

We also 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.

Given a well-defined GEBNF syntax $\mathbf{G} = \langle R, N, T, S\rangle$ of a modelling 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 FPL can be defined as usual (C.f. [19]) 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* $\exists$.

Definition 4 *(Inducted Predicate Logic)*
Let $\mathbf{G}$ be any given well-formed GEBNF syntax definition. The FPL induced from $\mathbf{G}$, denoted by FPL_G is defined inductively as follows.

Let $V = \bigcup_{\tau \in Type(G)} V_\tau$ be a collection of disjoint sets of variables, where each $x \in V_\tau$ is a variable of type τ, and V is disjoint to $Fun(\mathbf{G})$.

1. *Each literal constant c of type $s \in T$ is an expression of type s.*
2. *Each element v in V_τ, i.e. variable of type τ, is an expression of type $\tau \in Type(G)$.*
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. *$\{e(x)|Pred(x)\}$ 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 on type τ, if p and q are predicates on type τ.*
11. *$\forall x \in D \cdot (p(x))$ and $\exists x \in D \cdot (p(x))$ are predicates on type $\mathcal{P}(\tau)$, if D is a type τ, x is a variable of type τ, and $p(x)$ is a predicate on type τ.* □

For the sake of convenience, given an expression S of type $\mathcal{P}(\tau)$, we will also write $\forall x \in S \cdot (p(x))$ as abbreviation of the expression $\forall x \in \tau \cdot (x \in S \Rightarrow p(x))$ and $\exists x \in S \cdot (p(x))$ as abbreviation of $\exists x \in \tau \cdot (x \in S \wedge p(x))$.

In a FPL_G, functions and relations can be defined as usual. For the sake of readability, we will use a mixture of infix and prefix forms for defined functions and relations. Thus, we may also write the application of function f to argument x in 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 can be formally defined as follows using the functions induced from the syntax definition.

$$\begin{aligned}&UnweightedNodes(g : Graph) \triangleq \\ &\quad \{n | n \in g.nodes \wedge n.weight = \perp\}\end{aligned}$$

where $\perp$ means undefined. □

2.4 Meta-Modelling

Given the abstract syntax of a modelling language defined in GEBNF, meta-modelling within the framework of the modelling language can be performed by defining a predicate p such that the required subset of models are those that 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 FPL_G.

Example 4 *(Meta-modelling)*

Consider **DG** in Example 1. The set of strongly connected graphs can be defined as the set of models that satisfy the following condition.

$$\begin{aligned}&StronglyConnected(g : Graph) \triangleq \\ &\quad \forall x, y \in g.nodes \cdot (x = y \vee \\ &\qquad ((x \ reaches \ y) \wedge (y \ reaches \ x)),\end{aligned}$$

where the predicate $(x \ reaches \ y) : Node \times Node \rightarrow Bool$ is defined as follows.

$$\begin{aligned}&(x \ reaches \ y) \triangleq \\ &\quad \exists e \in g.edges \cdot (x = e.from \wedge y = e.to) \vee \\ &\quad \exists z \in g.nodes \cdot ((x \ reaches \ z) \wedge (z \ reaches \ y))\end{aligned}$$

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

$$\begin{aligned}&Acyclic(g : Graph) \triangleq \\ &\quad \forall x, y \in g.nodes \cdot ((x \ reaches \ y) \Rightarrow x \neq y).\end{aligned}$$

The set of connected graphs can be defined as follows.

$$\begin{aligned}&Connected(g : Graph) \triangleq \\ &\quad \forall x, y \in g.nodes \cdot (x \neq y \Rightarrow \\ &\qquad (x \ reaches \ y) \vee (y \ reaches \ x)).\end{aligned}$$

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

$$\begin{aligned}&Tree(g : Graph) \triangleq \\ &\quad Connected(g) \wedge Acyclic(g) \wedge \\ &\quad \exists x \in g.nodes \cdot (\forall y \in g.nodes \cdot (x \ reaches \ y)) \wedge \\ &\quad \forall e, e' \in g.edges \cdot (e.to = e'.to \Rightarrow e = e')\end{aligned}$$

□

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 [8, 11].

3 Axiomatization of Syntax Constraints

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

3.1 Optional Elements

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

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

The function f has the type $A \rightarrow B$, 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.

$$\forall x \in A \cdot (x.g \neq \perp), \quad (3)$$

where $\perp$ means undefined.

Example 5 *(Partial and Total 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.

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

□

3.2 Non-Empty Repetitions

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

$$A ::= \cdots , f : B^*, \cdots \quad (4)$$
$$A ::= \cdots , g : B^+, \cdots \quad (5)$$

The functions f and g induced from the above syntax rules are of the same type, i.e. $A \to \mathcal{P}(B)$. However, in case of (4), an element of type A may contain an empty set of elements of type B; while in case of (5), it can only contain a non-empty set of elements of type B. In other words, 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 empty occurrence. Therefore, the function *nodes* must satisfy the following axiom, but the function *edges* does not.

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

□

3.3 Referential and Creative Elements

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

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

Informally, 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 $A \to B$ 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 $e.from \neq e'.from$ and $e.to \neq e'.to$. 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 $e \in Edges$, we have $e.from \in g.nodes$ and $e.to \in g.nodes$ for $g \in Graph$. There is no any further restriction on $e \in Edge$. In other words, it allows $e.from = e.to$, $e.from = e'.from$, $e.to = e'.to$ and $e.from = e'.to$ to be true for some edges e and e'.

In general, the function symbols induced from creative occurrences of the 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,

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

When both $E(X)$ and $E'(X)$ are in the form of X and $[X]$ for $X \in N$, we require functions f and g satisfying the condition

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

When both $E(X)$ and $E'(X)$ are in the form of X^* and X^+ for $X \in N$, we require functions f and g satisfying the condition

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

Similarly, when $E(X)$ is in the form of X and $[X]$, but $E'(X)$ is in the form of X^* and X^+, we require functions f and g satisfying 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.

Suppose that two syntax rules are as follows:

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

When $E(X)$ is in one of the forms X and $[X]$, we require functions f and g satisfying the condition

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

When $E(X)$ is in one of the forms X^* and X^+, we require functions f and g satisfying the condition

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

Suppose that two syntax rules are in the form of

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

where $E(X)$ and $E'(X)$ are in any of the forms X^* and X^+. Then, we require functions f and g satisfying the following condition.

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

It is worth noting that the above constraints are in the predicate logic language 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.

$$\forall g \in Graph \cdot \forall e \in Edge \cdot (e.from \in g.nodes)$$
$$\forall g \in Graph \cdot \forall e \in Edge \cdot (e.to \in g.nodes)$$

□

Note that, the above conditions on edges may look ridiculous since one may read it as requiring the nodes associated to an edge to be in the set of nodes '*for all graphs g*'. However, it is correct, because *Graph* is the root non-terminal symbol, which we only allow the existence of one element of the type to represent a model in the language. Therefore, '$\forall g \in Graph$' should be read as '*for the graph g*'.

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

Example 8 *(Syntax Constraints)*

Consider the GEBNF syntax definition **DG** given in Example 1. The set *Axiom*(*DG*) contains the following predicates.

$$\forall g \in Graph \cdot \forall e \in Edge \cdot (e.from \in g.nodes)$$
$$\forall g \in Graph \cdot \forall e \in Edge \cdot (e.to \in g.nodes)$$
$$\forall g \in Graph \cdot (g.nodes \neq \emptyset)$$
$$\forall n \in Node \cdot (n.name \neq \bot)$$
$$\forall e \in Edge \cdot (e.from \neq \bot)$$
$$\forall e \in Edge \cdot (e.to \neq \bot)$$
$$\forall e \in Edge \cdot (e.weight \neq \bot)$$
$$\forall g \in Graph \cdot (g.nodes \neq \bot)$$
$$\forall g \in Graph \cdot (g.edges \neq \bot)$$

4 Algebraic Semantics

This section formally defines the semantics of GEBNF by regarding models as mathematical structures that satisfy the conditions imposed by the abstract syntax.

4.1 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

$$F_G = Fun(\mathbf{G}) \cup \bigcup_{s \in T} (Op_s \cup R_s).$$

Σ_G is called the *signature* induced from **G**.

Definition 5 *(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 $[\![X]\!]^T$ to the set $[\![Y]\!]^T$, where for each type τ, $[\![\tau]\!]^T$ is the semantics of the type τ defined as follows.

$$[\![\tau]\!]^T = \begin{cases} A_\tau, & if\ \tau \in N \cup T; \\ \mathbb{P}([\![\tau']\!]^T), & if\ \tau = \mathcal{P}(\tau'); \\ ([\![\tau_1]\!]^T \to [\![\tau_2]\!]^T), & if\ \tau = (\tau_1 \to \tau_2). \end{cases}$$

where $\mathbb{P}(X)$ is the power set of X, $(X \to Y)$ is the set of partial or total functions from X to Y. □

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.

We assume that the mathematical structure $\langle A_s, Op_s \cup R_s \rangle$ is fixed for all GEBNF syntax definitions. But, its detail is not important, thus omitted in this paper.

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

Definition 6 *(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 FPL such that for some $m \in A_R$ we have $e \in f(m)$.* □

Informally, we consider a Σ_G-algebra $\mathcal{A}$ as a model in the modeling language. Condition (1) means that there is only one root element. This is similar to the condition that a parsing tree of a program must have one and only one root. Condition (2) means that 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 parsing 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.

Example 9 *(A Model as an Algebra)*

Consider the directed graph shown in Fig. 1. It is a model of Example 1. It can be represented as a Σ_G-algebra as follows.

Carrier sets:
$Graph = \{g\}, Node = \{a, b, c, d\}, Edge = \{ab, ac, ad, bd\}$
Functions:
$nodes : Graph \to Node : \quad g.nodes = \{a, b, c, d\}$
$edges : Graph \to Edge : \quad g.edges = \{ab, ac, ad, bd\}$
$name : Node \to String :$
$a.name = 'a', \quad b.name = 'b',$
$c.name = 'c', \quad d.name = 'd'$
$weight : Node \to Real :$
$a.weight = 4.5, \quad b.weight = \bot,$
$c.weight = 2.6, \quad d.weight = \bot.$
$from : Edge \to Node :$
$ab.from = a, \quad ac.from = a,$
$ad.from = a, \quad bd.from = b$
$to : Edge \to Node :$
$ab.to = b, \quad ac.to = c, \quad ad.to = d, \quad bd.to = d$
$weight : Edge \to Real :$
$ab.weight = 0.1, \quad ac.weight = 0.5,$
$ad.weight = 0.3, \quad bd.weight = 1.2$

Note that, the above mathematical structure has no junk. In particular, we have that $|Graph| = 1$; thus, condition (1) of no junk holds. And, we also have that $Node = g.nodes$ and $Edge = g.edges$; thus, condition (2) holds.

If we modify the structure slightly by adding one more element e to the carrier set $Node$ (i.e. $Node = \{a, b, c, d, e\}$), it contains a junk element e, which cannot be reached from g. □

Note that, R is the root non-terminal symbol.

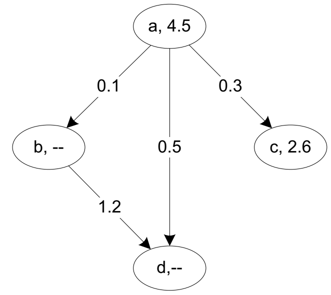

Fig. 1 An Example of Directed Graph

4.2 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 condition represented in the form of a predicate or statement in the FPL.

An assignment α to a set V of variables in an Σ-algebra $\mathcal{A}$ is a mapping from the set V to the elements of the algebra such that for each variable v of type τ, we have that $\alpha(v) \in [\![\tau]\!]^T$.

Definition 7 *(Evaluation of Expressions)*
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 $\tau \in T$;*
- $[\![v]\!]_\alpha = \alpha(v) \in [\![\tau]\!]^T$, *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 $[\![D]\!]^T$, $[\![p]\!]_{\alpha[x/e]}$ is true;*
- $[\![\exists x \in D \cdot (p)]\!]_\alpha = True$, *if there exists e in $[\![D]\!]^T$ such that $[\![p]\!]_{\alpha[x/e]}$ is true.*

where $\alpha[x/e]$ is an assignment such that $\alpha[x/e](x) = e$ and for all $x' \neq x \in V$, $\alpha[x/e](x') = \alpha(x')$. □

Let α be an assignment in Σ_G-algebra $\mathcal{A}$ and p be a predicate in FPL_G.

Definition 8 *(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 $\mathcal{A} \models_\alpha p$. □

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

Definition 9 *(Syntactically Valid Models)*
A Σ_G-algebra $\mathcal{A}$ (with no junk) is a syntactically valid model of $\mathbf{G}$, 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 GEBNG syntax definition $\mathbf{G}$ and a statement p in FPL_G. The semantics of the meta-model MM is a subset of syntactically valid models of $\mathbf{G}$ that satisfy the statement p. □

Note that, the definition of satisfaction relation is a standard treatment of predicate logics in the model theory of mathematical logics [19]. When a model is finite, the truth of a statement about the model is decidable.

4.3 Logic Inference about Models

The truth of a statement about models can also be formally deducted by logic inferences, for example, by applying natural deduction. Let Γ be a set of predicates in FPL_G. In the sequel, we will write $\Gamma \vdash p$ to denote that p can be deduced from Γ in a given formal predicate logic inference system.

Definition 10 *(Truth of Sentences)*
Let $\mathbf{G}$ be any given well-formed GEBNF syntax definition. A predicate p in FPL_G is true, written $\models_G p$, if for all syntactically valid model $\mathcal{A}$ of $\mathbf{G}$, we have that $\mathcal{A} \models p$. □

The completeness and soundness of the formal inference system can be defined as follows.

Definition 11 *(Completeness and Soundness)*
The inference system is complete *if we have that $\models_G p$ if and only if $Axiom(G) \vdash p$. It is* sound *if we have that for all syntactically valid model $\mathcal{A}$, $Axiom(G) \vdash p \Rightarrow q$ and $\mathcal{A} \models p$ imply that $\mathcal{A} \models q$.* □

In the sequel, we will not be so specific about the inference system, but generally assume that the inference is sound. This assumption is reasonable because the definition of the semantics of FPL_G is a standard treatment in the model theory of mathematical logics. In particular, natural deduction is sound for FPL_G. However, we will not assume the inference system being complete, because it depends on the mathematical property of the semantics of the terminal symbols and also because the quantified variables in a predicate can be of a higher order type. The theory to be developed in the remainder of the paper can be established without the completeness property of the inference system.

A model is finite if A_s is a finite set for all $s \in N$.

5 Institution of Meta-models

As discussed in Section 1, meta-modelling often involves multiple meta-models. Each meta-model defines a FPL. 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 [18] and entailment systems. In this section, we apply these theories to GEBNF.

5.1 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* (also called *arrows*) between objects together with the following three operations:

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

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 $f : A \rightarrow B$. 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,

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

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 12 *(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.* □

Example 10 *(Syntax Morphism)*

The following is a GEBNF syntax definition **AR** of the models of flight routes for an airline.

$$
\begin{aligned}
Map &::= cities : City^{+}, routes : Route^{*}\\
City &::= name, country :: String,\\
&\quad\;\; population : Real\\
Route &::= depart, arrive : City,\\
&\quad\;\; distance : Real, flights : TimeDay^{*}
\end{aligned}
$$

We define a syntax morphism from **DG** to **AD** by two mappings m and f as follows.

$$
\begin{aligned}
m &= (Graph \to Map, Node \to City, Edge \to Route),\\
f &= (nodes \to cities, edges \to routes,\\
&\quad name \to name, weight \to population,\\
&\quad to \to arrive, from \to depart, weight \to distance)
\end{aligned}
$$

It is easy to prove that these mappings preserve the root (i.e. $m(Graph) = Map$) and the types. Therefore, they form a syntax morphism from the GEBNF syntax definition **DG** given in Example 1 to **AR**. □

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

Definition 13 *(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 14 *(Identity Syntax Morphisms)*
For all $\mathbf{G} = \langle R, N, T, S \rangle$, the identity syntax morphism *of $\mathbf{G}$, denoted by 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 GEBNG 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 GEBNG 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 $\mathbb{SYN}$ in the sequel.
Proof. *The theorem directly follows Lemma 1 and 2.* □

5.2 Translation of Sentences

Given a syntax morphism from one GEBNF definition to another, we can define a translation between the FPLs 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* $\mathcal{F}$ 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 $\mathbb{SYN}$ of GEBNF syntax definitions to the category $\mathbb{SEN}$ of the sets of predicates in the FPL induced from GEBNF syntax definitions with morphisms being mappings between sets.

Definition 15 *(Category $\mathbb{SEN}$)*
Let $Sen(G) = \{p | p$ is a predicate in $FPL_G\}$, and

$$Sen_{obj} = \{Sen(G) | G \text{ is a } GEBNF \text{ syntax definition}\}.$$

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

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 $Sen_m(\mu)$. We now define

$$Sen_m = \{Sen_m(\mu) | \mu \text{ is a syntax morphism}\}.$$

□

It is easy to prove that Sen_{Obj} as objects and Sen_m as morphisms form a category, which is referred to by $\mathbb{SEN}$.

Lemma 3 $\mathbb{SEN} = \langle Sen_{Obj}, Sen_m \rangle$ *is a category.* □

Example 11 *(Translation of Sentence)*

Consider the syntax morphism defined in Example 10. The *reaches* predicate defined in Example 4 can be translated into the following sentence in FPL_{AR}.

$$\begin{aligned}&(x \text{ reaches } y) \triangleq \\ &\quad \exists e \in g.routes \cdot (x = e.depart \wedge y = e.arrive) \vee \\ &\quad\quad \exists z \in g.cities \cdot ((x \text{ reaches } z) \wedge (z \text{ reaches } y))\end{aligned}$$

□

Note that Sen is a mapping from objects in the category $\mathbb{SYN}$ to objects of $\mathbb{SEN}$. And, Sen_m is a mapping from morphisms of $\mathbb{SYN}$ to morphisms of category $\mathbb{SEN}$. Does the pair form a functor? The following theorem proves that (Sen, Sen_m) is a functor indeed.

Theorem 2 *(Soundness of the Definition of Functor Sen)*
The pair (Sen, Sen_m) is a functor from category $\mathbb{SYN}$ of GEBNF syntax definitions to the category $\mathbb{SEN}$. In the sequel, we use $\mathcal{SEN}$ to denote this functor.
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 $Sen(G)$, $Sen_m(\mu)(p)$ 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 $Sen(G)$. This directly follows the definition of Id_G. □

5.3 Constraint Preserving Syntax Morphisms

Let μ be a syntax morphism from $\mathbf{G}$ to $\mathbf{H}$. We require the syntax morphism to preserve the conditions such as an element is a referential occurrence and non-optional occurrence, etc. Thus, we define the notion of constraint preserving syntax morphisms as follows.

Definition 16 *(Constraint Preserving Morphisms)*
A syntax morphism μ from $\mathbf{G}$ to $\mathbf{H}$ is constraint preserving *if for all constraint $c \in Axiom(G)$ we have that $Axiom(H) \vdash Sen_\mu(c)$.* □

Example 12

Consider the syntax morphism given in Example 10. It is constraint preserving because for each constraint c in $Axiom(DG)$, which is given in Example 8, we can prove that $Axiom(AR) \vdash c'$, where c' is the translation of c into PL_{AR} according to the syntax morphism. For instance, the following constraint c on directed graph

$$c \triangleq \forall g \in Graph \cdot (g.nodes \neq \emptyset)$$

is translated into

$$c' \triangleq \forall g \in Map \cdot (g.cities \neq \emptyset).$$

according to the syntax morphism. It is easy to see that $Axiom(AR) \vdash c'$ because $c' \in Axiom(AR)$. □

Informally, constraint preserving means that the syntax constraints that GEBNF syntax definition $\mathbf{G}$ imposes on models are all satisfied by the modelling language defined by $\mathbf{H}$ when the notations in $\mathbf{G}$ is translated into notations in $\mathbf{H}$. The following theorem states that such constraint preserving syntax morphisms form a full subcategory of $\mathbb{SYN}$.

Theorem 3 *(Constraint Preservation Sub-Category)*
The set of well-formed GEBNF syntax definitions as objects and the set of constraint preserving syntax morphisms between them as morphisms form a category and this category is a full sub-category of $\mathbb{SYN}$, because the following statements are true.

1. *For all well-formed GEBNF syntax definition $\mathbf{G}$, Id_G is constraint preserving.*
2. *If μ and ν are constraint preserving syntax morphisms, so is $\mu \circ \nu$ provided that they are composable.*

Proof. *Statements 1) and 2) follow the logic properties of $\vdash$. Thus, the theorem is true.* □

In the sequel, we will use $\mathbb{GEBNF}$ to denote the constraint preserving sub-category of GEBNF syntax definitions.

5.4 Translation of Models

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

We first observe that the models in any given modelling 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 syntactical valid models of $\mathbf{G}$ by $Mod(G)$. The following defines the homomorphisms between models.

Definition 17 *(Homomorphisms between Models)*
Let $\mathcal{A}$ and $\mathcal{B}$ be syntactical valid models of $\mathbf{G}$, a homomorphism φ from $\mathcal{A}$ to $\mathcal{B}$ is a mapping $\varphi : A \to B$ such that, for all $s \in N \cup T$,

$$\forall x \in A_s \cdot (\varphi(x) \in B_s)),$$

and, for all $f \in F_G$, we have that

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

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

Lemma 4 *(Category of Models)*
For any given well-formed GEBNF syntax definition $\mathbf{G}$, the set of syntactically valid models of $\mathbf{G}$ as the set of objects and homomorphisms between models as the set of morphisms form a category, where for each model $\mathcal{A}$, 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 $\mathbf{G}$ varying over the set of GEBNF syntax definitions, and the morphisms are functors U_μ between these categories of models, where μ varies over the syntax morphisms 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 5 *(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 $\mathbb{MOD}_G$. □

Example 13 *(Translation of model)*

Consider the model of **AR** shown in Figure 2(a). It can be translated into the model of directed graph shown in (b) when the syntax morphism defined in Example 10 is applied. □

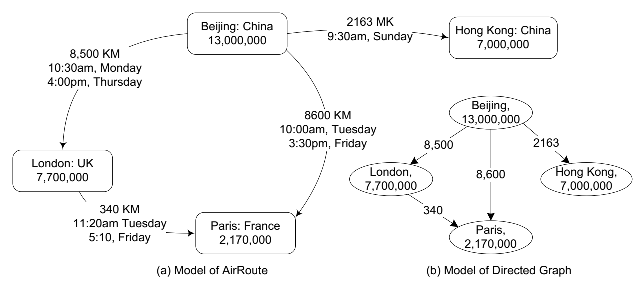

Fig. 2 Example of Translation of Models

Furthermore, we have the following theorem.

Theorem 4 *(Category of Modelling Languages)*
Let $Obj = \{\mathbb{MOD}_G | G \in |\mathbb{GEBNF}|\}$ and $Mor = \{U_\mu | \mu \in ||\mathbb{GEBNF}||\}$. (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 18 *(Model Translation)*
We define mappings $MOD_{obj} : |\mathbb{GEBNF}| \to |\mathbb{CAT}^{op}|$ and $MOD_m : ||\mathbb{GEBNF}|| \to ||\mathbb{CAT}^{op}||$ 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 5 *(Functor of Model Translation)*
MOD is a functor from $\mathbb{GEBNF}$ to $\mathbb{CAT}^{op}$. □

5.5 Institution of GEBNF

We are now ready to prove that GEBNF and its induced predicate logics form an institution. First let's review the notion of institution [18].

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 $(\models_\Sigma)$ called Σ-satisfaction, where for each $\Sigma \in |Sig|$, $\models_\Sigma \subseteq |Mod(\Sigma)| \times Sen(\Sigma)$. It must satisfy the condition that for any $(\phi : \Sigma \to \Sigma') \in ||Sig||$, any $M' \in |Mod(\Sigma')|$ and any $e \in Sen(\Sigma)$,

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

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

Theorem 6 *(GEBNF Institution)*
The tuple $(\mathbb{GEBNF}, MOD, Sen, \models)$ is an institution, where

1. *$\mathbb{GEBNF}$ is the category of well-formed GEBNF syntax definitions as proved in Theorem 3;*
2. *MOD is defined in Definition 18;*
3. *Sen is defined in Definition 2; and*
4. *$\models$ is the satisfaction relation defined in Definition 8.*

Proof.
The condition 1) of institution is true by Theorem 3.
Condition 2) is true by Theorem 2.
Condition 3) is true by Theorem 5.
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. □

Example 14 *(Truth Invariance under Translation)*

Let predicates $(x\ reaches_{DG}\ y)$ and $(x\ reaches_{AR}\ y)$ be the predicates defined in Example 4 and 13, respectively. Note that former is translated into the later by applying the sentence translation functor Sen with the syntax morphism μ defined in Example 10. Let $\mathcal{A}$ be the model given in Figure 2(a) and $\mathcal{B}$ be the model obtained by translation of $\mathcal{A}$ using syntax morphism μ. In fact, according to Example 13, $\mathcal{B}$ is given in Figure 2(b). It is easy to see that both statements

$$\mathcal{B} \models (Beijing\ reaches_{DG}\ Paris)$$

and

$$\mathcal{A} \models (Beijing\ reaches_{AR}\ Paris)$$

are true. And, both statements

$$\mathcal{B} \models (Hong\ Kong\ reaches_{DG}\ Paris)$$

and

$$\mathcal{A} \models (Hong\ Kong\ reaches_{AR}\ Paris)$$

are false. These are instances of the condition 4) of institution. □

6 Conclusion

6.1 Summary

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

- We have formally defined the semantics of GEBNF syntax definitions as algebras without junk and satisfying a set of constraints written in the induced FPL. These constraints are derived from the syntax rules in GEBNF. We have 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 modelling languages. Thus, this lays a solid foundation for model transformations and extension mechanisms of meta-modelling.
- We have also proved that the category of GEBNF syntax definitions, the categories of models in any given modelling language defined by GEBNF and the satisfaction relation form an institution. Therefore, GEBNF syntax definitions and the induced FPL form a valid specification language for meta-modelling.

6.2 Related work

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

UML class diagrams has been used as a general purpose meta-modelling language in MOF's four-layer architecture of UML language definition. In such a meta-model, the basic concepts of a modelling 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 associated to meta-classes and meta-relations, such as derived property, default values, etc.

There are two long lasting issues concerning the UML meta-modelling approach. First, the semantics of meta-models is informally defined. There is few research efforts to formalize the semantics of UML meta-models [20–22]. In [20], Shan and Zhu separated the descriptive semantics and functional semantics of UML models and formally defined the notion of 'instance-of' relation between meta-models and models. Poernomo [21] formalised the semantics of meta-models by applying constructive type theory. The semantics of MOF was defined as a higher order lambda-calculus expression. Boronat and Meseguer [22] used the Maude language that directly supports membership equational logic to specify the semantics of MOF as an executable specification so that whether a model is an instance of a meta-model can be determined. While these works help to clarify the key notion of 'instance-of' relation between meta-model and models, further research is required to address many other issues related to meta-modelling discussed in Section 1. The second is the weakness of graphic notation in its expressiveness and accuracy. This can be 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 closer to object oriented programming languages. Attempts to formalize the semantics of OCL have been reported in [23–28], etc. However, it is still unsatisfactory in the formal definition of OCL's semantics and understanding of its logic properties [29, 30]. Moreover, how to connect OCL to the formal semantics of MOF as defined in [20–22] is still unclear.

Many special purpose meta-modelling languages have been proposed, mostly for defining design patterns. Typical examples are LePUS [31, 32], RBML [6], DPML [33, 34], and PDL [35]. They all use graphic notation to represent meta-models. In general, graphic meta-modelling 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 while they can specify what must be in a model.

6.3 Future work

For future work, we are considering developing software tools to support meta-modelling in GEBNF. Further application of the theory to facilitate a meta-model extension mechanism is worthy investigating. 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.

Acknowledgements The author would like to thank the members of the Applied Formal Methods Research Group at the Oxford Brookes University, especially Dr. Ian Bayley and Dr. Mark Green, for valuable discussions on related topics and comments on an early version of the paper. The author is most grateful to Dr. Lijun Shan, Dr. Ian Bayley and Mr. Richard Amphlett for their collaboration in the research on related topics including applications of GEBNF to the formalisation of software design patterns and the formalisation of UML semantics.

References

1. Zhu H. On the theoretical foundation of meta-modelling in graphically extended BNF and first order logic. In: Proceedings of the 4th IEEE Symposium on Theoretical Aspects of Software Engineering (TASE'10), Taipei, Taiwan, August 2010, 95–104
2. Van Lamsweerde A. Formal specification: a roadmap. In: Proceedings of the 22nd International Conference on on Software Engineering - Future of SE Track (ICSE'00), Limerick, Ireland, June 2000, 147–159
3. OMG. Meta Object Facility (MOF) 2.0 Core Specification. Technical Report formal/2006-01-01, Object Management Group, Needham, MA 02494, USA, 2006. URL: http://www.omg.org/spec/MOF/2.0
4. OMG. MDA Specification. Object Management Group, Needham, MA 02494, USA, August 2010. URL: http://www.omg.org/mda/specs.htm, Last updated on 08/13/2010.
5. OMG. Unified Modeling Language: Superstructure, version 2.0. Technical Report formal/05-07-04, Object Management Group, Needham, MA 02494, USA, July 2005. URL: http://www.omg.org/spec/UML/2.0
6. France RB, Kim D-K, Ghosh S, Song E. A UML-based pattern specification technique. IEEE Transactions on Software Engineering, 2004, 30(3):193–206
7. Elaasar M, Briand LC, Labiche Y. A metamodeling approach to pattern specification. In: Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems, (MoDELS'06), Genova, Italy, October 2006, Lecture Notes in Computer Science, Vol. 4199, Springer, 2006, 484–498
8. Bayley I, Zhu H. Formal specification of the variants and behavioural features of design patterns. Journal of Systems and Software, 2010, 83(2):209–221
9. Mraidha C, Gerard S, Terrier F, Benzakki J. A two-aspect approach for a clearer behavior model. In: Proceedings of 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03), May 2003, 213–220
10. Zhu H, Shan L. Well-formedness, consistency and completeness of graphic models. In: Proceedings of the 9th

International Conference on Computer Modelling and Simulation (UKSIM'06), Oxford, UK, April 2006, 47–53
11. Bayley I, Zhu H. Specifying behavioural features of design patterns in first order logic. In: Proceedings of the IEEE 32nd International Computer Software and Applications Conference (COMPSAC'08), 2008, 203–210
12. Zhang Q. Visual software architecture description based on design space. In: Proceedings of The Eighth International Conference on Quality Software (QSIC'08), Oxford, UK, August 2008, 366–375
13. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, USA, 1995
14. Zhu H, Bayley I, Shan L, Amphlett R. Tool support for design pattern recognition at model level. In: Proceedings of the 33rd IEEE Annual Conference on Computer Software and Applications (COMPSAC'09), Seattle, Washington, USA, July 2009, 228–233
15. Bayley I, Zhu H. On the composition of design patterns. In: Proceedings of the 8th International Conference on Quality Software (QSIC'08), Oxford, UK, August 2008, 27–36
16. Bayley I, Zhu H. A formal language of pattern composition. In: Proceedings of The 2nd International Conference on Pervasive Patterns (PATTERNS'10), Lisbon, Portugal, Nov. 2010, 1-6
17. Zhu H, Bayley I. Laws of pattern composition. In: Proceedings of 12th International Conference on Formal Engineering Methods (ICFEM'10), Shanghai, China, Lecture Notes in Computer Science, Vol. 6447, Springer, Nov. 2010, 630-645
18. Goguen J, and Burstall RM. Institutions: Abstract model theory for specification and programming. Journal of ACM, 1992, 39(1):95–146
19. Chiswell I, Hodges W. Mathematical Logic, volume 3 of Oxford Texts in Logic. Oxford University Press, 2007
20. Shan L, Zhu H. Semantics of metamodels in UML. In: Proceedings of The 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09), Tianjin, China, July 2009, 55–62
21. Poernomo I. The meta-object facility typed. In: Proceedings of The 21st Annual ACM Symposium on Applied Computing (SAC'06), Haddad H (eds), Dijon, France, 2006, 1845–1849
22. Boronat A, Meseguer J. An algebraic semantics for MOF. Formal Aspects of Computing, 2010, 22(3-4):269–296
23. Cengarle MV, Knapp A. A formal semantics for OCL 1.4. In: Proceedings of 4th International Conference on The Unified Modelling Language–Modelling languages, Concepts and Tools (UML'01), Gogalla M, Kobryn C (eds), Lecture Notes in Computer Science, Vol. 2185, Springer, October 2001, 118–133
24. Clarke T, Warmer J. Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Lecture Notes in Computer Science, Vol. 2263, Springer, 2002
25. Brucker AD, Wolff B. A proposal for a formal OCL semantics in Isabelle/HOL. In: Proceedings of 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'02), Hampton, VA, USA, August 2002, Carreno VA, Munoz CA, Tahar S (eds), Lecture Notes in Computer Science, Vol. 2410, Springer, 147–175
26. Flake S. Towards the completion of the formal semantics of OCL 2.0. In: Proceedings of the 27th Australasian conference on Computer science (ACSC'04), Darlinghurst, Australia, 2004, 73–82
27. Hennicker R, Knapp A, Baumeister H. Semantics of OCL operation specifications. Electronic Notes in Theoretical Computer Science, November 2004, 102:111–132
28. Boronat A, Meseguer J. Algebraic semantics of OCL-constrained metamodel specifications. In: Proceedings of the 47th International Conference on Objects, Components, Models and Patterns (TOOLS EUROPE'09), Oriol M, and Meyer B (eds), Zurich, Switzerland, June - July 2009, Lecture Notes in Business Information Processing, Vol. 33, Springer, 96–115
29. Boronat A, Meseguer J. Algebraic semantics of EMOF/OCL metamodels. Technical Report UIUCDCS-R-2007-2904, Department of Computer Science, University of Illinois at Urbana-Champaign, USA, October 2007. URL: http://hdl.handle.net/2142/11398; Accessed on 5 Feb. 2010.
30. Brucker AD, Doser J, Wolff B. Semantic issues of OCL: Past, present, and future. In: Proceedings of the 6th OCL Workshop at UML/MoDELS'06, 2006, 213–228
31. Eden AH. Formal specification of object-oriented design. In: Proceedings of the International Conference on Multidisciplinary Design in Engineering, Montreal, Canada, November 2001
32. Gasparis E, Nicholson J, Eden AH. LePUS3: An object-oriented design description language. In: Proceedings of 5th International Conference on Diagrammatic Representation and Inference (Diagrams'08), Herrsching, Germany, September 2008, Lecture Notes in Computer Science, Vol. 5223, Springer,364–367
33. Maplesden D, Hosking J, Grundy J. A visual language for design pattern modelling and instantiation. In: Proceedings of 2001 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC'01), Stresa, Italy, September 2001, 338–339
34. Maplesden D, Hosking J, Grundy J. Design pattern modelling and instantiation using DPML. In: Proceedings of the Fortieth International Conference on Tools Pacific (TOOLS Pacific'02), Darlinghurst, Australia, 2002, 3–11
35. Albin-Amiot H, Cointe P, Guéhéneuc Y, Jussien N. Instantiating and detecting design patterns: Putting bits and pieces together. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01), San Diego, CA, USA, November 2001, 166–173