A new approach to the semantics of model diagrams

Johan G. Granström

Google
Brandschenkestrasse 110
8002 Zürich, Switzerland
georg.granstrom@acm.org

Department of Computer Science
King's College London
Strand, London, WC2R 2LS, U.K.

Abstract

Sometimes, a diagram can say more than a thousand lines of code. But, sadly, most of the time, software engineers give up on diagrams after the design phase, and all real work is done in code. The supremacy of code over diagrams would be leveled if diagrams were code. This paper suggests that model and instance diagrams, or, which amounts to the same, class and object diagrams, become first level entities in a suitably expressive programming language, viz., type theory.

The proposed semantics of diagrams is compositional and self-describing, i.e., reflexive, or metacircular. Moreover, it is well suited for metamodelling and model driven engineering, as it is possible to prove model transformations correct in type theory. The encoding into type theory has the additional benefit of making diagrams immediately useful, given an implementation of type theory.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, F.3.2 Semantics of Programming Languages.

Keywords and phrases model diagram, modelling, metamodelling, semantics, compositionality, self-description, metacircularity, reflexivity, universal model, MOF, UML.

Digital Object Identifier 10.4230/LIPIcs.TYPES.2011.28

1 Introduction

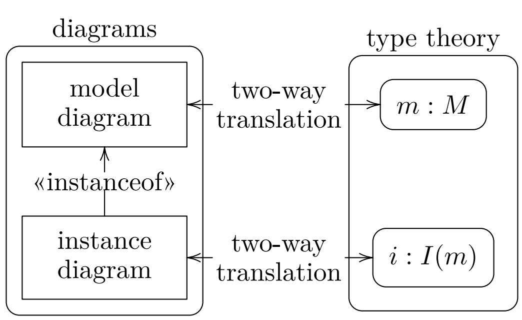

Figure 1 A modelling language (M, I) with corresponding model and instance diagrams.

The semantics of visual modelling languages, such as UML class diagrams, is surrounded by much confusion [21]. On the other hand, much is gained from using diagrams, as the same diagram can be understood to different degrees and from different angles by collaborators. In addition, with today's rapidly changing codebases, any documentation external to code is doomed to soon be out of date [30]. Consequently, documentation, in the form of diagrams, that is guaranteed to be in synch with code, because it is code, is worth much more than mere documentation.

18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson, Bengt Nordström; pp. 28–40
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Model diagrams can be translated to linear notation (Figure 1 and Sect. 4), and this linear notation can be completely formalized (Sect. 5) in a suitably expressive programming language like intuitionistic type theory [24, 16] or the calculus of constructions [8]. A benefit of translating into an expressive language is that model transformations can be proved correct [29, 14]. In addition, a direct translation into an executable language, such as type theory, has the pragmatic value of making models immediately useful when programming.

One important property of the suggested translation, from diagrams to linear notation, is that the resulting semantics is compositional. That is, a small addition to the diagram cannot give rise to a large change in its meaning. For example, the notion of inheritance is difficult to understand compositionally, as adding an inheritance relation between two classes (a small addition) may create an inheritance cycle (a large change in meaning). This phenomenon is further discussed in Sect. 8, and the modelling language of Figure 9 uses generalisation instead of inheritance to preserve compositionality.

The translation from diagrams to type theory will first be applied to a simple modelling language (Figure 4) with only three notions, and then to a less simple language (Figure 9). Both of these modelling languages are self-describing (Sect. 2 and the Theorem). That is, there is a particular model of the language, that describes the whole language.

Turing's discovery [34] of the universal machine, capable of interpreting any program, was of paramount importance as it lead to the design of the stored program computer [11]. The dichotomy between code and data makes it plausible that analogues of Turing's universal machine in the space of data, i.e., self-describing modelling languages, are more important than currently appreciated. This is one reason for studying self-describing modelling languages: further motivation is given in Sect. 2.

	an element of M is	an element of $I(m)$ is
UML	a class diagram	an instance of m
MOF	a metamodel	a metamodel instance of m
DSD	a DSD schema	a document valid w.r.t. m
EBNF	an EBNF grammar	a string conforming to m
RDB	a database schema	a database instance of m
types	a type	an object of type m

Table 1 Examples of modelling languages of different kinds: syntax description languages, like EBNF [35], XML schema languages, like DSD [26], the language of relational databases (RDB) [7], and any type system, fit the definition of modelling language.

2 Self-describing modelling languages

A pair

$$\begin{cases} M & : & \text{set} \\ I & : & M \to \text{set}. \end{cases} \tag{1}$$

will be called a *modelling language*.[1] In a given modelling language (M, I), an element of M is called a *model*, and an element of $I(m)$, for a model m, is called an *instance* of m. An example of a modelling language is displayed in Figure 2. It has two models, and each model has two instances. There are many interesting examples of modelling languages according to this definition, not all of them with a corresponding visual notation. Some noteworthy examples are given in Table 1.

[1] Or, to be more precise, a *formal modelling language*. This structure is known elsewhere in the literature as *world* [19, 17] or *container* [22].

A *universal model* of a modelling language (M, I) is a model $u : M$ where the set $I(u)$ is isomorphic to M. The parts of the isomorphism will be named ρ (reflection) and π (reification), i.e., the diagram

$$I(u) \underset{\pi}{\overset{\rho}{\rightleftarrows}} M \tag{2}$$

commutes. In particular, $\pi(u) : I(u)$. A modelling language will be called *self-describing*, *metacircular*, or *reflexive*, if it has a universal model.[2]

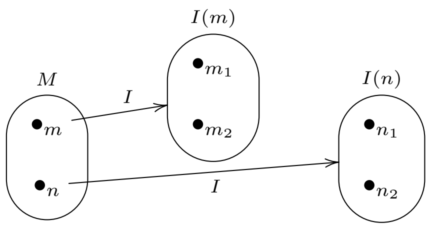

Figure 2 The leftmost oval shape represents the set of all models M in a modelling language (M, I).

For example, the DSD schema language for XML is self-describing in the sense that an XML document is a well-formed DSD schema if and only if it validates against the universal DSD schema [26, § 4]. Other schema languages for XML lack this feature.

Wirth succinctly describes the gist of EBNF's syntax by a universal EBNF grammar (Table 2). The only notions that remain to be explained are *character* and *identifier*. See Wirth's communication [35] for details. EBNF is probably the most concise self-describing language in current use.

There are at least three reasons why a modelling language should admit a (natural) universal model.

syntax	= { production }.
production	= identifier "=" expression ".".
expression	= term { "|" term}.
term	= factor { factor }.
factor	= identifier | literal | "(" expression ")" | "[" expression "]" | "{" expression "}".
literal	= """" character { character } """".

Table 2 The syntax of EBNF described by a EBNF grammar, verbatim after Wirth [35].

(1) The same query language can be used to query user models and metamodels alike. Relational database administrators have used this feature for decades to query the information schema [23]. Strictly speaking, only reification (π) is required for this to work. But at least a partial inverse ρ of π is needed if the results are to be useful.

(2) A modelling language that is not self-describing lacks, in a sense, expressivity, viz., the features necessary to describe itself. Moreover, a universal model exhibits a consistency among the notions used to explain the modelling language, and works as a kind of sanity check. The discussion about the notion of identifier in Sect. 7 exemplifies this form of sanity checking.

(3) The four layers of the OMG[3] pyramid [33] can be reduced to three, viz., the level of real-world entities (M0), the level of model instances (M1), and the level of models (M2). Given a modelling language (M, I), elements of M are M2 models and elements of $I(m)$, for an M2 model m, are M1 models. Clearly, a universal model $u : M$ resides in the M2 layer, despite being, as it were, a metamodel.

[2] To be precise, we should say that a model $u : M$ of a modelling language (M, I), is *universal with respect to an isomorphism* (ρ, π) *between* $I(u)$ *and* M. If the isomorphism is, as it were, unnatural, so is the universality of u.

[3] OMG (Object Management Group) is an international not-for-profit computer industry consortium and standards organization, responsible for, among other things, UML (Universal Modelling Language) and MOF (Meta-Object Facility).

3 A simple type system

Another example of a self-describing modelling language is the type system (D, T), that will be used in the definition of the simple modelling language (Sect. 5). It is defined by

$$\mathrm{D} = \{\mathrm{string}, \mathrm{money}, \mathrm{type}\}, \tag{3}$$

and

$$\begin{aligned} \mathrm{T}(\mathrm{string}) &= \{\textit{character strings}\} \\ \mathrm{T}(\mathrm{money}) &= \{\textit{monetary amounts}\} \\ \mathrm{T}(\mathrm{type}) &= \{\mathrm{string}, \mathrm{money}, \mathrm{type}\}. \end{aligned} \tag{4}$$

In particular, $\mathrm{T}(\mathrm{type}) = \mathrm{D}$, so 'type' is a universal model with ρ and π the identity function.

This rudimentary type system can be extended in several directions. For example, any number of basic types can be added, and the set D can be made closed under sum, product, and function space.

However, there are limitations on how the set of datatypes can be extended while maintaining the rule that type : T(type). It is for example known that the addition of the rule $\mathrm{U} : \mathrm{T}(\mathrm{U})$ to the rules for the type-theoretic universe U [24] leads to the paradox discovered by Girard [15].

4 From model diagrams to telescopes

Data modelling is first and foremost a *process*: relational modelling [7], entity-relationship modelling [6], object-role modelling [18], model driven engineering [30], etc. This process typically results in a set of diagrams. However, we are not trying to formalize the modelling process or the resulting diagrams, but the meanings underlying the diagrams. This is nontrivial, as, what a diagram *refers to*, *denotes*, or *means*, is elusive.

A first attempt is to say that a diagram refers to a *state of affairs*, so that, e.g., the symbol *Employee* of Figure 3 refers to a set of employees, etc. The problem with this explanation is that the diagram's state of affairs typically changes over time, so the diagram does not refer to any *particular* state of affairs: rather, the diagram signifies something general that various states of affairs fall under. That is, the entities of a diagram are *variable*, just as the relations of relational databases [10, pp. 17–18], [7, p. 4].

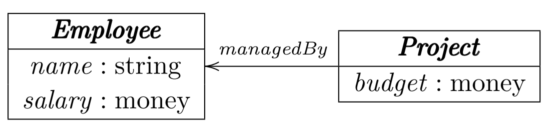

Figure 3 The model EP in the simple modelling language consists of two classes with attributes and a function between them.

The next observation is that, if there is to be any hope of systematically assigning meanings to diagrams, the meaning of a diagram somehow has to be composed of the meanings of its constituent parts. That is, the language behind the diagram has to adhere to the *principle of compositionality*, familiar from the philosophy of language [16, pp. 6–8]. Put differently, the meaning of a diagram should not change much due to a small change in the diagram.

To simplify the interpretation of diagrams, the following conventions will be adopted.

(1) A slanted font is used for uninterpreted symbols (e.g., *Employee*) and an upright font for interpreted symbols (e.g., string).

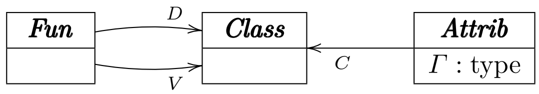

Figure 4 The universal model U of the simple modelling language: note that each construct of the modelling language (class, attribute, and function) is used by U.

(2) Interpreted symbols (e.g., money) may occur any number of times, whereas, if an uninterpreted symbol occurs more than once, it must be possible to disambiguate it.

(3) Uninterpreted symbols of a diagram range over certain categories of a formal language (e.g., *salary* ranges over money and *Project* ranges over the category of classes).

These conventions are best explained by taking Figure 3 as an example. Imagine a simple modelling language with only three notions: *class*, *attribute of class*, and *function between classes*.

In this language, Figure 3 is completely described by the following six assertions:

(1) *Project* is a class.
(2) *Employee* is a class.
(3) *budget* is an attribute of *Project* of type money.
(4) *name* is an attribute of *Employee* of type string.
(5) *salary* is an attribute of *Employee* of type money.
(6) *managedBy* is a function from *Project* to *Employee*.

The same assertions can be succinctly expressed using a yet to be defined formal language:

$$\begin{cases} \mathit{Project} : \text{class} \\ \mathit{Employee} : \text{class} \\ \mathit{budget} : \text{attrib}(\mathit{Project}, \text{money}) \\ \mathit{name} : \text{attrib}(\mathit{Employee}, \text{string}) \\ \mathit{salary} : \text{attrib}(\mathit{Employee}, \text{money}) \\ \mathit{managedBy} : \text{fun}(\mathit{Project}, \mathit{Employee}) \end{cases} \tag{5}$$

Such a sequence of assertions is similar to what a mathematician would write on the black board at the outset of an investigation: much like setting the stage for a play.

Now, we take a step back and recognize the above as a sequence of variable declarations. Thus, we have arrived at what de Bruijn [12] called a *telescope* and completed the informal path from model diagrams to telescopes. The reader is not required to be familiar with de Bruijn's telescopes, as the notion will only be used for purposes of comparison.

5 A simple modelling language

The simple modelling language is a fragment of UML's or MOF's class diagrams, with only three notions: *class*, *attribute*, and *function*. The benefit of treating such a limited language is that the semantics can be worked out in full detail without becoming too lengthy.

A *class* is the extension of a concept of the application domain;[4] and the first category of the simple modelling language is 'class'.

[4] This, and other explanations of UML concepts, serve only to guide the modelling process. They have no impact on the formal treatment. The use of the word *class* in logic originates with Peano who defines it as an "aggregation of entities" [28, p. x].

An *attribute* of a class is a characteristic applicable to every object in the extension of the class. Each attribute of a class is typed by a datatype drawn from the set D, called the *value type* of the attribute. For any given object of the class, the value of the attribute is of this type. The second category of the simple modelling language is $\mathrm{attrib}(A, \Gamma)$, where $A : \mathrm{class}$ and $\Gamma : \mathrm{D}$.

A *function* from one class to another is an assignment of exactly one object of the second class to each object of the first class. The third category of the simple modelling language is $f : \mathrm{fun}(A, B)$. The classes A and B will be called, respectively, the *domain* and *value* classes of the function f.

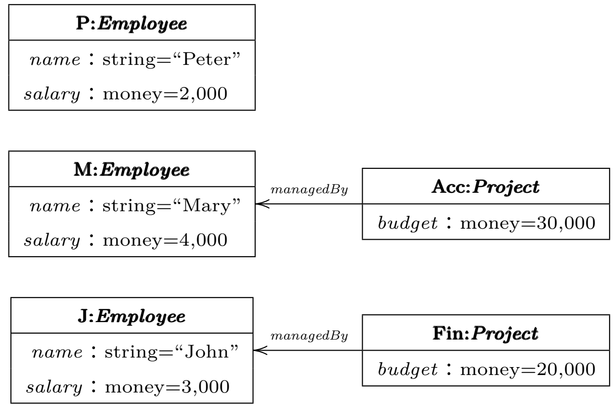

Figure 5 The instance ep of the model EP in the simple modelling language. The names of the instances are written before the class names, the values of the attributes are written after their declarations, and the value of a function at an instance is indicated by an arrow.

A *model* is a sequence of uninterpreted symbols (variables) declared to be of categories of the language, i.e., a telescope [12]. The categories of a model have to be well-formed in virtue of previously introduced uninterpreted symbols.[5] Thus, in general, a model has the form

$X_1 : \mathrm{class}, \ldots, X_m : \mathrm{class},$
$Y_1 : \mathrm{attrib}(X_{c_1}, \gamma_1), \ldots, Y_n : \mathrm{attrib}(X_{c_n}, \gamma_n),$
$Z_1 : \mathrm{fun}(X_{d_1}, X_{v_1}), \ldots, Z_p : \mathrm{fun}(X_{d_p}, X_{v_p}),$

where the symbols X_i are distinct, as are Y_i and Z_i; moreover, $1 \le c_i, d_i, v_i \le m$, and $\gamma_i : \mathrm{D}$. If needed, this can be encoded in type theory by

$$M = \sum_{(X,Y,Z)\,:\,\mathrm{enum}^3} \{c : X^Y, \gamma : \mathrm{D}^Y, d : X^Z, v : X^Z\}, \tag{6}$$

where 'enum' is the set of finite collections of names, the curly braces denote a standard record type, and X^Y means the same as $Y \to X$.

A *class diagram* (Figure 3) is the representation of a model as boxes and arrows according to the correspondence explained above. From this point onwards, the class diagram and the formal notation for the model will be considered interchangeable — as two expressions of the same thought.

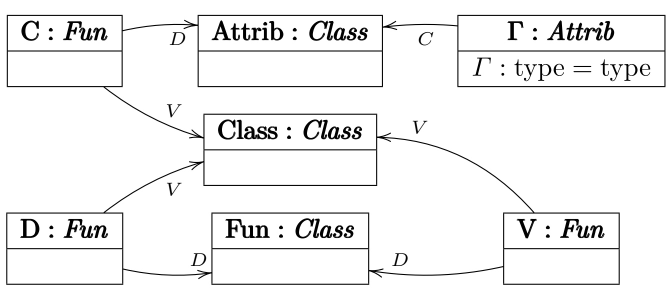

Figure 6 The instance u of the universal model U with the property that $\rho(\mathrm{u}) = \mathrm{U}$ and $\pi(\mathrm{U}) = \mathrm{u}$. Compare with Figure 4.

This formalisation of the notion of class diagram means, in

[5] So that, e.g., a class has to be introduced before its attributes, and the domain and value classes of a function have to be introduced before the function. Cf., the notion of *context* [16, 32].

particular, that it is easy to decide whether a given diagram is well-formed or not: simply write down the corresponding model and make sure it is well-formed.

An *instance* i of a model m is an interpretation of its uninterpreted symbols according to the following scheme:[6]

(1) a class symbol A : class is interpreted by a finite set A^i;

(2) an attribute symbol a : attrib(A, Γ) is interpreted by a function $a^i : A^i \to \mathrm{T}(\Gamma)$;

(3) and a function symbol f : fun(A, B) is interpreted by a function $f^i : A^i \to B^i$.

Note that there is at least one instance of any model, viz., the empty instance, in which all class symbols are interpreted by the empty set, and all attribute and function symbols by the "empty" function (from the empty set).

Instances can also be displayed as diagrams. For example, the instance ep (Figure 5) of the model EP (Figure 3) is defined as follows:

$\mathit{Employee}^{\mathrm{ep}} = \{\mathrm{P}, \mathrm{M}, \mathrm{J}\}$,
$\mathit{Project}^{\mathrm{ep}} = \{\mathrm{Acc}, \mathrm{Fin}\}$,
$\mathit{name}^{\mathrm{ep}} = \{\mathrm{P} \mapsto \text{“Peter”}, \mathrm{M} \mapsto \text{“Mary”}, \mathrm{J} \mapsto \text{“John”}\}$,
$\mathit{salary}^{\mathrm{ep}} = \{\mathrm{P} \mapsto 2,000, \mathrm{M} \mapsto 4,000, \mathrm{J} \mapsto 3,000\}$,
$\mathit{budget}^{\mathrm{ep}} = \{\mathrm{Acc} \mapsto 30,000, \mathrm{Fin} \mapsto 20,000\}$,
$\mathit{managedBy}^{\mathrm{ep}} = \{\mathrm{Acc} \mapsto \mathrm{M}, \mathrm{Fin} \mapsto \mathrm{J}\}$.

Encoded in type theory, the set of instances of a given model is defined by

$$I((X,Y,Z),\{c,\gamma,d,v\}) = \sum_{|\cdot|:\, X \to \mathrm{enum}} \left(\prod_{y:\, Y} |c(y)| \to \gamma(y) \right) \times \left(\prod_{z:\, Z} |d(z)| \to |v(z)| \right). \quad (7)$$

Recall that Σ and Π stand for disjoint union and Cartesian product of indexed families of sets.

6 A universal model for the simple modelling language

A universal model, written U, of the simple modelling language is presented in Figure 4. It corresponds to the following sequence of assertions:

Class : class,
Attrib : class,
Fun : class,
Γ : attrib$(\mathit{Attrib}, \mathrm{type})$,
C : fun$(\mathit{Attrib}, \mathit{Class})$,
D : fun$(\mathit{Fun}, \mathit{Class})$,
V : fun$(\mathit{Fun}, \mathit{Class})$.

▶ Theorem. The simple modelling language described in Sect. 5 is self-describing.

Proof. We must show that U is a universal model, i.e., we must define ρ and π and show that they are inverse of each other. Let s be a instance of the model U. Assume that

$\mathit{Class}^s = \{A_1, \ldots, A_m\}$,
$\mathit{Attrib}^s = \{a_1, \ldots, a_n\}$,
$\mathit{Fun}^s = \{f_1, \ldots, f_p\}$,
$\Gamma^s : \mathit{Attrib}^s \to \mathrm{T}(\mathrm{type})$,
$C^s : \mathit{Attrib}^s \to \mathit{Class}^s$,

[6] Using the terminology of logic, a model is an uninterpreted language and an instance is an interpretation of its uninterpreted symbols. Cf. [31] and [2].

$D^s : \mathit{Fun}^s \to \mathit{Class}^s$,
$V^s : \mathit{Fun}^s \to \mathit{Class}^s$.

Recall that a model is a sequence of uninterpreted symbols declared to be of certain categories. The model $\rho(s)$ is defined as follows:

A_1 : class, ..., A_m : class,
$a_1 : \mathrm{attrib}(C^s(a_1), \Gamma^s(a_1))$, ..., $a_n : \mathrm{attrib}(C^s(a_n), \Gamma^s(a_n))$,
$f_1 : \mathrm{fun}(D^s(f_1), V^s(f_1))$, ..., $f_p : \mathrm{fun}(D^s(f_p), V^s(f_p))$.

This model is always well-formed in the sense described above, i.e., symbols are unique within each form of category (class, attrib, and fun).

Conversely, let S be a model of the simple modelling language, given by

B_1 : class, ..., B_m : class,
$b_1 : \mathrm{attrib}(B_{c_1}, \gamma_1)$, ..., $b_n : \mathrm{attrib}(B_{c_n}, \gamma_n)$,
$g_1 : \mathrm{fun}(B_{d_1}, B_{v_1})$, ..., $g_p : \mathrm{fun}(B_{d_p}, B_{v_p})$,

where $\gamma_1, \ldots, \gamma_n$ are elements of the set $\mathrm{D} = \mathrm{T}(\mathrm{type})$, and each of the numbers $c_1, \ldots, c_n$, $d_1, \ldots, d_p$, and $v_1, \ldots, v_p$ are in the range $1, \ldots, m$. Then $\pi(S)$ is an instance of U given by

$\mathit{Class}^{\pi(S)} = \{B_1, \ldots, B_m\}$,
$\mathit{Attrib}^{\pi(S)} = \{b_1, \ldots, b_n\}$,
$\mathit{Fun}^{\pi(S)} = \{g_1, \ldots, g_p\}$,
$\Gamma^{\pi(S)}(b_x) = \gamma_x \;: \mathrm{T}(\mathrm{type})$,
$C^{\pi(S)}(b_x) = B_{c_x} \;: \mathit{Class}^{\pi(S)}$,
$D^{\pi(S)}(g_y) = B_{d_y} \;: \mathit{Class}^{\pi(S)}$,
$V^{\pi(S)}(g_y) = B_{v_y} \;: \mathit{Class}^{\pi(S)}$.

To show that $\pi(\rho(s)) = s$, let s and S be defined as above, and consider $\pi(\rho(s))$, where $S = \rho(s)$. Comparing the definition of S with the definition of $\rho(s)$, we get $A_i = B_i$ (as symbols), $a_i = b_i$, $f_i = g_i$, $C^s(a_x) = B_{c_x}$, $\Gamma^s(a_x) = \gamma_x$, $D^s(f_y) = B_{d_y}$, and $V^s(f_y) = B_{v_y}$. The result follows from a comparison with the definition of $\pi(S)$.

To show that π is also a right inverse of ρ, let S be given as above and plug $\pi(S)$ into the definition of ρ. The result is S. ◀

An obvious use of this Theorem is to apply the function π to the model U. The resulting instance, Figure 6, should be studied carefully. It is also instructive to compare it with Table 2.

Figure 7 shows the reification of the diagram of Figure 3.

7 A less simple modelling language

This Section is deliberately brief, and many details are left to the reader. It is best viewed as an extended example of how to apply the techniques introduced earlier in the paper. The example is based on Figure 9, showing the universal model of a significant fragment of the class diagrams of xUML [25].[7] The main differences between this less simple language and the previously introduced simple language are outlined below.

First, there is one more datatype, viz., 'mult', of multiplicities, i.e.,

$$\begin{aligned} \mathrm{D} &= \{\mathrm{string}, \mathrm{money}, \mathrm{type}, \mathrm{mult}\}, \\ \mathrm{T}(\mathrm{mult}) &= \{a..b \mid a : \mathrm{N}, b : \mathrm{N} \cup \{\star\}, a \le b\}, \end{aligned} \tag{8}$$

[7] xUML is a fragment of UML that is designed to facilitate the *execution* of models.

Model	Instance i
A : class	A^i : set
a : attrib(A, Γ)	$a^i : A^i \to \mathrm{T}(\Gamma)$
R : assoc(A, B)	$R^i : A^i \times B^i \to$ prop
r : rrole(A, B, R, o)	$r^i_1(x) : o$, $r^i_2(x) : r^i_1(x) \hookrightarrow B^i$, $r^i_3(x)(y) : R^i(x, y) \leftrightarrow (\exists z : r^i_1(x)) r^i_2(x)(z) = y$
l : lrole(A, B, R, λ)	$l^i_1(y) : \lambda$, $l^i_2(y) : l_1(y) \hookrightarrow A^i$, $l^i_3(y)(x) : R^i(x, y) \leftrightarrow (\exists z : l^i_1(y)) l^i_2(y)(z) = x$
e : ident(A, a, Γ)	$e^i : \mathrm{T}(\Gamma) \to A^i + \{\star\}$, s.t. $e^i(x) = \mathrm{left}(y)$ iff $a^i(y) = x$
g : gen$(A, S_1, \ldots, S_n)$	$g^i : A^i \cong S^i_1 \times \cdots \times S^i_n$
s : assclass(C, A, B, R)	$s^i : C^i \cong (\Sigma\ (x, y) : A^i \times B^i) R^i(x, y)$

Table 3 The forms of assertion of the less simple modelling language, together with their interpretations in an instance.

where $a..b$ is the set $\{a, a+1, \ldots, b\}$ if b is finite, and $a..\star$ stands for $\{a, a+1, \ldots\}$. The datatypes 'string' and 'money' are as before, and 'type' is still universal.

Table 3 lists the forms of assertions used when translating a less simple diagram to linear notation, together with their interpretations in an instance. Classes and attributes work exactly as for the simple modelling language.

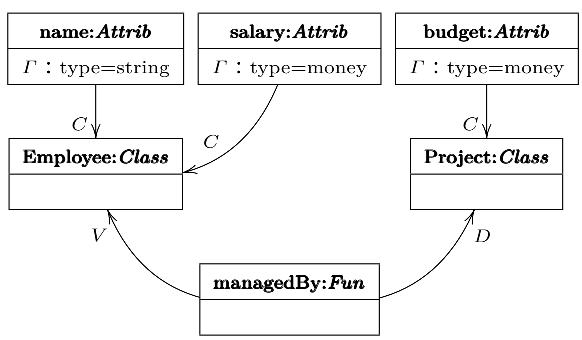

Figure 7 The instance ep of the universal model U with the property that $\rho(\mathrm{ep}) = \mathrm{EP}$ and $\pi(\mathrm{EP}) = \mathrm{ep}$. Compare with Figure 3.

Instead of functions, the less simple modelling language uses associations, which may have two kinds of roles: left and right. An association R : assoc(A, B) is interpreted in type theory by a binary relation R^i on A^i and B^i. A right role r : rrole(A, B, R, o), where o is a multiplicity, is interpreted as a triple valued function $r^i(x) = (r^i_1(x), r^i_2(x), r^i_3(x))$, where $x : A^i$. The first component $r^i_1(x) : o$ gives the multiplicity of x; the second component $r^i_2(x) : r^i_1(x) \hookrightarrow B^i$ is an injection of the multiplicity into B^i;[8] the third component is a proof that an element y of B^i is related to x by R^i if and only if y is in the image of $r^i_2(x)$. Another way to put it is that $r^i(x)$ identifies the subset of B^i, with a finite cardinality drawn from the set o, that is related by R^i to $x : A^i$. Left roles are treated analogously to right roles.

As a special case, when the multiplicity is $o = 1..1$, a right role induces a normal function $A^i \to B^i$. The virtue of this treatment of roles is that it is compositional, i.e., a left or right role can be added to a diagram without changing the interpretation of the original diagram. In fact, formally, nothing prevents an association from having several left or right roles.

Identifiers in xUML serve the same purpose as unique keys in relational databases, i.e., they make it possible to retrieve an instance (row or tuple in database parlance) from the value of an attribute. For example, if there were an identifier of the *name* attribute of the *Employee* class of Figure 3, names would have to be unique, and it would be possible to retrieve the instance corresponding to a name, if any.

[8] Here the number $r^i_1(x)$ is identified with the set on $r^i_1(x)$ elements.

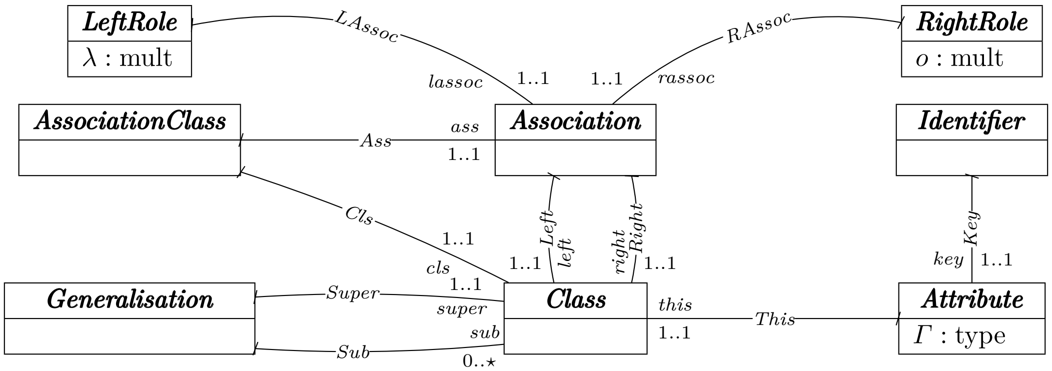

Figure 9 The universal model of a fragment of the modelling language xUML, capable of expressing the notions class, attribute, association, generalisation, association class, identifier, and left and right role.

An identifier $e : \mathrm{ident}(A, a, \Gamma)$ of an attribute a indicates that the values of the attribute are different for different instances of the class A.[9] The identifier e is interpreted in an instance i as a function e^i from $\mathrm{T}(\Gamma)$ to the set $A^i + \{\star\}$, such that e^i is a partial inverse of a^i, i.e., for all $x : A^i$ and $y : \mathrm{T}(\Gamma)$, $e^i(x) = \mathrm{inl}(y)$ if and only if $a^i(y) = x$. Here 'inl' denotes the canonical injection $A^i \hookrightarrow A^i + \{\star\}$.

A generalisation $g : \mathrm{gen}(A, S_1, \ldots, S_n)$ is interpreted in an instance i as an isomorphism between the interpretation of the superclass A^i and the interpretations of its subclasses $S_1^i \times \cdots \times S_n^i$.

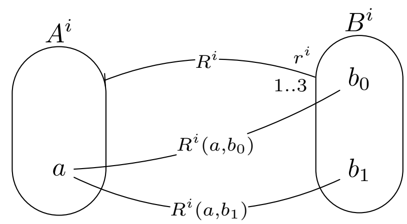

Figure 8 The interpretation a right role $r : \mathrm{rrole}(A, B, R, 1..3)$ in an instance i, where A^i has an element a related to exactly two elements b_0 and b_1 of B^i. In particular, $r_1^i(a) = 2$, and $r_2^i(a) : \{0, 1\} \hookrightarrow B$, with $r_2^i(a)(j) = b_j$.

An association class $s : \mathrm{assclass}(C, A, B, R)$ between a class C and an association R is interpreted as an isomorphism between C^i and the set of pairs (x, y) in $A^i \times B^i$ that are related by R^i.

8 Related work

There are several approaches to the semantics of UML and MOF class diagrams, e.g., logic based [3], graph based [33], coinductive [29], or, like this paper, algebraic [5, 13].

Our modelling languages depart from the MOF in two important respects. We consider generalisation instead of inheritance; and, as opposed to UML and MOF, we have no common genus of datatypes and classes.

Generalisation and inheritance are sometimes taken as synonymous, but I think there is

[9] This paper makes a significant departure from xUML identifiers (and database uniqueness constraints) by only allowing one attribute to participate in an identifier; a faithful encoding would require the multiplicity of the role *key* of Figure 9 to be one to many. However, if the multiplicity was simply changed, the model of Figure 9 would no longer be universal, as instances would include identifiers combining several attributes of different classes. Thus, the modelling language would have to be significantly strengthened to cater for identifiers with higher multiplicity.

an important distinction to be made. By inheritance, I mean the relation B *inherits from* A, that would be interpreted by $B^i \subset A^i$ in an extensional framework. This is difficult to formalize in type theory as there is no subset relation. However, the relation B *is generalised by* A can be interpreted by an injection $B^i \hookrightarrow A^i$.

As regards the existence of a common genus of datatypes and classes, it is interesting to review what Date [9, p. 865] calls *the great blunder*. There are three notions involved: the notion of *datatype*, i.e., our D or what Date calls *domain*; the notion of relational variable (*relvar* in relational database theory); and the notion of *class*. Date's main point is that *datatype* $\neq$ *relvar*, and this distinction is maintained in this paper. In fact, our notion of class is similar to the notion of *relvar* — to begin with, both are variables.

However, what Date actually calls *the great blunder* is the identification *relvar* = *class* (made here): that is, he considers the identification *datatype* = *class* correct. Date's identification is based on the conception of a class as a record type.

In this paper, the notion of class is identified with the notion of relvar (rather than with the notion of datatype) because object-oriented programming is based on the idea that a program can create a *new* instance of a class. The classes of this paper support the *new* operation, and relvars support the *insert* operation: in both cases, one element is added to the set interpreting the variable. Datatypes, on the other hand, are more like mathematical sets, and, e.g., the idea of creating a *new* number is repugnant. To conclude, this paper makes *the great blunder* in words, but not in spirit.

My approach to the translation of model diagrams into type theory differs from that of Poernomo et al. [29, 14] in one important respect: type-theoretic concerns have influenced my design of the modelling languages, while Poernomo et al. have taken the MOF at face value. Encoding the full MOF requires coinductive datatypes and definitions by corecursion, which soon lead to rather complex formalisations. In addition, the semantics becomes noncompositional, due to the outermost fixpoint operator in the definition of models. I have avoided these problems by simplifying the modelling language.

An analog to the notion of class diagram, with respect to how its semantics has evolved from a mere "blackboard" semantics, is the notion of state chart, as expounded by Harel [20]. A precise constructive semantics for a species of state charts is given by André [1].

9 Conclusion and future work

In my opinion, one of the main obstacles to model driven approaches gaining wide acceptance in the industry is insufficient tool support. One step in the right direction would be to formalize the simple (or less simple) modelling language inside a proof assistant like Coq [4] or Agda [27].

In addition to allowing formal manipulation of models, such a tool could make it possible to generate a diagram from a possibly annotated model instance, thus reinforcing the point that diagrams are valid formal expressions and, with time, changing a view held by many software engineers, viz., that diagrams are inherently vague [31].

The reader may have noticed that the two modelling languages presented in this paper, although using the notation of UML class diagrams, are semantically more akin to the entity-relationship model [6] or ORM [18]. It would be interesting to find out what characterises features of data modelling and object-oriented programming that can be interpreted using the direct approach of Sect. 4.

A difference between Figure 4 and Figure 9 is that, in the forme, all features of the modelling language are used to define the universal model, whereas, in the latter, the four notions *class*, *attribute*, *association*, *right role* would suffice. That is, the notions *generalisation*, *association class*, *left role*, and *identifier* are like appendices to a smaller modelling language. Does a modelling language with an irreducible universal model have any advantage over a modelling language with redundant features?

One potential direct application of the simple modelling language is as a data model for a non-relational database management system, using the identification *database schema = model*. Several database maintenance operations could be simplified by using the universal model U. For example, to define a new database schema one would simply have to define an instance of U. This definition would use the same syntax as the definition of an instance of any other model.

In this context, it would also be interesting to consider how data manipulation operations interact with ρ and π. For example, creating a new instance of the class *Class* in an instance of U could create a new class in the corresponding model.

Acknowledgements

This work was partially supported by Engineering and Physical Sciences Research Council (EPSRC) grant number EP/G03012X/1.

Thanks to I. Poernomo for teaching me about metamodelling and the MOF. Thanks also to P. Martin-Löf, E. Palmgren, O. Wilander, and other participants of the Stockholm-Uppsala logic seminar for valuable comments on an early version of this paper.

Moreover, I thank D. Calvanese, I. Feinerer, T. Halpin, D. Harel, G. Karsai, S. Mellor, B. Rumpe, and the anonymous reviewers for valuable corrections, amendments, and comments to a draft version. Finally, thanks to the anonymous reviewers for valuable corrections and comments.

References

1 C. André. Computing SyncCharts reactions. *Electronic Notes in Theoretical Computer Science*, 88:3–19, 2004.
2 C. Atkinson and T. Kühne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.
3 D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1–2):70–118, 2005.
4 Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
5 A. Boronat and J. Meseguer. An algebraic semantics for MOF. In J.L. Fiadeiro and P. Inverardi, editors, *FASE 2008*, volume 4961 of *LNCS*, pages 377–391. Springer, 2008.
6 P. P.-S. Chen. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
7 E. F. Codd. *The Relational Model for Database Management*. Addison-Wesley, 1990.
8 T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.
9 C. J. Date. *An Introduction to Database Systems*. O'Reilly, 7 edition, 2000.
10 C. J. Date. *Database in Depth*. O'Reilly, 2005.
11 M. Davis. *Engines of Logic: Mathematicians and the Origin of the Computer*. W. W. Norton & Company, NY, 2000.
12 N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Inform. Comput.*, 91(2):189–204, 1991.

13 I. Feinerer and G. Salzer. Consistency and minimality of UML class specifications with multiplicities and uniqueness constraints. In *TASE 2007*, pages 411–420. IEEE Computer Society Press, 2007.

14 C. Fiorentini, A. Momigliano, M. Ornaghi, and I. Poernomo. A constructive approach to testing model transformations. In L. Tratt and M. Gogolla, editors, *ICMT 2010*, volume 6142 of *LNCS*, pages 77–92. Springer, 2010.

15 J. Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

16 J. G. Granström. *Treatise on Intuitionistic Type Theory.* Logic, Epistemology, and the Unity of Science. Springer, 2011.

17 J. G. Granström. A new paradigm for component-based development. *Journal of Software*, 7(5):1136–1148, 2012.

18 T. A. Halpin. Object-role modeling: principles and benefits. *IJISMD*, 1(1):33–57, 2010.

19 P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. G. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, volume 1862 of *LNCS*, pages 317–331, 2000.

20 D. Harel. Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

21 D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.

22 P. Hoogendijk and O. de Moor. Container types categorically. *J. Funct. Program.*, 10(2):191–225, 2000.

23 ISO/IEC 9075-11:2008. Information and definition schemas (SQL/schemata). Technical report, ISO, Geneva, Switzerland, 2008.

24 P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory. Bibliopolis, Napoli, 1984.

25 S. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture.* Addison Wesley, 2002.

26 A. Møller. Document structure description. Technical report, BRICS, 2005.

27 U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2009.

28 G. Peano. *Arithmetices Principia Nova Methodo Exposita.* Fratelli Bocca, Turin, 1889.

29 I. Poernomo. The meta-object facility typed. In *SAC*, pages 1845–1849, 2006.

30 D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

31 E. Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, 2003.

32 A. Tasistro. *Formulation of Martin-Löf's type theory with explicit substitutions.* Licentiate thesis, Chalmers University of Technology, 1993.

33 X. Thirioux, B. Combemale, X. Crégut, and P. L. Garoche. A framework to formalise the MDE foundations. In R. Paige and J. Bézivin, editors, *International Workshop on Towers of Models*, pages 14–30, 2007.

34 A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

35 N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.