A Graph-Based Semantics for UML Class and Object Diagrams

Anneke Kleppe* and Arend Rensink

a.kleppe@utwente.nl, a.rensink@utwente.nl
Department of Computer Science
University of Twente, Enschede, The Netherlands

Abstract. In this paper we propose a formal extension of type graphs with notions that are commonplace in the UML and have long proven their worth in that context: namely, inheritance, multiplicity, containment and the like. We believe the absence of a comprehensive and commonly agreed upon formalisation of these notions to be an important and, unfortunately, often ignored omission. Since our eventual aim (shared by many researchers) is to give unambiguous, formal semantics to the UML using the theory of graphs and graph transformation, in this report we propose a set of definitions to repair this omission. With respect to previous work in this direction, our aim is to arrive at more comprehensive and at the same time simpler definitions.

1 Introduction

Software industry is showing an increasing interest in model-driven development. Indeed, we have little doubt that the future lies in higher-level models to take the place of code, in all but the most performance critical domains. With this trend, however, the quality of those models is of increasing importance. By this we do not mean the quality of the product being modelled (which obviously is the final consideration) but rather of the modelling paradigm. Good models may not guarantee good software, but on the other hand, a bad (ambiguous, inconsistent or unclear) model can never be expected to yield a good end product, in particular if the transformation from model to software is largely automatic.

The quality of models is determined by many aspects, among we believe precision, consistency and completeness to be paramount. The precision of a model corresponds to the lack of ambiguity, or in other words, the degree to which the model will be understood in exactly the same way by different persons and tools during the software development process. Consistency formally means the existence of an (i.e., at least one) instance, or implementation, of the model, whereas completeness means the inclusion of all relevant aspects, or (in other words) the ability to predict the behaviour of the system under all circumstances.

The above “quality criteria” have a clear, universally agreed-upon interpretation in the world of mathematics. To make the benefits of the mathematical interpretation available for everyday use in the world of software modelling, however, it is imperative that

* The research in this paper was carried out in the GRASLAND project, funded by the Dutch NWO (project number 612.063.408).
there be a translation from the latter to the former; in other words, a formal semantics of the modelling language. For instance, it is commonly agreed that a natural interpretation of (UML-type) diagrams is in terms of graphs — essentially, just nodes with connecting edges. Indeed, many authors use UML class (and object) diagrams claiming that they are representations of type graphs. Unfortunately, few provide an actual formal underpinning of this claim, or when they do, the semantics covers only a relatively small part of UML; for instance, [1, 16]. The most comprehensive is [17], but even there such basic notions as multiplicities are missing.

We see the absence of a more complete semantics as an important and regrettable omission, but there is little doubt in our minds about the reason for it. Namely, from a purely formal standpoint, there is no challenge at all in providing the necessary definitions; in particular, they do not give rise to new results, in the mathematical sense of proved theorems. Instead, getting the semantics right is an engineering effort, where choices about how to formalise various UML concepts are not dictated by rock-hard mathematical arguments, but rather by a balance between simplicity and expressiveness, between understandability and conciseness. Carrying out this effort requires a knowledge and facility with the required mathematical concepts that is outside the scope of most software engineers who would otherwise, by strength of their knowledge of the modelling discipline, be ideally positioned for this task.

Like the papers cited above, in this paper we distinguish the type and instance levels, or in other words, type graphs and instance graphs. We see a type graph as an intensional definition of a set of instance graphs, namely, those instance graphs for which it is a correct type. Type graphs are then enriched with constraints that capture UML concepts such as bi-directional associations, multiplicities, collection types, inheritance, redefinition of associations, and composition relationships. In this, we have based ourselves on the (verbal) descriptions in the UML 2.0 specification [12].

In searching for the aforementioned balance between simplicity and expressiveness of the semantics, we have used the following guidelines:

– Instance graphs should be as simple and straightforward as we can make them, if necessary at the price of increasing their sizes. In other words, where there is a choice between enriching the formalism (resulting in more concise but more complex graphs) or using larger (sub-)graphs to encode complexity, we have tended to choose in favour of the latter.

This consideration has led to the choice of simple, edge-labelled binary graphs, with as only non-standard feature that edges are numbered, and the numbers are (sometimes) used to reflect an ordering.

– Type graphs should be as close to instance graphs as we can make them; the number of special features or decorations should be minimised.

We have achieved this by using the concept of a graph constraint, which is essentially a template for a logical formula on top of an ordinary (type) graph. The number of such templates is small but extensible. Apart from this, the only notion that we have found to be indispensable is that of inheritance: this cannot be formulated as a graph constraint, but is captured instead by an explicit partial ordering over the type graph nodes.
The remainder of this paper is structured as follows: after providing the basic definitions to set the stage in Section 2, we discuss the graph constraints in Section 3. We consider these to be the heart of our contribution. In Section 4 we relate our constraints to the standardised UML concepts. Finally, in the conclusion (Section 5) we come back to the above considerations and re-evaluate our choices.

2 Basic concepts

UML class diagrams are, in the terminology of this paper, visualisations of models in a graphical format; likewise, object diagrams are visualisations of model instances. Before we can define these concepts formally, we need to introduce a number of auxiliary notions.

2.1 Names and namespaces

UML is a visual language; its “sentences” are diagrams. However, a major part of any diagram is still text, and so we need conventions for visualising text inside diagrams. Here we are especially interested in text used to name things; however, let us first recall the usual (often unspoken) agreements on the visualisation of text.

Formally speaking, any piece of text is a string of letters from some alphabet. For the purpose of this paper we fix an alphabet $\text{Alpha}$ not containing white space characters; then we can visualise any non-empty string in $\text{Alpha}^+$ unambiguously. (Note that we cannot visualise the empty string. Of course we can put quote symbols around it, but then the visualised text is not the actual string but a transformation of it to a non-empty string — and the transformation needs to be defined with care to make the visualisation unambiguous.)

Within the set of non-empty strings $\text{Alpha}^+$, we furthermore assume a fixed set of names $\text{Name}$. For most purposes it is sufficient to think of names as words conforming to the usual scheme of a letter followed by any number of letters, digits and (maybe) underscores and other special symbols; however, also the usual operation and constant symbols used in arithmetic will be names in our sense. In general, one may assume $\text{Name}$ to be defined by some regular expression over $\text{Alpha}$.

However, for a practically usable naming scheme this is not yet enough. In many cases it has been shown to be useful to identify things only partially, leaving it up to the context in which the identification takes place to disambiguate the name. In order to take this into account, we rely on the (well-known) concept of namespace.

**Definition 1 (namespace).** Let $\text{Name}$ be some globally fixed set of names. The set $\text{ID}$ of identifiers, and the set $\text{NS}$ of namespaces are the smallest sets such that

\[
\text{ID} = \text{NS} \times \text{Name}
\]

\[
\text{NS} = \{\top\} \cup \text{ID}
\]

Less cryptically, $\text{NS}$ and $\text{ID}$ are built up as follows:

- An **identifier** is a pair $\langle \text{ns}, \text{name} \rangle$ of a namespace $\text{ns}$ and a name $\text{name}$;
There is a root or top namespace $\top \in NS$;

For every $ns \in NS$ and $name \in Name$, the identifier $\langle ns, name \rangle$ is again a namespace.

For a given identifier $id (\in ID)$, we use $ns(id) (\in NS)$ to denote the name space and $name(id) (\in Name)$ to denote the name.

To visualise an identifier we use a well-known notation, which is less cumbersome than the angular brackets: the name space and name are separated by a dot, and the top namespace is omitted altogether. Thus, $\langle ns, name \rangle$ is actually written $ns.name$. At the same time we assume that this dot (which is in Alpha) is not used any element of Name, so as to avoid ambiguity in the notation.

For instance, the identifier $a.name.space$ consists of the name space $a$ in the namespace $a.name$, which itself is an identifier with namespace $a$ and name $name$. The identifier $a$, finally, consists of the name $a$ in the top name space.

In this setup, we can identify things unambiguously by just giving the name, provided that the intended name space is clear, either implicitly or explicitly. This makes for a more open-ended and flexible naming scheme than relying on globally unique names.

### 2.2 Signatures and algebras

For our definition of model we use the notion of attributed type graph, as defined in [3]. We repeat the definition here; for a discussion see op. cit. We also need the formal concept of an algebra, which we likewise repeat here, for the sake of completeness and to introduce the notations.

**Definition 2.** A data signature is a tuple $\text{Sig} = \langle \text{Sort}, \text{Oper}, \text{type} \rangle$, where

- $\text{Sort}$ is a set of data sorts;
- $\text{Oper}$ a set of operation symbols;
- $\text{type}: \text{Oper} \rightarrow \text{Sort}^+$ is a mapping that associates to every operation $op \in \text{Oper}$ a non-empty string of sorts $\text{type}(op) = s_0 \cdots s_n$ for $n \geq 0$, of which the elements $s_0 \cdots s_{n-1}$ constitute the parameter sorts of the operation, and $s_n$ constitutes the result sort.

A $\text{Sig}$-algebra is a tuple $\text{Alg} = \langle \text{Data}, \text{Func} \rangle$, where

- $\text{Data}$ is a set of disjoint carrier sets, one per sort $s \in \text{Sort}$, denoted $D_s$;
- $\text{Func}$ is a set of functions, one per operation $op \in \text{Oper}$, denoted $f_{op}$;
- The type of the functions in $\text{Func}$ should be consistent with $\text{type}$, in the sense that $f_{op} : D_{s_0} \times \cdots \times D_{s_{n-1}} \rightarrow D_{s_n}$ whenever $\text{type}(op) = s_0 \cdots s_n$.

In practice it is typically the case that $\text{Sort}, \text{Oper} \subseteq ID$ and $\text{Sort} \cap \text{Oper} = \emptyset$. This guarantees that we can refer to the sorts and operations unambiguously by identifier.

For this paper we make the same assumptions.

Note that an operation $op$ with no parameters (i.e., such that $|\text{type}(op)| = 1$) represents a constant value, which in a $\text{Sig}$-algebra is given by $f_{op}()$. 
2.3 Graphs

One of the core concepts of this paper is that of graphs. We start by repeating the usual definition of a directed, multi-sorted graph.

**Definition 3 (graph).**

- A graph is a tuple $G = \langle \text{Node}, \text{Edge}, \text{src}, \text{tgt} \rangle$ where Node is a set of nodes, Edge a set of (directed) edges, and src, tgt: Edge $\rightarrow$ Node are source and target functions, respectively.
- A path through a graph $G$ is a sequence $e_1 \cdots e_n \in \text{Edge}^+$, such that consecutive edges in the path are connected in the graph, i.e., $\text{tgt}(e_i) = \text{src}(e_{i+1})$ for $1 \leq i < n$.
- A set of edges $E \in \text{Edge}$ is cycle-free if there is no path $e_1 \cdots e_n$ in $G$, such that $e_i \in E$ for all $1 \leq i \leq n$, and $\text{tgt}(e_n) = \text{src}(e_1)$. $E$ is a forest if it is cycle-free and for all $e \in E$, there is at most one $e' \in E$ such that $\text{tgt}(e') = \text{src}(e)$.

Note that although this definition does not yet specify node or edge labels, the nodes and edges do have identities. In some circumstances it will be the case that $\text{Node}, \text{Edge} \subseteq \text{ID}$ and the identities are actually meaningful to the reader; it then makes sense to include them in a visualisation of the graph. In particular, this is the case for type graphs — see below.

We will use two kinds of graph: instance graphs and type graphs. Both extend the notion of graph with some further structure. To start with instance graphs: these have an additional labelling function that associates an identifier with every node and edge. Furthermore, edges have indices, which are chosen from the set of natural numbers in such a way that the combination of source node, index and label together completely determine the edge.

**Definition 4 (labelled graph).** A labelled graph is a tuple $IG = \langle \text{Node}, \text{Edge}, \text{src}, \text{tgt}, \text{ix}, \text{lab} \rangle$ where $\langle \text{Node}, \text{Edge}, \text{src}, \text{tgt} \rangle$ is a graph and

- $\text{ix}: \text{Edge} \rightarrow \mathbb{N}$ is an indexing function assigning a natural number to every edge;
- $\text{lab}: (\text{Node} \cup \text{Edge}) \rightarrow \text{ID}$ is a labelling of nodes and edges;
- For $e_1, e_2 \in \text{Edge}$, if $\text{src}(e_1) = \text{src}(e_2)$, $\text{ix}(e_1) = \text{ix}(e_2)$ and $\text{lab}(e_1) = \text{lab}(e_2)$, then $e_1 = e_2$.

For a given node $n \in \text{Node}$ and label $a \in \text{ID}$, the set of outgoing edges is defined by

$$\text{out}(n, a) = \{ e \in \text{Edge} \mid \text{src}(e) = n, \text{lab}(e) = a \}.$$  

The indices assigned by the function $\text{ix}$ are used for two purposes:

- To distinguish edges. Graphs may have distinct edges going out of the same node and bearing the same label, and even going to the same node (sometimes called parallel edges). These are useful to represent some UML concepts; in particular, ordered associations and bags. The indices serve to distinguish such edges, i.e., give them their own identity. In fact, as can be seen from the definition, the identity of an edge is completely determined by its source node, index, and label.
To order edges. One of the more powerful UML concepts is that of an ordered association; this does not only define a one-to-many relation between objects of one type to objects of another, but also establishes a local ordering over the set of (target) objects related to a single (source) object. (The ordering is local in the sense that it is defined with respect to the source object; another source object, or another ordered association, may contain the same target objects in a different order.)

In contrast to edges, the encoding of node identities is not fixed by the above definition. It should, however, be understood that there is indeed some distinguishing mechanism, apart from the labelling function, that tells nodes apart. On the implementation level, for instance, this mechanism is typically based on memory addresses, or, for nodes in Node ∩ Data, by the data value. On the modelling level, the position within a diagram in principle suffices as the distinguishing mechanism. On the other hand, for ease of reference it is very common to use symbolic names for nodes. Thus, we arrive at a graphical representation of labelled graphs based on the following conventions:

- Nodes are drawn as boxes with inscribed labels. The labels are preceded by a colon (‘:’). In front of a colon, there may either be a symbolic name, which is in fact itself an element of Name, but which plays no role in the formal meaning of the graph and in fact has no counterpart in Definition 4; or, in the case of nodes that are actually data values, the string representation of the data value may be displayed. (We will see below that the label is typically the type, which for data values \( v \in Data \) is given implicitly by \( \text{type}(v) \).)

If a node has either a symbolic name or a data value indicating its identity, then there is no objection against having this node occur multiple times in the graphical representation: these multiple occurrences still represent the same node, without ambiguity.

- Edges are drawn as arrows with superimposed labels. The labels may be preceded by a number representing the edge index, separated from the label by a colon; in particular, this is necessary if there is more than one outgoing edge with that label and the numbering is needed to determine an ordering.

- As an important special case, edges pointing to nodes that are explicitly identified, either by data values or by symbolic names, may be represented by inscribed equations of the form “\( \text{label} = \text{id} \)” or “\( \text{label}:\text{Type} = \text{id} \)” instead of arrows. Here, label is the edge label, Type the label of the target node, and id the target node identifier; i.e., either the string representation of some data value or a symbolic name defined by some node label elsewhere in the graph. In either case, id actually stands for a node of the graph.

This special equational edge representation is used only if there is a singular outgoing edge with the given label; i.e., there are no distinct \( e_1, e_2 \) with \( \text{src}(e_1) = \text{src}(e_2) \) and \( \text{lab}(e_1) = \text{lab}(e_2) \). This way, it is never necessary to include edge indices in the equation.

Labelled graphs are used to represent concrete systems; in other words, they are on the level of individual programs or object diagrams. An example showing all of the graphical representation features is given in Figure 1. Here, \( y \) and \( z \) are symbolic names having no formal meaning within the graph, whereas \( 10 \) and 'yes' are data values of type
Fig. 1: Example graphical representation of a labelled graph

Int and String, respectively, and 123, 45 etc. are edge indices. The graph represented by this figure is formally (though not uniquely) captured by the following tables:

<table>
<thead>
<tr>
<th>Node</th>
<th>lab name</th>
<th>Edge</th>
<th>src lab ix tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>A z</td>
<td>$(n_1, \text{nh}, 1)$</td>
<td>$n_1$ nh 1 10</td>
</tr>
<tr>
<td>$n_2$</td>
<td>C</td>
<td>$(n_2, \text{xyz}, 45)$</td>
<td>$n_2$ xyz 45 $n_1$</td>
</tr>
<tr>
<td>$n_3$</td>
<td>A y</td>
<td>$(n_2, k1, 19)$</td>
<td>$n_2$ p1 19 $n_3$</td>
</tr>
<tr>
<td>$n_4$</td>
<td>A</td>
<td>$(n_2, \text{abc}, 45)$</td>
<td>$n_2$ abc 45 $n_3$</td>
</tr>
<tr>
<td>10 Int</td>
<td></td>
<td>$(n_2, \text{abc}, 88)$</td>
<td>$n_2$ abc 88 $n_4$</td>
</tr>
<tr>
<td>“yes” String</td>
<td></td>
<td>$(n_4, n, 8)$</td>
<td>$n_4$ n 8 “yes”</td>
</tr>
</tbody>
</table>

These tables should be read as follows:

- The first table defines the nodes and their labelling. Note that the node identities $n_1$ through $n_4$ are not part of the graphical representation and hence are not uniquely determined by it; instead, they can be picked at random from a pool of node identifiers (or, in an implementation, allocated dynamically). The node identities 10 and “yes”, on the other hand, are data values: they are fixed by the graph. The node labels are always shown in the graph. The third row captures the symbolic names introduced in the graph. These names are not properly part of the graph as it is defined in Definition 4, but are used to refer to nodes without relying on their identity — which, as we have seen above, may be randomly chosen. For instance, symbolic node names can be used to represent edges as equations written in (a second compartment) of a node, rather than as arrows. (Note that this notation is traditionally used for attributes only, but can be extended uniformly and without risk of confusion to other types of edges.)

- The second table defines the edges, by listing each edge as a triple of source node, label, and index. Due to the third clause of Definition 4 this indeed uniquely identifies every edge. The source node and label are always part of the graphical representation; the edge index, on the other hand, may be omitted, in which case (as for the node identities) it may be chosen at random. The table furthermore lists the functions required for edges in Definition 4; clearly, the first three of these merely select the appropriate elements from the edge structure, whereas the fourth is taken from the graph itself, taking the choice of node identities into account.
As a further illustration, Figure 2 shows two different ways to depict one and the same graph, which has a single non-data node with a single String-type attribute. Whereas the first representation coincides with the usual one and will probably be preferable in practice, the second notation makes clear that data values are, formally, graph nodes.

2.4 Graph morphisms

With respect to our aim of providing a sound and comprehensive formalisation of UML concepts, one aspect is not yet completely covered, namely the fact that node identities and edge indices are not uniquely determined by the diagrams. In this sense, the formal interpretation of the diagrams remains ambiguous.

The reason why we are nevertheless content with this solution is that this ambiguity is not harmful, because the choice in no way matters to the actual meaning. Put differently, it is allowed to abstract away from the precise identities, provided the nodes and edges remain distinguishable. The standard way to formalise this type of argument is by interpreting the structures under consideration — here, our graphs — up to or modulo some equivalence. In this particular case, the standard way to define an appropriate equivalence is through the notion of graph isomorphism.

**Definition 5 (graph (iso)morphism).** Given two graphs $G, H$, a morphism from $G$ to $H$ is a pair of mappings $f = (f_{\text{Node}}: \text{Node}_G \rightarrow \text{Node}_H, f_{\text{Edge}}: \text{Edge}_G \rightarrow \text{Edge}_H)$ such that

- Node and edge labels are preserved: $\text{lab}_H \circ (f_{\text{Node}} \cup f_{\text{Edge}}) = \text{lab}_G$;
- Sources and targets are preserved: $\text{src}_H \circ f_{\text{Edge}} = f_{\text{Node}} \circ \text{src}_G$ and $\text{tg}_H \circ f_{\text{Edge}} = f_{\text{Node}} \circ \text{tg}_G$.

$f$ is an isomorphism if $f_{\text{Node}}$ and $f_{\text{Edge}}$ are bijective, i.e., provide a one-to-one mapping between $\text{Node}_G$ and $\text{Node}_H$, resp. $\text{Edge}_G$ and $\text{Edge}_H$. We write $G \cong H$ (G is isomorphic to H) to denote that there is an isomorphism from $G$ to $H$.

It is especially important to realise that (iso)morphisms are not required to either respect node identities or edge indices, symbolic names, or diagram layout. Thus, for instance, the two graphs in Figure 3 are isomorphic, despite the fact that their edge indices and symbolic names are different, as is their layouting.

To be even more precise, assuming that the node identities of both graphs are chosen such that each $X$-labelled node has identity $n_X$, the following is the $f_{\text{Edge}}$ part of the
isomorphism from the left hand side graph to the right hand side graph of Figure 3:

\[
\begin{align*}
(x_C, abc, 45) & \rightarrow (x_C, abc, 18) \\
(x_C, abc, 88) & \rightarrow (x_C, abc, 7) \\
(x_C, qp, 45) & \rightarrow (x_C, qp, 93) \\
(x_B, kl, 27) & \rightarrow (x_B, kl, 2)
\end{align*}
\]

In general, we will set up all our definitions in such a way that they are insensitive to isomorphism; (meaning that a property holds for a graph \( G \) if and only if it holds for all \( H \cong G \)).

For one particular purpose we will later on strengthen the requirements on morphisms, in such a way that the ordering on edge indices is sometimes required to be preserved; namely, when we use the index to reflect an ordering over the edges themselves.

### 2.5 Type graphs

For purposes of documentation, structuring and correctness, it is common to impose a discipline over labelled graphs, comparable to the grammar of programming languages, or more to the point here, comparable to a class diagram. In particular, we use a **type graph** to impose local constraints on the allowed labels and connections between edges and nodes, and associated **constraints** to impose other, more sophisticated or less local, properties.

**Definition 6 (type graph).** A **type graph** is a tuple \( TG = \langle \text{NType}, \text{EType}, \text{src}, \text{tgt}, \text{inh} \rangle \) where

1. \( \text{NType} \subseteq \text{ID} \) is a set of node types and \( \text{EType} \subseteq \text{ID} \) a set of edge types;
2. \( \langle \text{NType}, \text{EType}, \text{src}, \text{tgt} \rangle \) is a graph, with \( \text{NType} \) as node set and \( \text{EType} \) as edge set, such that \( \text{src}(e) = \text{ns}(e) \) for any \( e \in \text{EType} \);
3. \( \text{inh} \subseteq \text{NType} \times \text{NType} \) is a reflexive partial ordering relation expressing that some node types inherit from others. (Reflexivity here means that \( T \text{ inh} T \) holds for all node types \( T \in \text{NType} \).)

We typically use capital letters \( (T, E) \) to range over node and edge types. We also use \( \text{ext} \subseteq \text{inh} \) to denote extension, which is a sub-relation of inheritance, defined such that \( X \text{ ext} Y \) if and only if (i) \( X \text{ inh} Y \) and (ii) there is no \( Z \) with \( Z \notin \{X, Y\} \) and \( X \text{ inh} Z \text{ inh} Y \). (Formally, \( \text{ext} \) is the direct predecessor relation inside \( \text{inh} \), or in other words, the smallest relation generating \( \text{inh} \).)
The condition on the source function of edges (clause 2 in the definition) states that the source type of an edge is at the same time its name space. Since edge types are identifiers and identifiers are pairs of names and namespaces (Definition 1), it follows that edge types are uniquely determined by their source type and name. This setup allows us to use edge types with the same name, but only for distinct source types — which is consistent with the situation in most, if not all, object-oriented paradigms.

Also note that $inh$ is a partial order, but not necessarily a forest: this implies that a node type can extend more than one other node type (in common terminology, our type graphs support multiple inheritance). At the same time, the partial order nature of $inh$ implies that there can be no inheritance cycles.

For “node type” in the definition above, one may for most purposes read “class;” the only difference is that the node types typically include data sorts. We say that $TG$ builds on a signature $Sig$ (or is a $Sig$-type graph), if $Sort \subseteq NType$. The operations of $Sig$ are not included in the type graph. In this paper we assume that there is no inheritance $s \ ext t$ with $s \in Sort$ and no edge $e$ with $src(e) \in Sort$.

A visual representation of a type graph can be given by drawing every node type as a box with the type identifier inscribed, every edge type as a “normal” arrow with the edge name as label, and every extension as an unlabelled arrow with triangular arrow head. Figure 4 shows an example type graph. This is very close to the traditional class diagram view, except that the data sorts are not treated as special cases (i.e., data type attributes are not distinguished from associations). Formally, this type graph is captured by the following tables of nodes and edges:

2.6 Typing and instance graphs

The meaning of a type graph is defined by the set of its (correctly typed) instances. The idea is that the instances of a type graph $TG$ are labelled graphs with labels chosen from the types of $TG$, and consistent with the graph structure of $TG$ modulo inheritance. To formalise it, we use the following auxiliary notation for arbitrary nodes $n$ and node types $T$, resp. edges $e$ and edge types $E$:

$$n:T :\leftrightarrow lab(x) \ inh T$$
$$e:E :\leftrightarrow lab(e) = E .$$

In words, $n:T$ expresses that the label of the node $n$ (in the instance graph under consideration) is a node type that inherits from $T$. Note that it follows that, for a given $n$, there can easily be more than one node type $T$ such that $n:T$, ranging from $T = lab(n)$ to all...
Definition 7 (instance graph). Let \( TG \) be a type graph. A labelled graph \( IG \) is typed by \( TG \), or an instance graph of \( TG \), if for every node \( n \in \text{Node} \) and every edge \( e \in \text{Edge} \):

- \( \text{lab}(n) \in \text{NType} \) and \( \text{lab}(e) \in \text{EType} \);
- \( \text{src}(e) : \text{src}(\text{lab}(e)) \) and \( \text{tgt}(e) : \text{tgt}(\text{lab}(e)) \).

The set of instance graphs of \( TG \) is denoted \( \text{Inst}[TG] \).

For instance, the labelled graph in Figure 1 is not an instance graph of the type graph in Figure 4, since it contains several edge labels that are not present in the type graph. An example of an instance graph for Figure 4 is shown in Figure 5.

3 Constraints

The concepts introduced in the previous section are, in the sense of existing graph theory, straightforward; in fact, the only non-standard concepts are the structure we have chosen for identifiers, and the fact that we are using indexed edges in labelled (instance) graphs. In this section, we introduce a way to enrich type graphs, and so constrain the set of valid instance graphs, in ways that formalise the concepts found in UML.

First of all, we give a general definition of a constraint set over a graph; then, we define a series of special types of constraints tuned towards UML concepts.

Definition 8 (graph constraint). Let \( TG \) be a type graph. A constraint set over \( TG \) is a tuple \( \langle \text{Con}, \text{sat} \rangle \) where

- \( \text{Con} \) is a set of graph constraints;
- \( \text{sat} \subseteq \text{Inst}[TG] \times \text{Con} \) is a satisfaction relation over the instances of \( TG \).

We denote \( IG \sat c \) to denote that an instance graph \( IG \) satisfies a constraint \( c \).

This definition only specifies that a graph constraint is something for which there exists an interpretation, expressed in terms of the graphs that satisfy the constraint. The interpretation is embodied in the satisfaction relation, \( \text{sat} \). The real question is how \( \text{sat} \) is defined. By combining type graphs with a constraint set, we arrive at the concept of a model, which is our equivalent to a UML class diagram.
Table 6: A classification of constraints

<table>
<thead>
<tr>
<th>Category</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node type</td>
<td>Abstractness</td>
</tr>
<tr>
<td>Association</td>
<td>Bidirectionality, Multiplicities, Indexing, Uniqueness</td>
</tr>
<tr>
<td>Containment</td>
<td>Acyclicity, Unsharedness</td>
</tr>
<tr>
<td>Specialisation</td>
<td>Subsetting, Redefinition, Union</td>
</tr>
<tr>
<td>General</td>
<td>OCL</td>
</tr>
</tbody>
</table>

Definition 9 (model). A model is a pair $\text{Mod} = \langle TG, \text{Con} \rangle$ where
- $TG$ is a type graph;
- $\text{Con}$ is a constraint set over $TG$, consisting of constraints of the types listed below.

We say that a labelled graph $G$ is an instance of $\text{Mod}$ if $G \in \text{Inst}[TG]$, and $G$ sat $c$ for all $c \in \text{Con}$.

The main contribution of this work, apart from the selection of the appropriate type and instance graph definitions, lies in the definition of a number of useful graph constraint “templates” and the corresponding satisfaction relations. The constraints can be subdivided into a number of categories, listed in Table 6. The constraints are collected in Table 17.

3.1 Abstractness constraints

A fairly common constraint is that a given class in a class diagram is abstract, meaning that it may have no direct instances; instead, only proper subclasses can be instantiated. In terms of type and instance graphs, this means that the node type may not itself be used as a label.

Definition 10 (abstractness constraint). Let $TG$ be a type graph. An abstractness constraint over $TG$ is a predicate $\text{abstract}(T)$ where $T \in \text{NType}$ is a node in $TG$. Satisfaction is defined for all $G \in \text{Inst}[TG]$ by

$$G \text{ sat } \text{abstract}(T) :\Leftrightarrow \nexists n \in \text{Node}_G. \text{lab}(n) = T.$$  

Figure 7 shows an example of an abstractness constraint. The type graph (left hand side) has an associated constraint $\text{abstract}(A)$, visualised by the annotation $\{\text{abstract}\}$ at the type node. The centre graph does not satisfy this constraint, as it contains a node labelled $A$. This is repaired in the right hand side graph.

Fig. 7: Example type graph with abstract node type.
3.2 Bidirectionality constraints

Associations in UML class diagrams have the property that they can (in principle) be traversed in either direction. Moreover, in general the ends of an association can have their own names. This is in contrast to the graphs of this paper, where edges are unidirectional. To model bidirectional associations, we therefore need two edges, one for either direction, which oppose each other.

**Definition 11 (bidirectionality constraint).** Let $TG$ be a type graph. A bidirectionality constraint over $TG$ is a pair $\text{oppose}(D, E)$ where $D, E \in E_{\text{Type}}$ are edges in $TG$, such that $\text{src}(D) = \text{tgt}(E)$ and $\text{tgt}(D) = \text{src}(E)$. Satisfaction is defined for all $G \in \text{Inst}[TG]$ by

$$G \text{ sat oppose}(D, E) \iff \forall n_1:\text{src}(D), n_2:\text{tgt}(D). \left| \{d \in \text{out}(n_1, D) \mid \text{tgt}(d) = n_2\} \right| = \left| \{e \in \text{out}(n_2, E) \mid \text{tgt}(e) = n_1\} \right|. $$

Figure 8 gives an example of a bidirectionality constraint. The type graph (left hand side) has an associated constraint $\text{oppose}(\text{B.c}, \text{C.b})$, visualised as a two-headed arrow. The centre graph does not satisfy this constraint, as there is a C.b-typed edge without an opposing B.c-typed one. In the right hand side graph this is repaired, so that this graph is a valid instance of the (enriched) type graph.

3.3 Multiplicity constraints

Multiplicity constraints regulate how many instances of an incident edge type are allowed per instance of a node type. In UML, this information is an important part of a class diagram. In order to formalise it, we first define what multiplicities are, and then introduce the corresponding graph constraint type.

Formally, multiplicities are defined as ordered pairs of natural numbers, where the second (higher) number may take the special value $\star$ to indicate unboundedness. A number is considered to be within the range of the multiplicity if it is higher than or equal to the lower bound and smaller than or equal to the upper bound.

**Definition 12 (multiplicity).** A multiplicity is a pair $[i, j] \in \mathbb{N} \times (\mathbb{N} \cup \{\star\})$ with $i \leq j$ or $j = \star$. The set of multiplicities is denoted $\text{Mult}$. The special value $\star$ indicates that the maximum number is not constrained.
To retrieve the lower and upper bounds of multiplicities, we use mappings $low, upp : \text{Mult} \rightarrow (\mathbb{N} \cup \{\ast\})$; i.e., if $\mu = [i, j]$ then $low(\mu) = i$ and $upp(\mu) = j$.

For an arbitrary number $n$ and multiplicity $\mu \in \text{Mult}$, we write $n \in \mu$ if $low(\mu) \leq n$ and either $upp(\mu) = \ast$ or $n \leq upp(\mu)$.

In UML, multiplicities can be attached to both the source and the target end of associations; a multiplicity at the source end represents the potential degree of sharing of a node. We will restrict ourselves to target multiplicities, and refer to them simply as the multiplicity of an edge.

**Definition 13 (multiplicity constraint).** Let $TG$ be a type graph. A multiplicity constraint over $TG$ is a pair $\text{mult}(E, \mu)$ with $E \in E\text{Type}$ and $\mu \in \text{Mult}$. Satisfaction is defined for all $G \in \text{Inst}[TG]$ by

$$G \text{ sat } \text{mult}(E, \mu) :\Leftrightarrow \forall n : src(E). |out(n, E)| \in \mu .$$

Figure 9 gives an example of a multiplicity constraint. The type graph (left hand side) has an associated constraint $\text{mult}(C.b, [2, 5])$, visualised by writing the pair $[2, 5]$, without the brackets, near the head of the arrow. The centre graph does not satisfy this constraint, as there is a $C$-typed node (call it $n$) with only a single outgoing $C.b$-typed edge (i.e., $out(n, C.b) = 1 \notin [2, 5]$). In the right hand side graph this is repaired, so that this graph is a valid instance of the (enriched) type graph.

### 3.4 Indexing constraints

To capture the notion of an ordered collection from class diagrams, we need to formalise what it means for a set of graph nodes to be ordered. To capture this correctly is actually quite involved, even though it is conceptually straightforward. Here we make use of the edge indices that are part of the instance graphs (see Definition 7); if an edge type is declared as indexed, the edge indices have to be picked from a consecutive range from 1 upwards; and moreover (in fact, more importantly), morphisms are required to respect the edge indices.

**Definition 14 (indexing constraint).** Let $TG$ be a type graph. An indexing constraint over $TG$ is a predicate $\text{indexed}(E)$, with $E \in E\text{Type}$. Satisfaction is defined for all $G \in \text{Inst}[TG]$ and all morphisms $f$ between instance graphs $G, H \in \text{Inst}[TG]$ by

$$G \text{ sat indexed}(E) :\Leftrightarrow \forall n : src(E), \forall e \in out(n, E). 1 \leq ix(e) \leq |out(n, E)|$$

$$f \text{ sat indexed}(E) :\Leftrightarrow \forall e : E \Rightarrow ix(f_{Edge}(e)) = ix(e) .$$
In the context of (type graphs with) indexing constraints, we will always implicitly restrict the (iso)morphisms under consideration to those satisfying the constraints.

Figure 10 gives an example of an indexing constraint. The type graph (left hand side) has an associated constraint indexed(C.b), visualised by the annotation {indexed} near the arrow head. The centre graph does not satisfy this constraint, as it has two outgoing C.b-typed edges with indices {54, 129}, which do not form a consecutive range. This is repaired in the right hand side graph. More importantly, where ordinarily the right hand side graph would be considered symmetric (having two interchangeable B-typed nodes), this is no longer true in the presence of the indexing constraint: the symmetry (formally, an isomorphism from the graph to itself) maps \((n, C.b, 0)\) to \((n, C.b, 1)\) (where \(n\) is the C-typed node in the graph) and hence does not preserve edge indices.

It follows that, in instance graphs, the edge indices for indexed edge types have to be included in the diagram, in order for the visual representation to be unambiguous.

3.5 Uniqueness constraints

In general, there may be more than one edge of the same type between the same nodes (as long as they are distinguished by their indices). In many cases, however, this is actually undesirable: one would like to ensure that target nodes occur at most once — in other words, they are unique in the context of the source node and edge type.

**Definition 15 (uniqueness constraint).** Let \(TG\) be a type graph. A uniqueness constraint over \(TG\) is a predicate \(\text{unique}(E)\), where \(E \in E\text{Type}\) is an edge type. Satisfaction is defined for all \(G \in \text{Inst}[TG]\) by:

\[
G \text{ sat } \text{unique}(E) \iff \\
\forall n: \text{src}(E). \forall e_1, e_2 \in \text{out}(n, E). \text{tgt}(e_1) = \text{tgt}(e_2) \Rightarrow e_1 = e_2 .
\]

Figure 11: Example type graph showing a uniqueness constraint.
Figure 11 shows an example of a uniqueness constraint. The type graph (left hand side) has an associated constraint $\text{unique}(C.b)$, visualised by the annotation $\{\text{unique}\}$ near the arrow head. The centre graph does not satisfy this constraint since there are two distinct $C.b$-labelled edges from the right hand side node to the left hand side node. This is repaired in the right hand side graph.

### 3.6 Acyclicity constraints

Another notion from UML class diagrams that has proved to be quite useful in practice is that of aggregation or containment. Whereas ordinary edges may impose an arbitrary structure on the nodes they connect, containment is intended to reflect a hierarchy of things. Therefore, when edges in a type graph are declared to be acyclic, the intention is that the edges in the corresponding instance graphs do not form a cycle.

This type of constraint is in fact quite powerful if the edge types in the hierarchy do form a cycle in the type graph. In that case, there could in principle be instance graphs with arbitrarily large cycles, all of which are ruled out by a single acyclicity constraint. From this it can be seen that the acyclicity constraint is a non-local property, and hence outside the class of first-order logic.

**Definition 16 (acyclicity constraint).** Let $TG$ be a type graph. An acyclicity constraint over $TG$ is a tuple $\text{acyclic}(E_1, \ldots, E_n)$ where $E_1, \ldots, E_n \in E_{\text{Type}}$ is a collection of edge types. Satisfaction is defined for all $G \in \text{Inst}[TG]$ by

$$G \ \text{sat} \ \text{acyclic}(E_1, \ldots, E_n) :\Leftrightarrow \{e : E_i \mid 1 \leq i \leq n\} \text{ is cycle free.}$$

It should be noted that the edge types $E_1, \ldots, E_n$ should be read as a set, in which the ordering is irrelevant; hence, there are many ways to write down the same constraint.

Figure 12 shows an example of an acyclicity constraint. The type graph (left hand side) has an associated constraint $\text{acyclic}(C.b, B.c)$ visualised by diamond-shaped decorations at the sources of the edge types. (Note that this visualisation always specifies a single acyclicity constraint, consisting of all diamond-decorated edge types. In other words, where a node or edge type could in principle be part of distinct acyclic hierarchies, this cannot be visualised without adding further distinguishing information to the diamonds, for instance in the form of identifiers.) Since the edge types $C.b$ and $B.c$ indeed form a cycle in the type graph, as remarked above this is a non-local constraint: in principle, there can be instance graphs with arbitrarily large $C.b/B.c$-typed cycles. The centre graph of Figure 12 shows a small instance of such a cycle; hence this graph violates the constraint. In the right hand side graph this is repaired, so that this is a valid instance of the (enriched) type graph.
3.7 Unsharedness constraints

The acyclicity constraint guarantees the absence of cycles (as its name suggests), but it does not guarantee the absence of sharing; in other words, on its own it is not certain that the structure imposed by acyclic edges is a forest. To complement this, we also introduce a constraint that specifies the absence of sharing; as will see, the UML composite is a combination of acyclicity and unsharedness.

**Definition 17 (unsharedness constraint).** Let TG be a type graph. An unsharedness constraint over TG is a tuple unshared\((E_1, \ldots, E_n)\), where \(E_1, \ldots, E_n \in EType\). Satisfaction is defined for all \(G \in \text{Inst}[TG]\) by:

\[ G \text{ sat } \text{unshared}(E_1, \ldots, E_n) :\Leftrightarrow \forall d: E_i, e: E_j, \ tgt(d) = tgt(e) \Rightarrow d = e . \]

Figure 13 shows an example of an unsharedness constraint. The type graph (left hand side) has an associated constraint unshared\((C.b1,D.b2)\), visualised by an annotation at one of the edge types (in this case, at D.b2) the filled diamond at the arrow source. The centre graph does not satisfy this constraint, since it has a B-type node with two distinct incoming C.b1- resp. D.b2-typed edges. This is repaired in the right hand side graph.

It follows that unsharedness in fact implies a particular form of source multiplicity: in a constraint unshared\((E_1, \ldots, E_n)\), each of the edge types \(E_i\) effectively receives source multiplicity \([0, 1]\). However, unsharedness also imposes a condition across edge types, and so goes beyond this multiplicity.

3.8 Subset constraints

We have included node type inheritance as a basic notion in type graphs, reflecting the common concept from UML and other object-oriented settings. For edges, on the other hand, although there is likewise a notion of specialisation, but no single commonly accepted way to capture this. Instead, UML knows several ways to define specialisation-like relationships between edges, which we here formalise through edge type constraints.

The first of these is subsetting. An edge type can be declared as a subset of another if both its source type and its target type specialise those of the other; its meaning is then that an edge of the supertype should exist whenever an edge of the subtype exists.

**Definition 18 (subset constraint).** Let TG be a type graph. A subset constraint over TG is a pair subset\((D, E)\), where \(D, E \in EType\) are edges in TG, such that...
Fig. 14: Example type graph with a subset constraint

$$\text{src}(D) \text{ inh src}(E) \text{ and tgt}(D) \text{ inh tgt}(E)$$.
Satisfaction is defined for all $$G \in \text{Inst}[TG]$$ by

$$G \text{ sat subset}(t, u) :\iff \forall d : D. \exists e : E. \text{src}(e) = \text{src}(d) \land \text{tgt}(e) = \text{tgt}(d)$$.

Figure 14 shows an example of a subset constraint. The type graph (left hand side) has an associated constraint \text{subset}(F.b, A.d), visualised by annotating the arrow head of F.b (the "subtype") with \{subsets d\}. (Note that d here is just the name, and not the entire identifier, of the edge type A.d; the namespace A can either be derived from the context — in particular the fact that it must be a supertype of F — or, if it is ambiguous, it should be included in the visualisation.) The centre graph does not satisfy the constraint, since it has a F.b-type edge between two nodes without a corresponding A.d-type edge. In the right hand side graph this is repaired, so that this graph is a valid instance of the (enriched) type graph. Note that the stand-alone d-named edge is actually of type A.d, and needs no corresponding F.b-edge; in other words, the implication in Definition 18 works in one direction only.

### 3.9 Redefinition constraints

Another specialisation-type relation between edges is that of \textit{redefinition}. This is in its intention quite similar to subset, but (loosely formulated) instead of \textit{implying} the presence of an edge of the “supertype”, redefinition \textit{forbids} the presence of such an edge. More precisely, if an edge type D redefines another type E, then a node of D’s source type may no longer have an outgoing E-type edge — instead, this should be a D-type edge.

\textbf{Definition 19 (redefinition constraint).} Let TG be a type graph. A redefinition constraint over TG is a pair redefine(D, E), where D, E \in \text{EType} are edges in TG, such that \text{src}(D) \text{ inh src}(E) \text{ and tgt}(D) \text{ inh tgt}(E). Satisfaction is defined for all $$G \in \text{Inst}[TG]$$ by:

$$G \text{ sat redefine}(D, E) :\iff \not\exists e : E. \text{src}(e) : \text{src}(D)$$.

Figure 15 shows an example of a redefinition constraint. The type graph (left hand side) has an associated constraint redefine(F.b, A.d), visualised just like the subset constraint (see above) but with the keyword redefine rather than subset. The centre graph does not satisfy the constraint, since there is an A.d-type edge going out of an F-type node. In the right hand side this is repaired, by changing the offending edge into an F.b-type; as a result, this instance graph satisfies the redefinition constraint.
3.10 Union constraints

Both subset and redefinition constraints impose consequences on the existence on a “subtype” edge — in one case, the existence and in the other case, the absence of a “supertype” edge. To complement this capability, the union works in the other direction: in this case, the existence of a “supertype” edge implies the existence of a “subtype” edge.

**Definition 20 (union constraint).** Let $TG$ be a type graph. A union constraint over $TG$ is a tuple $\text{union}(D, E_1, \ldots, E_n)$, where $D, E_1, \ldots, E_n \in EType$ are edges in $TG$, such that $\text{src}(E_i) \in \text{src}(D)$ and $\text{tgt}(E_i) \in \text{tgt}(D)$ for all $1 \leq i \leq n$. Satisfaction is defined for all $G \in \text{Inst}[TG]$ by:

$$G \text{ sat } \text{union}(D, E_1, \ldots, E_n) :\Leftrightarrow \forall 1 \leq i \leq n : \text{subset}(E_i, D) \land \forall d : D. \exists 1 \leq i \leq n, e : E_i. \text{src}(e) = \text{src}(d) \land \text{tgt}(e') = \text{tgt}(e) .$$

Figure 16 shows an example of a union constraint. The type graph (left hand side) has an associated constraint $\text{union}(A.d, F.b, F.c)$, visualised by an annotation at the “supertype”, that is, at the edge type A.d. The centre graph does not satisfy this constraint since there is an F.b-edge without a corresponding “supertype” edge, but also an A.d-edge without a corresponding “subtype” edge. This is repaired in the right hand side graph by adding both of those edges; as a result, this instance graph satisfies the redefinition constraint.

---

**Fig. 15:** Example type graph with a redefinition constraint

**Fig. 16:** Example type graph showing a union constraint.
3.11 OCL constraints

Wherever there is a need for specifying properties that cannot be expressed by the built-in UML concepts, one can always associate an OCL constraint [13] with a UML class diagram. Since the subject of this paper is not to define another OCL semantics, to cap-
pose existing work [13, 15], where in essence a translation is defined from OCL to first order logic (FOL) over typed structures. For the purpose of this paper, we assume this semantics to be given in the form of a mapping $\llbracket \cdot \rrbracket$ which, for any OCL constraint $\phi$, yields a first order logic formula $\llbracket \phi \rrbracket$; moreover, we assume an interpretation $\models \subseteq \Inst[TG] \times \text{FOL}$ which provides an interpretation of FOL formulae over instance graphs of $TG$. On this basis, we introduce a graph constraint template which provides a bridge from the existing OCL semantics to our framework.

**Definition 21 (OCL constraint).** Let $TG$ be a type graph. An OCL constraint over $TG$ is a predicate $\text{ocl}(\phi)$, where $\phi$ is an OCL formula. Satisfaction is defined for all $G \in \Inst[TG]$ by:

$$G \text{ sat } \text{ocl}(\phi) \iff G \models \llbracket \phi \rrbracket .$$

4 UML Semantics

So far, we have set up a general framework of definitions regarding two special forms of graphs: type graphs and labeled graphs. This is then extended to the notion of model, which is a combination of a type graph and a constraint set. Furthermore, we have defined a number of types of constraints, and we have defined what it means for a labeled graph to be an instance of a model. In this section we will apply this general framework to UML class and object diagrams, thus providing a formal, graph-based semantics for these diagrams.
Table 18: Mapping of UML class diagram concepts to graphs

<table>
<thead>
<tr>
<th>Category</th>
<th>UML class diagram</th>
<th>Graph model</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>class</td>
<td>type node</td>
</tr>
<tr>
<td></td>
<td>primitive type attribute</td>
<td>type edge $E$ with $tgt(E) \in \text{Sig}$</td>
</tr>
<tr>
<td></td>
<td>non-primitive type attribute</td>
<td>type edge $E$ with $tgt(E) \notin \text{Sig}$</td>
</tr>
<tr>
<td>Association</td>
<td>directed association</td>
<td>type edge</td>
</tr>
<tr>
<td></td>
<td>non-directed/bi-directional multiplicity</td>
<td>pair of type edges with oppose mult-constraint</td>
</tr>
<tr>
<td></td>
<td>set (default for mult $&gt; 1$)</td>
<td>unique but not indexed</td>
</tr>
<tr>
<td></td>
<td>bag</td>
<td>neither indexed nor unique</td>
</tr>
<tr>
<td></td>
<td>sequence</td>
<td>indexed but not unique</td>
</tr>
<tr>
<td></td>
<td>ordered set</td>
<td>both indexed and unique</td>
</tr>
<tr>
<td></td>
<td>aggregation</td>
<td>acyclic</td>
</tr>
<tr>
<td></td>
<td>composition</td>
<td>both acyclic and unshared</td>
</tr>
<tr>
<td></td>
<td>OCL constraint</td>
<td>ocl</td>
</tr>
<tr>
<td>Specialisation</td>
<td>inheritance</td>
<td>inh-relation on type nodes</td>
</tr>
<tr>
<td>Specialisation</td>
<td>subset</td>
<td>subset constraint</td>
</tr>
<tr>
<td>Specialisation</td>
<td>redefines</td>
<td>redefine constraint</td>
</tr>
<tr>
<td>Specialisation</td>
<td>union</td>
<td>union constraint</td>
</tr>
</tbody>
</table>

UML object diagram | Labelled graph
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
</tr>
<tr>
<td>object</td>
<td>node</td>
</tr>
<tr>
<td>object type</td>
<td>node label</td>
</tr>
<tr>
<td>link</td>
<td>edge</td>
</tr>
<tr>
<td>link type</td>
<td>edge label</td>
</tr>
<tr>
<td>instance name</td>
<td>symbolic id</td>
</tr>
</tbody>
</table>

4.1 Class Diagrams

The formal meaning of a UML class diagram is that it is a model. An overview of the mapping of UML class diagram concepts to the concepts in our framework can be found in Table 18. The model’s type graph can be easily recognized: each class in the diagram is a node and each directed association is an edge. Non-directed associations translate to pairs of edges with a bi-directionality constraint, where the edge labels correspond to the names of the association ends. Note that our semantics does not provide the possibility to add a name to the association, i.e. to the pair of edges.

Taking into account the abbreviated notation for attributes as presented in Figure 2, an attribute is an edge to a node from the data signature. In the case of UML the data signature consists of the four primitive types (String, Real, Int and Boolean). UML is not very strict in demanding that the type of each attribute should be a primitive type. Therefore, an attribute whose type is non-primitive (i.e. another class) is an edge to a node that is not part of the data signature. In other words, its meaning is exactly the same as a directed association.

4.2 Constraint Types

Most of the constraint types in our graph-based framework can also be easily recognized in a class diagram, for instance a multiplicity or bidirectionality constraint is shown in
a class diagram in the same manner as we have shown in Figures 8 and 9. However, in class diagrams a number of constraint types are often used in combination. The UML notation is such that there are separate symbols for certain combinations of constraint types.

First, we have defined indexing and unique separately, whereas in UML there are four forms of notation for each combination of these constraints.

1. The default situation in a UML class diagram is that the unique constraint holds for every association, whereas the indexing constraint does not, i.e. an association end represents a set. Whenever no adornment is attached to the association end, the default applies.
2. The notation for the situation where both unique and indexing hold, is to adorn the association with ordered, which according to the UML specification [12] shows “that the end represents an ordered set.”
3. The UML notation bag as adornment to an association means that neither the unique nor the indexing constraint type hold.
4. Finally, when an association end is adorned with sequence, this means that the unique constraint does not hold, but the indexing constraint does.

The second combination of constraints that is commonly denoted by one symbol in a UML class diagram, is acyclic with unshared. The case were both hold, is known in UML as a composition or composite aggregation, which is denoted using a black diamond at the source end of the association edge. When acyclic holds, but unshared not, the case is known in UML as an aggregation, which is denoted by a white diamond at the source end of the association edge.

### 4.3 OCL Constraints over Class Diagrams

UML class diagrams can be accompanied by OCL constraints. In our semantics, as we have seen, these are also translated to graph constraints, of the type $\text{ocl}(\phi)$, by relying on the existing OCL semantics (which essentially provides a translation to first order logic). In other words, a class diagram together with its OCL constraints is translated to a model, in the sense of Definition 9. This illustrates the fact that OCL constraints cannot be seen as separate from the class diagram. The base of every OCL expression lies in the type system defined by a class diagram.

### 4.4 Object Diagrams

Finally we need to address object diagrams. It will be no surprise that we define an object diagram to be a labeled graph. When a labeled graph satisfies a certain model, for instance a class diagram or a class diagram combined with constraints, it is a valid instance of that model. An overview of the mapping of UML object diagram concepts to the concepts in our framework can be found in Table 18.
4.5 Evaluation

The semantics presented here should, as any semantics, uphold the commonly known characteristics of UML diagrams, even those that have (unfortunately) not been made explicit in the UML specification. A few such characteristics are listed below. This (incomplete) list should provide some confidence in the correctness of our semantics with respect to the common intuition. Let \( \langle TG, C \rangle \) be the model representing the class diagram under consideration.

- The common sense that the \{bag\}, \{sequence\}, and \{ordered\} annotations should always be used in combination with a multiplicity larger than one, can be obtained from our semantics in the following way. Suppose we have \( \text{mult}(E, \mu) \in C \) with \( \text{upp}(\mu) \leq 1 \) for some edge type \( E \), meaning that any instance graph \( G \) satisfies
  \[
  \forall n : \text{src}(E), |\text{out}(n, E)| \leq 1 .
  \]
  It immediately follows that also \( G \text{ sat unique}(E) \), so adding \( \text{unique}(E) \) to \( C \) does not change the model. This can also be formulated as an implication:
  \[
  \text{mult}(E, \mu) \land \text{upp}(\mu) \leq 1 \Rightarrow \text{unique}(E) .
  \]
  Furthermore, without loss of generality (since all choices lead to isomorphic graphs) we can assume \( i_x(e) = 1 \) for all \( e \in \text{out}(n, E) \). This in turn implies that
  \[
  \forall n : \text{src}(E), \forall e \in \text{out}(n, E), 1 \leq i_x(e) \leq |\text{out}(n, E)| \leq 1 .
  \]
  and hence \( G \text{ sat indexed}(E) \); so, the same implication holds with respect to \( \text{indexed}(E) \).

- A commonly known characteristic of UML class diagrams is that if the one end of a bi-directional association is marked \{bag\} then the other end should be marked \{bag\} or \{sequence\}. Likewise, if one end is an (ordered) set, the other end must be a set as well.
  In our semantics, this situation arises when the bi-directionality constraint is combined with the uniqueness constraint. This UML characteristics then translates to the following “law” of graph constraints:
  \[
  \text{oppose}(D, E) \Rightarrow (\text{unique}(D) \Leftrightarrow \text{unique}(E))
  \]

- According to the UML specification, the acyclicity constraint should always be combined with bi-directionality: “Only binary associations can be aggregations” ([12], page 37). At the same time, only one end of this association can be marked as aggregate. This is in accordance with common sense, which says that a part cannot contain its container. In our framework this forbidden situation would occur when the acyclicity constraint is defined for both edges of a bi-directional association, i.e.:
  \[
  \text{oppose}(D, E) \land \text{acyclic}(D) \land \text{acyclic}(E) .
  \]
  In our semantics, there are no instances that would satisfy such a model; in other words, this combination of constraints is inconsistent (i.e., a contradiction).
Again according to the UML specification “a multiplicity on an aggregate end of a composite aggregation must not have an upper bound greater than 1” [12], page 119). In terms of graph constraints, this means that the following must hold:

\[
\text{oppose}(D, E) \land \text{acyclic}(E) \land \text{unshared}(E) \Rightarrow \text{mult}(D, \mu) \text{with upp}(\mu) \leq 1.
\]

In our semantics, \text{unshared}(E) implies that for any \(n:tgt(E)\) there can be at most one incoming edge \(e_1:E\) such that \(n = tgt(e_1)\). Because of the bi-directionality constraint there can then also be at most one edge \(e_2:D\) with \(src(e_2) = n\), namely the one paired with \(e_1\). Thus, the informal characteristic is confirmed by our formal, graph-based semantics.

These cases give confidence that the presented semantics really conforms to the intuition behind UML. On the other hand, our semantics does not support all UML aspects; in particular, the following are not included:

- Names of associations. Only the role names associated with the association ends are taken into account, because we consider these to be more important.
- N-ary associations, i.e. associations between more than two classes. These tend to occur very rarely in class diagrams.
- Derived attributes or association ends. As the name suggests, these are derived values and need not be explicitly part of the formal representation.
- Navigability of associations. We feel that the directionality of the edges in the association pair provides enough information.
- Operations. These cannot be expressed by a static structure.

5 Conclusions

In this paper we present an elegant and simple semantics of UML class and object diagrams based on graph structures that are as close as possible to familiar notions in graph theory. The main insight used is that a UML class diagram cannot be treated as a simple type graph. It is a much richer structure, which is embodied in our framework by the use of (graph) constraints. The use of constraints also makes it possible to change the given semantics to include or exclude certain semantic elements. For instance, by disallowing the abstract class constraint type one can easily define class diagrams without abstract classes. Furthermore, our definitions do not only provide a semantics for both diagram types, but for the relationship between them as well.

As stated in the introduction, simplicity was one of our main guidelines. The only addition we made to the familiar notion of labelled graph is the edge indexing function, in order to capture ordered associations. We investigated (and rejected) several alternatives. The use of special “collection node types” would have made our definition of instance graph much more complex. Another possibility would have been to use hypergraphs, but that in itself makes the model much more complex. A third option is to use special edges between the target nodes of an ordered association to represent the ordering, as we have done before in [8]. The problem with this solution is that the ordering needs to be local not only to all edges of the given type, but also to the source node. For
instance, in Figure 10 one could add a “next”-edge between the two B-typed nodes, but when another C-typed node would in the same manner be connected to the two B-typed nodes there would be no way to distinguish between the two “next”-edges.

In the introduction we stated that the quality of models is determined by precision, consistency, and completeness. Our semantics provide a precise meaning to class and object diagrams. Furthermore, the consistency of a model, i.e. the existence of instances, can be checked using the given definitions. For instance, we can prove that a model with a bi-directional association that is an aggregate in both directions is inconsistent. Research into this “logic of UML models” has so far been scarce (see e.g. [11]). The completeness of a model cannot be guaranteed by our semantics. However, the semantics themselves are more complete than any other graph-based semantics that we have found in the literature. For instance, [1] only visualizes a type graph with inheritance as a UML class diagram, thus implying a graph-based meaning for class diagrams without actually defining semantics. [17] includes attributes, associations, and inheritance, but not containment, multiplicity, abstract classes, or edge specialisation.

As a next step, we intend to investigate the integration of our framework with the existing theory of graph transformation. An important issue is to reconcile our encoding of indexed edges with the requirements of algebraic graph transformations.

A final point we would like to make goes back to the introduction, and concerns the (scientific) merit of the type of effort we have undertaken in this paper. Although we believe to have ended up with a fairly simple, intuitive and workable semantics, this was not in itself a simple task (as evidenced by the relative lack of completeness of previous efforts, e.g., [10, 6, 14, 9, 4, 2, 7]). Moreover, there is a great need of such formalisations, if ever model-driven engineering is to become a dependable method. However, at the same time we know full well that our end product, being a set of definitions only, will not be judged as innovative, indeed runs the risk of being rejected due to lack of results. There is a contradiction here. If there is no well-defined forum where this type of effort can receive recognition, there will be no incentive, and the gap between theory and practice may remain with us forever.

References


