Typing Graph-Manipulation Operations

Jan Hidders

University of Antwerp (UIA)
Universiteitsplein 1
B-2610 Antwerp, Belgium.
E-mail: hidders@uia.ua.ac.be

Abstract. We present a graph-based data model called GDM where database instances and database schemas are described by certain types of labeled graphs called instance graphs and schema graphs. For this data model we introduce two graph-manipulation operations, an addition and a deletion, that are based on pattern matching and can be represented in a graphical way. For these operations it is investigated if they can be typed such that it is guaranteed for well-typed operations that the result belongs to a certain database schema graph, and what the complexity of deciding this well-typedness is.

1 Introduction

Since the introduction of the Entity-Relationship (ER) Model [1] labeled graphs have been used in many data models to represent schemas. This is not only the case for subsequent extensions of the ER Model but also in object-oriented data models such as LDM (the Logical Data Model) [2] and IFO [3].

Although the representation of instances as labeled graphs was already considered in FDM (the Functional Data Model) [4] the first data model that explicitly used labeled graphs as instances and regarded database transformation as graph transformations was GOOD (the Graph-Oriented Object Database Model) [5]. To represent complex values more naturally some data models have proposed to use generalizations of graphs such as *hypergraphs* [6], *hierarchical graphs* [7], *hygraphs* as used in the Hy^+ system [8], and finally graphs in the *hypernode model* [9].

The data model we propose here is based on the GOOD approach and represents both instances and schemas as labeled graphs called instance graph and schema graphs. The differences with GOOD are that (1) these two notions are defined independently such that instance graphs can exist without a corresponding schema graph and the data model can be used to represent *semistructured data* [10, 11] and (2) the data model explicitly supports the notions of *complex values*, *inheritance* and n-ary *symmetric relationships* as are found in ER models.

The language introduced with the GOOD data model was one of the first graph-based languages that was shown to be able to express all *constructive* database transformations [12]. This was followed by languages such as PaMaL [13, 14] and GOAL [15] that showed that only an addition and a deletion are

sufficient to express all these transformation if certain nodes explicitly represent complex values. In this paper we show how these two operations can be redefined for our data model and we investigate the decidability of whether these operations respect a certain schema, i.e., whether the result of an operation will stay within that schema.

2 The Graph-based Data Model GDM

2.1 Instance Graphs

In GDM an instance is represented by labeled graphs such as shown in Fig. 1 which are called *instance graphs*. The nodes in the graph represent entities, the edges represent attributes of these entities and the nodes are labeled with zero or more class names to indicate that they belong to certain classes. If from a certain node several edges leave that have the same label then this is interpreted as a single set-valued attribute. For example, the Department in Fig. 1 has an attribute sections that contains two sections. As is usual in object-oriented databases we distinguish three mutually exclusive kinds of entities [16]: *objects* which are represented by square nodes, *composite values* (or complex values) which are represented by round empty nodes and *basic values* which are represented by round nodes with a basic-type name inside and labeled with a representation of the basic value it represents.

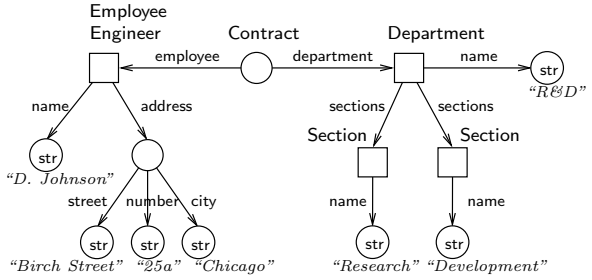

Fig. 1. An instance graph

Before we proceed with the formal definition of instance graphs we postulate the following symbols and sets. The postulated symbols are: **isa**, to label **isa** edges with, **is**, to label **is** edges with, **com**, to indicate composite-value nodes, and **obj**, to indicate object nodes. Given a set X we let $\mathcal{P}(X)$ denote the *power set* of X, i.e., the set of subsets of X, and $\mathcal{P}_{fin}(X)$ denotes the set of *finite* subsets of X. The postulated sets are: the set $\mathcal{A}$ of attribute names, not containing **isa** or **is**, the set $\mathcal{B}$ of basic-type names, not containing **com** and **obj**, the set $\mathcal{C}$ of class names, the set $\mathcal{D}$ of representations of basic values, and the function $\delta : \mathcal{B} \to \mathcal{P}(\mathcal{D})$ that gives for every basic-type name the corresponding domain of basic-value representations.

The definition of instance graph will be based on the notion of *data graph*. A data graph is defined as $I = \langle N, E, \lambda, \sigma, \rho \rangle$ where $\langle N, E, \lambda \rangle$ is a finite labeled graph with node labels $\mathcal{P}_{fin}(\mathcal{C})$ and edge labels $\mathcal{A}$, and with the function $\sigma : N \to \{\mathbf{com}, \mathbf{obj}\} \cup \mathcal{B}$ that gives the *sort* of every node, and the function ρ :

$\{ n \in N \mid \sigma(n) \in \mathcal{B} \} \rightarrow \mathcal{D}$ that gives a basic-value representation for basic-value nodes.

In a data graph nodes with sort **obj** are called *object nodes*, nodes with sort **com** are called *composite-value nodes* and nodes with a sort in $\mathcal{B}$ are called *basic-value nodes*. If $\lambda(n) = \emptyset$ then n is called a *class-free node* and otherwise it is called a *class-labeled node*. If the components of I are not explicitly named then they are presumed to be N_I, E_I, λ_I, σ_I and ρ_I, respectively.

A data graph is called a *well-formed data graph* if (I-BVA) no edge leaves from a basic-value node, (I-BVT) if $\rho(n)$ is defined then $\rho(n) \in \delta(\sigma(n))$, (I-REA) for every class-free node n there is a path that ends in n and starts in a class-labeled node, and (I-NS) composite-value nodes have either exactly one incoming edge or are labeled with exactly one class name, but not both.

The constraints I-BVA and I-BVT follow from the meaning of basic-value nodes. The constraint I-REA is introduced to prevent "floating entities", i.e., entities that do not belong to any class and are not in the value of any attribute. Finally, the constraint I-NS is introduced because we assume that representations of composite values are weak entities that are not identified by only their attributes but also by the attribute or class that they belong to. This is consistent with, for example, the relational model and complex-value data models where an update to a tuple in a certain table or attribute does not imply that representations of the same tuple in other tables or attributes are also updated.

To establish when two nodes in a data graph I represent the same complex value we define the *value-equivalence* relation $\cong_I \subseteq N_I \times N_I$ as the smallest reflexive relation such that (1) two basic-value nodes are value equivalent if they are labeled with the same basic-value representation and (2) two composite-value nodes n_1 and n_2 are value equivalent if for every edge $\langle n_1, \alpha, n_1' \rangle$ in E_I there is an edge $\langle n_2, \alpha, n_2' \rangle$ in E_I such that $n_1' \cong_I n_2'$ and vice versa.

Finally, we define *instance graphs* as well-formed data graphs for which it holds that (1) basic-value nodes are value-equivalent only if they are the same node, (2) two composite-value nodes that are labeled with the same class name are value-equivalent only if they are the same node and (3) two composite-value nodes that have both an incoming edge from the same node are value-equivalent only if they are the same node.

2.2 Schema Graphs

In GDM a schema is represented by labeled graphs such as shown in Fig. 2 which are called *schema graphs*. The nodes in a schema graph represent classes and the edges labeled with attribute names indicate that entities in that class may have that attribute. The nodes are labeled with zero or one class name to indicate the name of the class. As in data graphs the nodes have a sort which in this case indicates the sort of the entities in this class. The special edges that are not labeled with an attribute name but are drawn as hollow edges are **isa** edges that indicate that the class where the edge leaves is a subclass of the class where it arrives, i.e., all entities in the first class also belong to the second class.

Formally, a schema graph is defined as $S = \langle N, E, \lambda, \sigma \rangle$ where $\langle N, E, \lambda \rangle$ is a finite partially labeled graph, i.e., λ may be undefined for some nodes, with node labels $\mathcal{C}$ and edge labels $\mathcal{A} \cup \{\mathbf{isa}\}$, and the function $\sigma : N \rightarrow \{\mathbf{com}, \mathbf{obj}\} \cup \mathcal{B}$ gives the *sort* of every node.

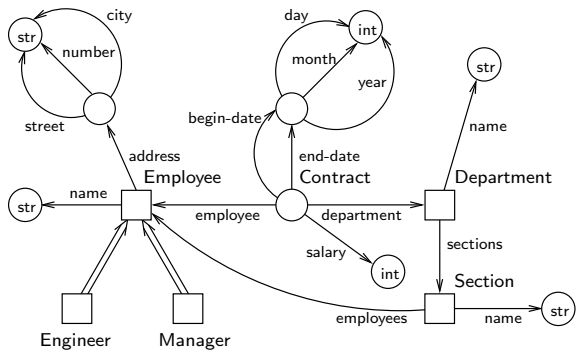

Fig. 2. A schema graph

Nodes with sort **obj** are called *object class nodes*, node with sort **com** are called *composite-value class nodes* and nodes with a sort in $\mathcal{B}$ are called *basic-value class nodes*. If $\lambda(n)$ is undefined then n is called an *implied class node* and otherwise n is called an *explicit class node*. The difference between an explicit class and an implied class is not that one has a name and the other does not, but rather that for the explicit class class-membership is explicitly indicated in the instance graph. Note that this is similar to the usual distinction between classes and types.

If the components of a schema graph S are not explicitly named then they are presumed to be N_S, E_S, λ_S and σ_S, respectively. The reflexive transitive closure of the binary relation over N_S that is defined by the **isa** edges is written as $\mathbf{isa}_S^*$.

2.3 Extension Relations

The relationship between data graphs and schema graphs is established through *extension relations* which are many-to-many relations between the nodes in the data graph and nodes in the schema graph that indicate which entity belongs to which class. Such a relation must respect the meaning of the class names, the attribute edges and the **isa** edges, i.e., the extension relation must at least associate the nodes that should be associated according to these labels and edges. Therefore we define an extension relation from a schema graph S to a data graph I is defined as a relation $\xi \subseteq N_S \times N_I$ for which it holds that[1] (ER-CLN) if $\lambda_S(m)$ is defined and $\lambda_S(m) \in \lambda_I(n)$ then $\xi(m, n)$, (ER-ATT) if $\xi(m_1, n_1)$, $\langle n_1, \alpha, n_2 \rangle \in E_I$ and $\langle m_1, \alpha, m_2 \rangle \in E_S$ then $\xi(m_2, n_2)$, and (ER-ISA) if $\langle m_1, \mathbf{isa}, m_2 \rangle \in E_S$ and $\xi(m_1, n)$ then $\xi(m_2, n)$.

A node in the data graph should be labeled with a class name iff it belongs to a class with that name. Therefore we say that an extension relation ξ from S to I is *class-name correct* if whenever $\lambda_S(m)$ is defined and $\xi(m, n)$ then $\lambda_S(m) \in \lambda_I(n)$.

[1] We will usually abbreviate $\langle m, n \rangle \in \xi$ to $\xi(m, n)$.

Another requirement is that nodes of different sorts may not be associated with each other. Therefore we say that an extension relation from S to I is *sort correct* if it only associates nodes with the same sort.

Finally, we require that all nodes, edges and class names in the data graph are somehow justified by corresponding nodes, edges and class names in the schema graph. We say that an extension relation ξ from S to I *covers* I if (CV-N) for every node $n \in N_I$ then there is some node $m \in N_S$ such that $\xi(m, n)$, (CV-E) for every edge $\langle n_1, \alpha, n_2 \rangle$ in E_I there is some edge $\langle m_1, \alpha, m_2 \rangle$ in E_S such that $\xi(m_1, n_1)$ and $\xi(m_2, n_2)$, and (CV-C) for every node $n \in N_I$ and class name $c \in \lambda_I(n)$ there is some explicit class node $m \in N_S$ such that $\xi(m, n)$ and $c = \lambda_S(m)$.

Summarizing, we say that a data graph I *belongs to a schema graph* S if it holds for the minimal extension relation from S to I that it is class-name correct, sort correct and covers I. It can be verified that under this definition the data graph in Fig. 1 belongs indeed to the schema graph in Fig. 2. To understand why only the *minimal* extension relation is considered look, for example, at the instance graph in Fig. 3 that would otherwise have belonged to the schema graph in Fig. 2 because the node at the end of the address edge might have been associated with the node at the end of the begin-date edge in the schema graph.

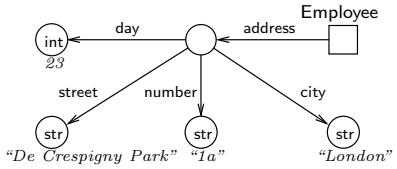

Fig. 3. An instance graph that not belongs to the schema graph in Fig. 2

2.4 Discussion of GDM

The purpose of GDM is to present a data model that is based on labeled graphs such that we can think of data manipulations as graph manipulations, and at the same time generalize over existing data models such as object-oriented data models and semi-structured data models.

An example of a simulation of an object-oriented schema is shown in Fig. 4 (a). This schema graph defines a class B and a subclass A that inherits the attributes of B.

By allowing composite-value class nodes to play the same roles as object-class nodes GDM can also simulate the relational model [17] and even the

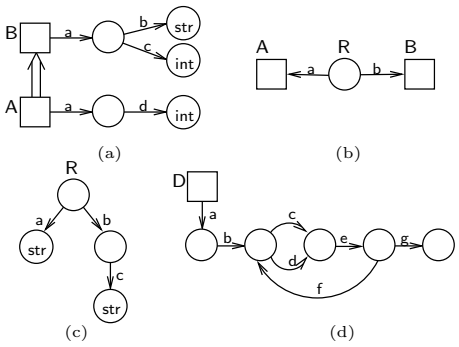

Fig. 4. Simulating other data models

nested relational model [18] that allows non-first-normal-form relations and the Format Model [19] that allows arbitrary nesting of sets, tuples and tagged unions. An example of a simulation of this in GDM is given in Fig. 4 (c).

Moreover, since the attributes of composite values can also contain objects we can represent relationships between objects as shown in Fig. 4 (b). This enables GDM to simulate relationships as found in the ER model and ORM [20].

Finally, GDM can also be used to describe semistructured data because because instance graphs are self-describing and exist independently of any schema graph, it can represent arbitrarily nested values and a schema graph can express that the paths in a composite-value tree form a prefix of a certain regular expression. For example in Fig. 4 (d) the attribute of an D object can contain only paths that are a prefix of ab(c∪d)e(f(c∪d)e)*g.

3 The Graph-based Update Language GUL

The basic mechanism of GUL is pattern matching. This means that every operation contains a pattern, i.e, a labeled graph that is similar to a well-formed data graph, and everywhere in the instance graph that this pattern can be embedded the operation is performed. Based on this mechanism we define an addition and a deletion and finally we introduce a reduction operation that reduces well-formed data graphs to instance graphs.

3.1 Patterns

The main difference between data graphs and patterns is the presence of **is** edges between composite-value nodes, which are drawn as hollow undirected edges. An **is** edge between two nodes specifies that these must be embedded upon two value-equivalent nodes in the instance graph. An example of a pattern with an **is** edge is given in Fig. 5. This pattern looks for an Engineer and a Manager that work for the same Section and live at the same address.

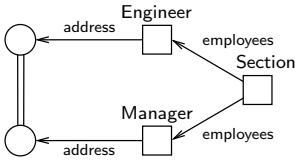

Fig. 5. A pattern with an **is** edge

More formally we define a pattern as a well-formed data graph $J = \langle N, E, \lambda, \sigma, \rho \rangle$ except that ρ may be undefined for certain basic value nodes and we allow extra edges labeled with **is** between composite-value nodes if (1) these edges are symmetric, i.e., for every edge $\langle n_1, \mathbf{is}, n_2 \rangle$ there is an edge $\langle n_2, \mathbf{is}, n_1 \rangle$ and (2) in every cycle of edges with at least one attribute edge there is at least one object node. The final constraint is necessary because recursive composite-values are not allowed in well-formed data graphs.

An *embedding* of a pattern J in a well-formed data graph I is a function $h : N_J \to N_I$ that respects the class names, the sorts, the basic-value representations, the attribute edges and the **is** edges, i.e., for all nodes n_1 and n_2 in N_J

it holds that (1) $\lambda_J(n_1) \subseteq \lambda_I(h(n_1))$, (2) $\sigma_J(n_1) = \sigma_I(h(n_1))$, (3) if $\langle n_1, r\rangle \in \rho_J$ then $\langle h(n_1), r\rangle \in \rho_I$, (4) if $\langle n_1, \alpha, n_2\rangle$ in E_J then $\langle h(n_1), \alpha, h(n_2)\rangle$ in E_I, and (5) if $\langle n_1, \mathbf{is}, n_2\rangle \in N_J$ then $h(n_1) \cong_I h(n_2)$. The set of all embeddings of J into I is written as $Emb(J, I)$.

3.2 The Addition

An addition is specified by giving two patterns; a *base pattern* J and an *extension pattern* J' that is, except for the **is** edges, a super-graph[2] of the base pattern with extra nodes, edges and class names such that all basic-value nodes in J' that are not in J are labeled with a basic-value representation. We will write such an addition as $\texttt{Add}(J, J')$ where J is the base pattern and J' is the extension pattern. An example of an addition is shown in Fig. 6.

The nodes, edges, and class names of the base pattern are drawn with normal lines and written in a normal font, whereas the additional nodes, edges and class names in the extension pattern are drawn with bold lines and written in a bold font.

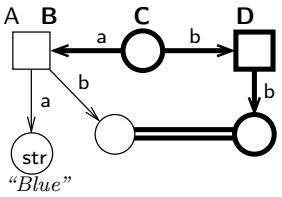

Fig. 6. An addition

Informally the result of applying an addition $\texttt{Add}(J, J')$ to a well-formed data graph I is obtained by extending I for each embedding of J in I with the nodes, edges and class names that are in J' and it additionally adds nodes and edges to satisfy the **is** edges in J'. More formally we define the result of applying an addition $\texttt{Add}(J, J')$ to a well-formed data graph I, written as $[\![\texttt{Add}(J, J')]\!](I)$, as the well-formed data graph I' where I' is a minimal super-graph of I such that there is a function $\eta : Emb(J, I) \to Emb(J', I')$ such that (1) $\eta(h)$ equals h on N_J, (2) all distinct nodes in $N_{J'} - N_J$ are mapped by $\eta(h)$ to distinct nodes in $N_{I'} - N_I$, and (3) extensions of distinct embeddings map nodes in $N_{J'} - N_J$ to distinct nodes.

For example, the addition in Fig. 6 does for every embedding of the base pattern with the A node, the *"Blue"* node and the composite-value node the following: (1) it adds the extra nodes in the extension pattern, i.e., the C node, the D node and the second composite-value node, (2) it adds the class name B to the A node, and (3) it extends the new composite-value node such that it represents the same composite-value as the old composite-value node.

It is easy to see that the result of an addition without **is** edges in the extension pattern is always well-defined. However, in order for the result of an addition with **is** edges in the extension pattern to be always well-defined four well-formedness constraints are required. The first three are: (WA-CON) **is** edges are in J' only allowed between nodes in J and nodes not in J, (WA-NEC) if a node n in J' is not in J and participates in an **is** edge in J' then no attribute edge leaves from n, and (WA-NMC) every node in J' that is not in J is involved in at most one **is**

[2] The notion of *sub-graph* is defined as usual for labeled graphs except that nodes in a subgraph may be labeled with a subset of the corresponding set in the super-graph.

edge in J'. These constraints prevent that the addition has to merge composite values in order to satisfy **is** edges in J'. The reason that we want to prevent this is that there is not always a unique minimal extension of a weak instance graph that makes two composite-value nodes represent the same composite value.

The reason for the fourth and final well-formedness constraint is illustrated by the addition in Fig. 7. If we apply the addition (a) to the instance graph (b) as indicated by the nodes and edges drawn with solid lines, then we should extend it as indicated by the edges and nodes drawn with dotted lines, but this defines a composite value that contains itself which is not allowed.

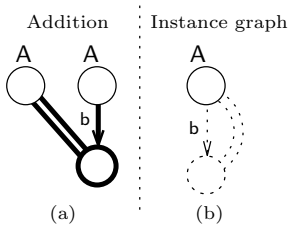

Fig. 7. An addition and its hypothetical result

To prevent such cycles we define the notion of *maximally merged version of an addition* $\texttt{Add}(J, J')$ which is constructed by merging the composite-value nodes and object nodes in J if they have the same sort and the result is still an addition until no more nodes can be merged. The fourth well-formedness constraint then says that (WA-NRI) in the maximally merged version of the addition every cycle that contains at least one attribute edge also contains at least one object node.

Summarizing, an addition that satisfies WA-CON, WA-NEC, WA-NMC and WA-NRI is called a *well-formed* addition. It can be shown that the result of a well-formed addition is always well-defined since it can be constructed by first performing the addition without the **is** edges in the extension pattern, and then extending the data graph to satisfy the **is** edges in the extension pattern by copying for every **is** edge the sub-tree under the old node to the new node.

3.3 The Deletion

A *deletion* is specified by giving two patterns; a *base pattern* J and a *core pattern* J' that is, except for the **is** edges, a sub-graph of the base pattern. Both J and J' are patterns except that in J' we allow class-free nodes that are not reachable from a class-labeled node. We will write such a deletion as $\texttt{Del}(J, J')$ where J is the base pattern and J' is the core pattern. An example of an addition is shown in Fig. 8.

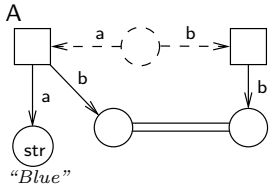

Fig. 8. A deletion

The nodes, edges, and class names of the core pattern are drawn with normal lines and written in a normal font, whereas the nodes, edges and class names that are not in the core pattern are drawn with dashed lines and written in an outline font.

The result of applying a deletion $\texttt{Del}(J, J')$ to a well-formed data graph I, written as $[\![\texttt{Del}(J, J')]\!](I)$, is defined as the well-formed data graph I' where I' is the maximal well-formed sub-graph of I such that for every embedding h of J in I it holds that (1) if $n \in N_J - N_{J'}$ then $h(n) \notin N_{J'}$, (2) if $\langle n_1, \alpha, n_2 \rangle \in E_J - E_{J'}$ then $\langle h(n_1), \alpha, h(n_2) \rangle \notin E_{J'}$, and (3) if for a node n in J there is a $c \in \lambda_J(n) - \lambda_{J'}(n)$ then $c \notin \lambda_{J'}(h(n))$.

Note that I' can be constructed by first removing from I for every embedding h of J in I the nodes, edges and class names not in J' and after that removing all class-free nodes that are no longer reachable from a class-labeled node.

3.4 The Reduction

The third and final operation of GUL is the *reduction* that transforms well-formed data graphs into instance graphs. It does this by merging two basic-value nodes if they are labeled with the same basic-value representation and merging two composite-value nodes if they are value equivalent and labeled with the same class name or both have an incoming edge from the same node, until no more nodes can be merged.

4 Typing GUL

4.1 Typing Patterns

A pattern J is said to be *sound under a schema graph* S if there is a well-formed data graph I that belongs to S and there is an embedding of J in I. To detect such patterns we introduce a syntactical notion of well-typedness. The first case we consider is patterns with no **is** edges and schema graphs with no implicit object class nodes.

An extension relation ξ from S to I is said to be *minimal on the composed-value nodes* if there is no strict subset of ξ that is also an extension relation from S to I but is identical to ξ on the object nodes and the basic-value nodes.

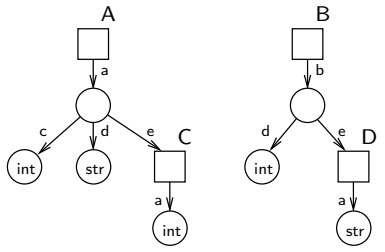

Fig. 9. A schema graph with no implicit object class nodes

Definition 1. *Given a schema graph S with no implicit object class nodes a pattern J without* **is** *edges is said to be* well-typed under *S if there is an extension relation from S to J that* supports *J, i.e., that is minimal on the composed-value nodes and covers J.*

Theorem 1. *Given a schema graph S with no implicit object class nodes a pattern J without* **is** *edges is sound under S iff it is well-typed under S.*[3]

[3] Proofs are omitted because of lack of space but are given in [21]

Theorem 2. *Deciding well-typedness of a pattern with no* **is** *edges under a schema graph with no implicit object class nodes is in PTIME.*

The next problem we consider is typing patterns with **is** edges. The introduction of **is** edges in patterns introduces extra typing problems. Consider, for example, the schema graph in Fig. 9 and the three patterns in Fig. 10. The problem in pattern (a) is that the **is** edge implies that there is a c edge under the b edge but this edge will not be covered in the schema graph in Fig. 9. The problem in pattern (b) is that the implied d edge ends in a node with the wrong sort. Finally, the problem in pattern (c) is that the implied e edge requires the C node to be in the class D and it should therefore be labeled with D, which may not be the case.

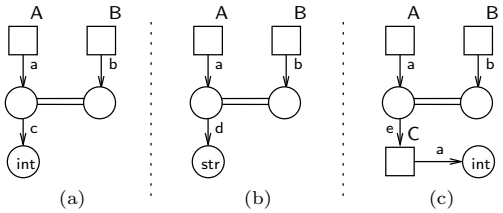

Fig. 10. Three patterns with **is** edges

To remedy this we have to check if these problems occur for all *value paths* in a pattern, where a value path is defined as a path of attribute, **is** and **isa** edges that contains only edges that start in composite-value nodes. If the list of attribute names that consecutively appear in such paths are the same then such paths are said to be *similar*. This leads to the following definition of well-typedness.

Definition 2. *Given a schema graph S with no implied object classes a pattern J is said to be* well-typed *under S if there is an extension relation ξ from S to J that* supports *J, i.e., it is minimal on the composite-value nodes, covers J without the* **is** *edges and for every composite-value node n in J it holds for every value path in J that starts in n that (TP-CVV) if the path contains at least one attribute edge then there is a similar value path in S starting in a node m such that $\xi(m, n)$, (TP-CSV) for every similar value path in S that starts in a node m such that $\xi(m, n)$ it holds that these paths end in nodes with the same sort, and (TP-OCV) if the path ends in an object node n' then it holds for every similar value path in S that begins in m such that $\xi(m, n)$ and ends in m' that $\xi(m', n')$.*

Theorem 3. *Given a schema graph S with no implicit object class nodes a pattern J is sound under S iff it is well-typed under S.*

Theorem 4. *Deciding well-typedness of a pattern under a schema graph with no implied object class nodes is co-NP complete.*

4.2 Typing Additions

A well-formed addition $\texttt{Add}(J, J')$ is said to *respect a schema graph S* if for every well-formed data graph I that belongs to S it holds that $[\![\texttt{Add}(J, J')]\!](I)$ also belongs to S.

The well typedness of a well-formed addition $\texttt{Add}(J, J')$ under a schema graph S will be defined by considering all extension relations from S to J that support J and then extending these minimally to J'. Obviously it should hold for this minimal extension that it covers J' and is sort correct.

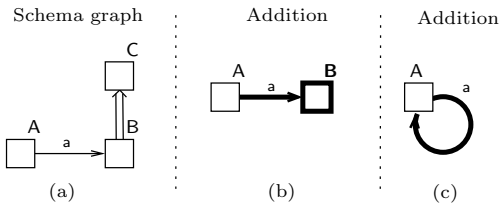

Fig. 11. A schema graph and two additions that add new nodes and edges

Two more constraints that should be checked are illustrated by Fig. 11. The addition (b) does not respect the schema graph (a) because the new node should also be labeled with C. So it should also be checked if the minimal extension places new nodes only in explicit classes that they are not labeled with. The addition (c) also not respects the schema graph for the same reason except here it is an old node that is placed in an extra explicit class that it is not already labeled with.

Two other constraints that should also be checked for the minimal extension of the supporting extension relation are illustrated by Fig. 12. The addition (b) does not respect the schema graph (a) because the A node might have an a edge ending in a int node. The int node would however conflict with the str class node at the end of the a edge from the B class node. The addition (c) also does not respect the schema graph (a) because here the A node might have an a edge ending in a class-free node. However, after the addition of the C label this node should be labeled with the D label, which it is not.

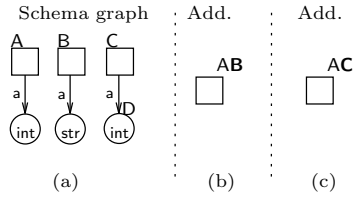

Fig. 12. A schema graph and two additions that add class names to old nodes

In order to prevent the previous two conflicts we introduce some new notions. A *weak value path* is a path of edges such that all inner nodes are composite-value nodes. Given a well-formed data graph I, a schema graph S and an extension relation ξ from S to I a *schema path for a node* $n \in N_I$ *under* ξ is a path in S that starts in a node m such that $\xi(m, n)$. Such a path is said to be a *potential path* if for all prefixes p' of the path there is not a path p'' from node m' in S such that $\xi(m', n)$ and p' and p'' end in nodes with different sorts. It is now easy to see that the problems demonstrated in Fig. 12 can be detected by checking for all potential weak value paths if they cause sort problems or class-name problems at the ends of these paths.

Definition 3. *Given a schema graph S with no implied object class nodes a well-formed addition* $\texttt{Add}(J, J')$ *with no* **is** *edges in J' is said to be* well-typed under S *if for every extension relation from S to J that supports J the minimal superset ξ'*

of ξ that is an extension relation from S to J' is sort correct, covers $I_{J'}$ and (TA-NCN) for every node n in J' that is not in J it holds that if $\xi'(m,n)$ and $\lambda_S(m)$ is defined then $\lambda_S(m) \in \lambda_{J'}(n)$, and for every node n in J it holds that (TA-NCO) if $\xi'(m,n)$ and not $\xi(m,n)$ and $\lambda_S(m)$ is defined then $\lambda_S(m) \in \lambda_{J'}(n)$, (TA-CPW) every potential weak value path in S for n under ξ is also a potential weak value path for n under ξ', and (TA-NPW) for every potential weak value path in S for n under ξ and a similar path in S for n under ξ' that ends in an explicit class node m'_2 there is a similar path in S for n under ξ that ends in m'_2.

Theorem 5. *Given a schema graph S with no implied object class nodes a well-formed addition* $\mathtt{Add}(J, J')$ *with no* **is** *edges in J' respects S if it is well-typed under S.*

Unfortunately it is not true the well-typedness is a necessary condition for respecting a schema graph as is shown in Fig. 13.

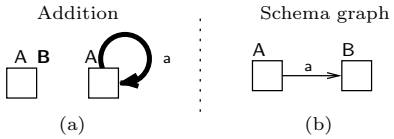

Fig. 13. A well-formed addition that respects a schema graph but is not well-typed

The presence of **is** edges in the extension pattern makes it necessary to check extra constraints. This is illustrated in Fig. 14. The addition (b) does not respect the schema graph (a) because the potential a attribute is copied to the new B node but according to the schema graph B nodes can only have b attributes. The addition (b) does not have this problem but here the sort of the potential a attribute under C is different from that of the a attribute under A. Finally the addition (c) does not respect (a) because the potential a attribute is forced into a new explicit class E with which it is not yet labeled.

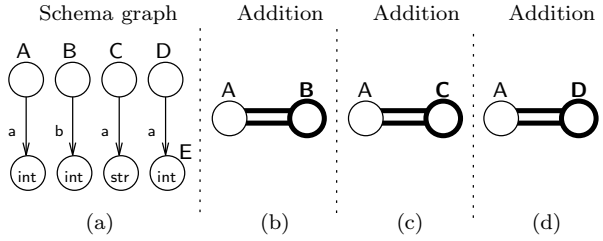

Fig. 14. A schema graph with three additions with **is** edges in the extension pattern

These considerations lead to the following extension of the definition of well-typedness.

Definition 4. *Given a schema graph S with no implied object class nodes a well-formed addition* $\mathtt{Add}(J, J')$ *is said to be* well-typed under *S if it satisfies*

the conditions for an addition without **is** *edges in J' and it holds for ξ and ξ' that for every* **is** *edge $\langle n_1, \mathbf{is}, n_2\rangle$ in J' with n_1 in J it holds that (TA-CPI) for every potential weak value path in S for n_1 under ξ there is a similar potential weak value path in S for n_2 under ξ' that ends in a node with the same sort, and (TA-NPI) for every potential weak value path in S for n_1 under ξ and a similar path in S for n_2 under ξ' that ends in an explicit class node m_2' there is a similar path in S for n_1 under ξ that ends in m_2'.*

Theorem 6. *Given a schema graph S with no implied object class nodes a well-formed addition $\mathtt{Add}(J, J')$ respects S if it is well-typed under S.*

Theorem 7. *Deciding well-typedness of a well-formed addition under a schema graph with no implied object class nodes is in PSPACE.*

Theorem 8. *Deciding if a well-formed addition respects a schema graph with no implied object class nodes is PSPACE hard.*

Theorem 9. *Deciding well-typedness of an addition under a schema graph with no implied object class nodes and no composite-value classes is in PTIME.*

4.3 Typing Deletions

Given a schema graph S with no implied object class nodes a deletion $\mathtt{Del}(J, J')$ is said to *respect* S if for all well-formed data graphs I that belong to S it holds that $[\![\mathtt{Add}(J, J')]\!](I)$ also belongs to S.

In order to understand what needs to be checked for well-typedness consider the deletion (a) in Fig. 15 and schema graph (b) in the same figure. In instance graph (a) the problem is that the class name that is deleted is required to be there since the node is still labeled with the name C of a subclass. In instance graph (b) the problem is that after the deletion the b edge is no longer covered by the schema graph. Finally, in instance graph (e) has the problem that after the deletion the c edge forces the node back into the B class.

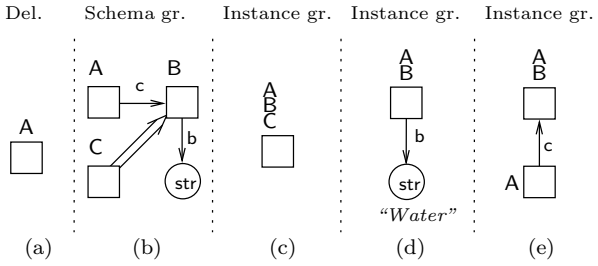

Fig. 15. A deletion, a schema graph and three instance graphs that demonstrate potential class name deletion problems

To deal with these problems we have to consider two sets of class nodes: the set of nodes that an instance graph was associated with and the set of nodes

that it is no longer associated with because the class names of these nodes are removed. For this purpose we introduce the notion of *basic deletion pair* which is defined for a deletion $\texttt{Del}(J, J')$ and a schema graph S with no implied object class nodes, as the set of pairs $\langle M_1, M_2 \rangle \in \mathcal{P}(N_S) \times \mathcal{P}(N_S)$ such that there is an extension relation $\bar{\xi}$ from S to J that supports J and a node $n \in J$ such that $M_1 = \{\ m \in N_S \mid \bar{\xi}(m, n)\ \}$ and $M_2 = \{\ m \in N_S \mid \lambda_S(m) \in \lambda_J(n) - \lambda_{J'}(n)\ \}$.

Because embeddings are not injective and different embeddings can embed differently upon the same node, it is possible that what is removed from a node is a combination of basic deletion pairs. Therefore we define the set of deletion pairs as the smallest superset of the basic deletion pair that satisfies the rule that if $\langle M_1, M_2 \rangle$ and $\langle M_1, N_3 \rangle$ are deletion pairs then $\langle M_1, M_2 \cup N_3 \rangle$ is also a deletion pair.

With the help of this set we can now define a notion of well-typedness that prevents the three problems that were demonstrated in Fig. 15.

Definition 5. *Given a schema graph S with no implied object class nodes a deletion $\texttt{Del}(J, J')$ is said to be* well-typed under *S if for every deletion pair $\langle M_1, M_2 \rangle$ with no composite-value class nodes in M_1 it holds that (TD-NSC) there are not two class nodes m_1 and m_2 in S such that $m_1 \in M_1 - M_2$, $m_2 \in M_2$ and m_1 $\mathbf{isa}_S^*$ m_2, (TD-EC) there is not a potential weak value path p in S from $\{m_1\}$ where m_1 is an explicit class node m_1 and the end node m_2 of p such that $m_2 \in M_2$ and for all similar weak value paths in S from m_1 to m_2' it holds that $m_2' \in M_1$, and (TD-CPE) for every potential weak value path in S from M_1 there is similar weak value path in S from $M_1 - M_2$.*

Theorem 10. *Given a schema graph S with no implied object class nodes a deletion $\texttt{Del}(J, J')$ respects S if it is well-typed under S.*

Unfortunately it is not true the well-typedness is a necessary condition for respecting a schema graph as is shown in Fig. 16.

Theorem 11. *Deciding well-typedness of an addition under a schema graph with no implied object class nodes is in PSPACE.*

Theorem 12. *Deciding if a deletion respects a schema graph with no implied object class nodes is PSPACE hard.*

4.4 Typing the Reduction

Typing the reduction is trivial because if a well-formed data graph belongs to a schema graph then the reduction will also belong to that schema graph. This can be understood if we look at one step of the reduction where two nodes are merged. It is easy to see that the result of merging these two nodes will belong to a schema graph iff the original well-formed instance graph belongs to it. It follows that the result of the reduction belongs to a schema graph iff the original well-formed instance graph does.

5 Conclusion

In this paper we introduced a graph-based data model GDM that represents instances and schemas as labeled graphs, and incorporates features from object-oriented data model, ER data models and semistructured data models. Together with this data model we introduced a graph-based update language GUL that is based on pattern matching and allows the user to specify additions and deletions in a graphical way.

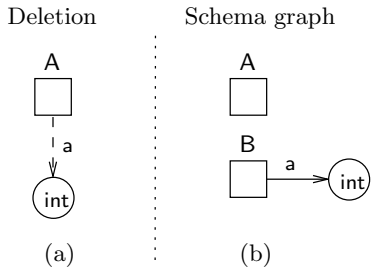

Fig. 16. A deletion that is not well-typed but respects the schema graph

For patterns a notion of well-typedness was introduced that captures exactly when there is a well-formed data graph that belongs to the schema graph and the pattern embeds into this data graph. For patterns without **is** edges this notion can be checked in PTIME and with **is** edges it was shown to be co-NP complete.

For additions a notion of well-typedness was introduced that is a sufficient condition, but not necessary condition, for an addition to respect a schema graph. Deciding this notion of well-typedness was shown to be PSPACE complete. In the special case where there are no composite-value nodes deciding well-typedness can be done in PTIME.

For the deletion also a notion of well-typedness was introduced that is a sufficient condition, but not a necessary condition, for a deletion to respect a schema graph. Deciding this notion of well-typedness was also shown to be PSPACE complete.

Finally the reduction operation that reduces all well-formed data graphs to instance graphs, was shown to be trivially well typed.

References

1. Chen, P.P.: The Entity-Relationship Model: Toward a unified view of data. ACM Transactions on Database Systems **1** (1976) 9–36
2. Kuper, G.M., Vardi, M.Y.: The logical data model. ACM Transactions on Database Systems **18** (1993) 379–413
3. Abiteboul, S., Hull, R.: IFO: A formal semantic database model. ACM Transactions on Database Systems **12** (1987) 525–565
4. Shipman, D.W.: The functional data model and the data language DAPLEX. ACM Transactions on Database Systems **6** (1981) 140–173
5. Gyssens, M., Paredaens, J., Van den Bussche, J., Van Gucht, D.: A graph-oriented object database model. IEEE Transactions on Knowledge and Data Enginering **6** (1994) 572–586

6. Catarci, T., Tarantino, L.: A hypergraph-based framework for visual interaction with databases. Journal of Visual Languages and Computing **6** (1995) 135–166
7. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. In: Foundations of Software Science and Computation Structure. (2000) 98–113
8. Consens, M.P., Eigler, F.C., Hasan, M.Z., Mendelzon, A.O., Noik, E.G., Ryman, A.G., Vista, D.: Architecture and applications of the Hy^+ visualization system. IBM Systems Journal **33** (1994) 458–476
9. Poulovassilis, A., Hild, S.G.: Hyperlog: a graph-based system for database browsing, querying and update. IEEE Data & Knowledge Engineering **13** (2001) 316–333
10. Abiteboul, S.: Querying semi-structured data. In: ICDT. (1997) 1–18
11. Suciu, D.: An overview of semistructured data. SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory) **29** (1998)
12. Van den Bussche, J., Van Gucht, D., Andries, M., Gyssens, M.: On the completeness of object-creating database transformation languages. Journal of the ACM **44** (1997) 272–319 A revised and extended version of [22].
13. Gemis, M., Paredaens, J.: An object-oriented pattern matching language. In: Proceedings of the First JSSST International Symposium. Number 742 in LNCS, Springer-Verlag (1993) 339–355
14. Gemis, M.: Graph-based languages in DBMS. PhD thesis, University of Antwerp (1996)
15. Hidders, J., Paredaens, J.: GOAL: A graph-based object and association language. In Paredaens, J., Tenenbaum, L., eds.: Advances in Database Systems - Implementations and Applications. Volume 347 of CISM Courses and Lectures. Springer-Verlag (1994) 247–265
16. Beeri, C.: A formal approach to object-oriented databases. Data and Knowledge Engineering **5** (1990) 353–382
17. Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM **13** (1970) 377–387
18. Jaeschke, G., Schek, H.J.: Remarks on the algebra of non first normal form relations. In: Proc. of the 1st ACM Symp. on Principles of Database Systems, Los Angeles, California (1982) 124–138
19. Hull, R.B., Yap, C.K.: The format model: A theory of database organization. Journal of the ACM **31** (1984) 518–537
20. Nijssen, G.M., Halpin, T.: Conceptual Schema and Relational Database Design: a fact oriented approach. Prentice Hall, Sydney, Australia (1989)
21. Hidders, J.: GUL, a Graph-based Update Language for Object-Oriented Data Models. PhD thesis, Eindhoven University of Technology (2001)
22. Van den Bussche, J., Van Gucht, D., Andries, M., Gyssens, M.: On the completeness of object-creating query languages. In: Proc. of the 33rd Symposium on Foundations of Computer Science, IEEE Computer Society Press (1992) 372–379