A Directed Recursive Hypergraph Data Model for Representing the Contents of Multimedia Data

Teruhisa HOCHIN* and Tatsuo TSUJI*

(Received Aug. 28, 2000)

This paper proposes a data model incorporating the concepts of directed graphs, recursive graphs, and hypergraphs in order to represent the contents of multimedia data. In the proposed data model, an instance is represented with a directed recursive hypergraph. This graph is called an instance graph. A collection of instance graphs is managed as a graph named a collection graph. A shape graph, which represents the structure of a collection graph, is also introduced. An operation rewriting collection graphs is introduced to manipulate the collection graphs. This operation enables users to make recursive queries, and specify regular expressions on paths. This paper presents an illustrative example, and the formal definition of the proposed data model. This paper clarifies that the depth of the edges of an instance graph can indicate whether the instance graph can be divided, or not. The consideration on the management of shape graphs is also presented. Moreover, it is clarified that the operation can be converted into the datalog program extended for treating complex values.

Key Words: Data Model, Directed Graph, Hypergraph, Recursive Graph

1 Introduction

In recent years, handling multimedia data stored in databases is extensively investigated. Content retrieval of multimedia data is included in the topics on handling multimedia data. One approach to address the content retrieval uses the feature values of multimedia data[3, 4]. For example, when a picture is given as a desired one, similarity between the picture and one in a database is calculated by using the feature values. The picture having the high score of similarity is presented to the user as the query result. Another approach uses the graphs representing the contents of multimedia data. Petrakis et al[1] propose the representation of the contents of medical images by using directed labeled graphs. Uehara et al[2] uses the semantic network for representing the contents of the scene of video. These are the examples of this approach. Although directed labeled graphs are frequently used in these researches, conceptual graphs[5], which are used for the knowledge representation, require to be the recursive graphs, of which nodes may recursively be graphs. Moreover, hypergraphs, where a
set of nodes is treated as an edge, are used in representing rules[6]. Therefore, the graphs having the characteristics of recursive graphs and hypergraphs as well as directed ones will be required in the representation of the contents of multimedia data.

If graphs are simply directed ones, they are naturally represented in object-oriented, or graph-based data models[7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20, 21]. However, graphs having the characteristics of recursive graphs or hypergraphs cannot be naturally represented in these data models.

Hypernode model[22] and Hy+[23] incorporate the concepts of recursive graphs. In these models, regular expressions on paths can be specified in the retrieval. Recursive queries also can be made. These query facilities give these data models the sufficient querying power. However, graphs having the characteristics of hypergraphs cannot be naturally represented.

The directed recursive labelnode hypergraph model[24] and the self-structured semantic relationship model[25] incorporate the concepts of recursive graphs as well as hypergraphs. These models have more expressive power in representing the contents of multimedia data. However, the operations in querying is not sufficient to users because the operations are mainly for updating the structures of graphs. Recursive queries are not considered. Regular expressions on paths can not be specified.

In this paper, a graph-based data model is proposed. The proposed data model is called the Directed Recursive Hypergraph data Model (DRHM). This model incorporates the concepts of directed graphs, recursive graphs, and hypergraphs. An instance graph is the fundamental unit in representing data. A collection graph is the graph having instance graphs as its components. A shape graph of a collection graph represents the structure of a collection graph. The rewrite operation is a logic-based one. Recursive queries can be specified through this operation. It enables users to specify the regular expressions on paths. It is used in updating instance graphs as well as querying on a database. After the formal definition is presented, several considerations are made. First, the characteristics of an instance graph is investigated. The depth of an instance graph is introduced in order to identify whether it can be decomposed into sub-instance graphs. When the depth of an instance graph is equal to zero, the instance graph can be decomposed into sub-instance graphs. Decomposing instance graphs could make their treatment easy. Next, the method of managing shape graphs is presented. The information on the shape graphs is represented with the instance graphs. Lastly, it is shown that the rewrite operation can be represented with the datalog program extended for treating complex values.

This paper is organized as follows: In Section 2, the proposed data model is informally described by using examples. Section 3 defines the Directed Recursive Hypergraph Data Model. Section 4 gives the considerations on the characteristics of an instance graph, the shape graph management, and the querying power. In Section 5, the related works are briefly mentioned. Lastly, Section 6 concludes this paper.

2 Descriptive Examples

In DRHM, the fundamental unit in representing data is an instance graph. An instance graph is an atomic one or a normal one. An atomic instance graph is a node. A normal one is a directed recursive hypergraph. An instance graph has a label composed of its identifier, its name, and its data value.

Here, DRHM is described by using a simple example.

Example 1 Consider the representation of the picture shown in Figure 1(a). In this picture, a
A set of the instance graphs having the similar structure is captured as a *collection graph*. A *collection graph* is the graph of which components are instance graphs.

**Example 2** An example of a collection graph is shown in Fig. 2. In this figure, a collection graph is represented with a dashed dotted line. A collection graph has a unique name in a database. The name of the collection graph shown in Fig. 2 is *Picture*. The instance graph *g1* is that shown in Fig. 1. The instance graph *g2* is for another picture. These instance graphs are called *representative instance graphs*.

The structure of a collection graph is represented with the graph called a *shape graph*.

**Example 3** Figure 3 shows the shape graph for the collection graph *Picture*. This shape graph represents the following structures. The instance graph *picture* includes a normal instance graph *object*. A normal instance graph *object* is connected with a normal instance graph *object* by the edge *pos*. A normal instance graph *object* contains an atomic instance graph *parts*. An atomic instance graph *parts* is connected with an atomic instance graph *parts* by the edge *pos* inside of the
object or outside it.

The shape graph has the nature of the hard shape[27]. That is, a shape graph does not have to exist prior to the creation of the collection graph. It may, of course, exist prior to the collection graph creation. The shape graph must exist while the collection graph exists. Creating the shape graph and updating it are driven by the insertion of instance graphs. However, once shape graphs are created, they are not deleted by deleting the instance graphs. Shape graphs can be deleted only by the operation deleting shape graphs.

The operation rewrite is a general and logic-based one. This operation rewrites collection graphs. This uses query graphs. One is for representing the structure of the destination instance graphs. The
others are for specifying the retrieval condition. The structure of a query graph may be similar to that of a collection graph. The label of a query graph is a triplet of variables for an identifier, a name, and a value. The result of this operation is a collection graph.

**Example 4** An example of the rewrite operation is shown in Fig. 4. This is for obtaining such pictures that the parts included by an object is connected with the parts included by an object through an edge of which name is on. The first argument is a destination query graph. The second argument is a query graph specifying the retrieval condition. The instance graphs, which are in the collection graph Picture, satisfying the query graph become the instance graphs in the collection graph My-picture. The labels of query graphs, e.g. X, Z1, may used in a retrieval condition. Variables for an identifier, a name, and a value of a label X are described as Xid, Xname, and Xval, respectively, in Fig. 4.

```
rewrite('My-picture')
```

```
Xname = "picture" and
Yname = "pos" and Yval = "on" and
Z1name = "object" and Z2name = "object" and
V1name = "parts" and V2name = "parts"
```

Figure 4: An example of the rewrite operation

In a query graph, the regular expression can be specified.

**Example 5** An example of the rewrite operation including regular expression is shown in Fig. 5. This is for obtaining such Pictures that the parts included by an object is connected with the parts included by an object through two or more edges of which names are on. Brackets are used in order to represent the regular expression. The plus mark denotes that the part in the brackets may appear one or more times.

The rewrite operation can also be used for inserting instance graphs into a collection graph. The rewrite operation only having the destination query graph means insertion.

**Example 6** An example of the insertion of an instance graph is shown in Fig. 6. In this figure, an instance graph is inserted into the instance graph Picture. Names and values are specified by using variables.

```
rewrite('Picture')
```

```
Xname = "picture" and
Yname = "pos" and Yval = "on" and
Z1name = "object" and Z2name = "object" and
V1name = "parts" and V2name = "parts"
```

In the insertion, identifiers are assigned by the system. If the variable for an identifier is used in a destination query graph in order to specify a specific instance graph or edge, this means modification of the instance graph or edge.
3 Formal Definition

In this section, DRHM is defined. First, the structure in representing data is defined. Next, an operation is defined.

3.1 Structure

3.1.1 Instance, collection, and shape graphs

Definition 1 *Instance graphs* are defined as follows.

- An instance graph is an atomic instance graph or a normal one. An instance graph has a label. A label is a triplet \((d_{id}, n_{rn}, d)\), where \(d_{id}\) is an identifier, \(n_{rn}\) is a name, and \(d\) is a tuple of a data type and a data value.
- An *atomic instance graph* is a node.
- A *normal instance graph* \(g\) is an octuple \((V, E, L_v, L_e, \phi_v, \phi_e, \phi_{connect}, \phi_{comp})\), where \(V\) is the set of the instance graphs included in \(g\), \(E\) is the set of the edges, \(L_v\) and \(L_e\) are the sets of the labels of the instance graphs, and the edges, respectively, \(\phi_v\) is a mapping from the set of the instance graphs to the set of the labels of the instance graphs: \(V \rightarrow L_v\), \(\phi_e\) is a mapping from the set of the edges to the set of the labels of the edges: \(E \rightarrow L_e\), \(\phi_{connect}\) is a mapping representing

![Figure 5: Specification of regular expression.](image)

![Figure 6: Insertion of an instance graph.](image)
the connections between sets of instance graphs $\phi_{\text{connect}} : 2^V \times 2^V \to E$, and $\phi_{\text{comp}}$ is a mapping representing the inclusion relationships $\phi_{\text{comp}} : 2^{V \cup E} \to V$.

From here on, we use the notations shown in Table 1.

<table>
<thead>
<tr>
<th>notation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i(e)$</td>
<td>the set of initial elements of an edge $e$</td>
</tr>
<tr>
<td>$t(e)$</td>
<td>the set of terminal elements of an edge $e$</td>
</tr>
<tr>
<td>$V(g)$</td>
<td>the set of instance graphs included in an instance graph $g$</td>
</tr>
<tr>
<td>$E(g)$</td>
<td>the set of edges included in an instance graph $g$</td>
</tr>
<tr>
<td>$id(l)$</td>
<td>the identifier in the label $l$</td>
</tr>
<tr>
<td>$nm(l)$</td>
<td>the name in the label $l$</td>
</tr>
<tr>
<td>$val(l)$</td>
<td>the value in the label $l$</td>
</tr>
</tbody>
</table>

Next, we consider the equalities of instance graphs.

**Definition 2** Two labels $l_1$ and $l_2$ are identical ($l_1 = id l_2$) if and only if $id(l_1) = id(l_2)$. These labels are equal ($l_1 = l_2$) if and only if $nm(l_1) = nm(l_2)$ and $val(l_1) = val(l_2)$. These are value-equial ($l_1 = val l_2$) if $val(l_1) = val(l_2)$. Sets of labels $L_1$ and $L_2$ are identical ($L_1 = id L_2$) if and only if $\forall l_1 \in L_1 \exists l_2 \in L_2 (l_1 = l_2)$ and $\forall l_2 \in L_2 \exists l_1 \in L_1 (l_2 = l_1)$. $L_1$ and $L_2$ are equal ($L_1 = L_2$) if and only if $\forall l_1 \in L_1 \exists l_2 \in L_2 (l_1 = l_2)$ and $\forall l_2 \in L_2 \exists l_1 \in L_1 (l_2 = l_1)$.

Two instance graphs $g_1 = (V_1, E_1, L_{v1}, L_{e1}, \phi_{v1}, \phi_{e1}, \phi_{\text{connect}}, \phi_{\text{comp}})$, and $g_2 = (V_2, E_2, L_{v2}, L_{e2}, \phi_{v2}, \phi_{e2}, \phi_{\text{connect}}, \phi_{\text{comp}})$ are identical ($g_1 \equiv g_2$) if and only if $l_1 = id l_2$, where $l_1 = \phi_{v1}(g_1)$, and $l_2 = \phi_{v2}(g_2)$. These instance graphs are equal ($g_1 = g_2$) if and only if $V_1 = V_2, E_1 = E_2, L_{v1} = L_{v2}, L_{e1} = L_{e2}, \phi_{\text{connect}} = \phi_{\text{connect}},$ and $\phi_{\text{comp}} = \phi_{\text{comp}}$.

Next, direct constructing elements of an instance graph are defined.

**Definition 3** Let $g = (V, E, L_v, L_e, \phi_v, \phi_e, \phi_{\text{connect}}, \phi_{\text{comp}})$ be an instance graph. For $G_i \subseteq 2^{V \cup E}$, $v$ is directly consisted of $G_i$ if $\phi_{\text{comp}}(G_i) = v$. $G_i$ is called a set of direct constructing elements, and is denoted as $\phi_{\text{comp}}^{-1}(v)$. A set of instance graphs in a set of direct constructing elements of $v$ ($\phi_{\text{comp}}^{-1}(v) \cap V$) is called as a set of direct constructing instance graphs of $v$. Similarly, a set of edges in a set of direct constructing elements of $v$ ($\phi_{\text{comp}}^{-1}(v) \cap E$) is called as a set of direct constructing edges of $v$.

Let a set of the $n$th constructing elements $V_{ce}(g)$ for an instance graph $g$ be defined by the following recursion formula:
where $DC(v)$ is the set of the direct constructing elements of $v$. Here, $n$ is called the order of the constructing elements. A set of instance graphs (edges, respectively) in a set of $n$th constructing elements of $g$ is called as a set of $n$th constructing instance graphs (edges) of $g$.

$\bigcup_{n=1}^{\infty} V_{ce}^n(g)$ is called a set of constructing elements of $g$. A set of instance graphs (edges, respectively) in a set of the constructing elements of $g$ is called as a set of constructing instance graphs (edges) of $g$.

**Example 7** The direct constructing elements of the instance graph $g_1$ shown in Fig. 1 are $g_{11}$, $g_{12}$, $e_{13}$, and $e_{14}$. The direct constructing instance graphs are $g_{11}$ and $g_{12}$. The direct constructing edges of $g_1$ are $e_{11}$ and $e_{12}$.

Next, *collection graphs* are introduced to capture a set of the instance graphs.

**Definition 4** A *collection graph* is a graph having instance graphs as its components. That is, when a collection graph $cg$ has $n$ instance graphs: $g_1 = (V_1, E_1, L_{v1}, L_{e1}, \phi_{v1}, \phi_{e1}, \phi_{connect1}, \phi_{comp1}), \ldots, g_n = (V_n, E_n, L_{vn}, L_{en}, \phi_{vn}, \phi_{en}, \phi_{connectn}, \phi_{compn})$, $cg$ is represented with a 9-tuple $(nm, V, E, L_v, L_e, \phi_v, \phi_e, \phi_{connect}, \phi_{comp})$, where $nm$ is the name of a collection graph, $V = V_1 \cup V_2 \cup \cdots \cup V_n$, $E = E_1 \cup E_2 \cup \cdots \cup E_n$, $L_v = L_{v1} \cup L_{v2} \cup \cdots \cup L_{vn}$, $L_e = L_{e1} \cup L_{e2} \cup \cdots \cup L_{en}$, $\phi_v : V \rightarrow L_v$, $\phi_e : E \rightarrow L_e$, $\phi_{connect} : 2^V \times 2^V \rightarrow E$, and $\phi_{comp} : 2^{V \cup E} \rightarrow V$. Component instance graphs $(g_1, g_2, \ldots, g_n)$ are called representative instance graphs. A database is a set of collection graphs. The name of a collection graph must be unique in a database.

A *shape graph* is introduced to represent the structure of the instance graphs of a collection graph.

**Definition 5** *Shape graphs* are defined as follows.

- A shape graph is an atomic shape graph or a normal one. A shape graph has a shape label. A shape label is a triplet $(s_{id}, nm, DT)$, where $s_{id}$ is an identifier, $nm$ is a name of a shape graph, and $DT$ is a set of data types of data values.
- An *atomic shape graph* is a node.
- A normal shape graph $sg$ of a collection graph $(nm, V, E, L_v, L_e, \phi_v, \phi_e, \phi_{connect}, \phi_{comp})$ is a 9-tuple $(nm, V_s, E_s, L_{vs}, L_{es}, \phi_{vs}, \phi_{es}, \phi_{connects}, \phi_{comps})$, where $nm$ is the name of the collection graph, $V_s$ is the set of the shape graphs included in $sg$, $E_s$ is the set of the shape edges, $L_{vs}$ is the sets of the labels of the shape graphs, $L_{es}$ is the sets of the labels of the shape edges, which are the edges having shape labels, $\phi_{vs} : V_s \rightarrow L_{vs}$, $\phi_{es} : E_s \rightarrow L_{es}$, $\phi_{connects} : 2^{V_s} \times 2^{V_s} \rightarrow E_s$, and $\phi_{comps} : 2^{V_s \cup E_s} \rightarrow V_s$. Moreover, there are the following relationships between a collection graph and its corresponding shape graph.
  - $nm(L_{vs}) \supseteq nm(L_v)$,
  - $nm(L_{es}) \supseteq nm(L_e)$,
  - There is a mapping $\theta_v$ from $V$ to $V_s$. For a set of instance graphs $V = \{v_1, \ldots, v_n\}$, a set of shape graphs $\{\theta_v(v_1), \ldots, \theta_v(v_n)\}$ is denoted as $\theta_v(V)$. 

$V^0_{ce}(g) = \{g\}$

$V^{i+1}_{ce}(g) = \bigcup DC(v), v \in V^i_{ce}(g) \cap V$
There is a mapping $\theta_e$ from $E$ to $E_s$. For a set of edges $E = \{e_1, \ldots, e_n\}$, a set of shape edges $\{\theta_e(e_1), \ldots, \theta_e(e_n)\}$ is denoted as $\theta_e(E)$.

$\phi_{\text{connect}}(V, U) = e \Rightarrow \phi_{\text{connect}}(\theta_e(V), \theta_e(U)) = \theta_e(e)$, and $\phi_{\text{comp}}(V \cup W) = g \Rightarrow \phi_{\text{comp}}(\theta_e(V) \cup \theta_e(W)) = \theta_e(g)$.

A shape graph have to be changed in order to satisfy the conditions described above when new instance graphs are inserted, or instance graphs are modified.

### 3.2 Operation

The rewrite operation is generally used for manipulating collection graphs. The specification of a query has the graphical nature like in the graphical query languages\[10, 11, 26\]. First, query graphs, which can easily correspond to collection graphs, are defined.

**Definition 6** Query graphs are defined as follows.

- A query graph is an atomic query graph or a normal one. A query graph has a query label. A query label is a triplet $(v_{id}, v_{name}, v_{val})$, where $v_{id}$ is a variable for an identifier, $v_{name}$ is a variable for a name, and $v_{val}$ is a variable for a value.

- An atomic query graph is a node.

- A normal query graph $qg$ is a 9-tuple $(nm, V_q, E_q, L_{eq}, \phi_{eq}, \phi_{connectq}, \phi_{compq})$, where $nm$ is the name of a collection graph, $V_q$ is the set of the query graphs included in $qg$, $E_q$ is the set of the query edges, which are the edges having query labels, of $qg$, $L_{eq}$ are the sets of the query labels of the query graphs, and the edges, respectively, $\phi_{eq} : V_q \rightarrow L_{eq}$, $\phi_{connectq} : 2^{V_q} \times 2^{V_q} \rightarrow E_q$, and $\phi_{compq} : 2^{V_q \cup E_q} \rightarrow V_q$. If $nm$ is the name of the existing collection graph, then the structure of the query graph must obey that of the shape graph of the collection graph.

Regular expression is the popular method in specifying the query on a graph-based database\[10, 11, 26\]. First, query paths are introduced for enabling the specification of regular expression.

**Definition 7** A query path is recursively defined as follows.

- A query edge of a query graph is a query path.

- Concatenation of a query edge of a query graph and a query path is a query path.

- $(qp)^+$ is a query path for a query path $qp$.

- $(qp_1 | qp_2)$ is a query path for query paths $qp_1$ and $qp_2$.

- $\neg qp$ is a query path for a query path.

A query path has a label, which is a quintuple of a variable for an identifier, that for a name, that for a value, the sign of the positive closure, and the sign of the negation.

The regular query graph, which introduces query paths into the normal query graph, is defined as follows.

**Definition 8** A regular query graph $qg$ is a 14-tuple $(nm, V_q, E_q, P_q, L_{eq}, L_{pq}, \phi_{eq}, \phi_{compq}, \phi_{connectq}, \phi_{pathconc}, \phi_{pathunion})$, where $P_q$ is a set of the query paths of the query graph $qg$, $L_{pq}$ is the set
of the labels of the query paths, \( \phi_{pq} : P_q \to L_{pq} \). \( \phi_{\text{pathconc}} \) is a mapping representing the concatenations of query paths \( \phi_{\text{pathconc}} : \Pi(E_q \cup P_q) \to P_q \). \( \phi_{\text{pathunion}} \) is a mapping representing the union of query paths \( \phi_{\text{pathunion}} : \Pi P_q \to P_q \), and the others are the same as those of the normal query graph. Here, \( \Pi P = P_1 \times P_2 \times \cdots \times P_n \). □

Next, query and assignment specifications are defined.

**Definition 9** A query specification is a tuple \((rqg, rc)\), where \(rqg\) is a regular query graph, and \(rc\) is a retrieval condition on variables \(X\), which is a list of variables in \(rqg\). An assignment specification is a tuple \((nqg, va)\), where \(nqg\) is a normal query graph, and \(va\) is a list of value assignments to variables \(X\), which is a list of variables in \(nqg\). □

The operation \textit{rewrite} is defined as follows.

**Definition 10** The operation \textit{rewrite}(target, sl) rewrites the collection graphs specified in the query graphs in \(sl\) to produce the collection graph satisfying \(target\), where \(target\) is an assignment specification, \(sl\) is a list of query specifications, and \(X_t \subseteq X_s\), where \(X_t\) and \(X_s\) are the sets of the variables in \(target\), and that of those in \(sl\), respectively. If the name of the collection graph specified in the assignment specification \(target\) is that of an existing collection graph, the rewrite operation rewrites the collection graph. Otherwise, this operation creates the temporal collection graph of which name is that specified in \(target\). □

4 Considerations

4.1 Characteristics of instance graphs

From here on, we limit the discussions on the instance graphs having the edges, each of which has an instance graph as its initial element, and an instance graph as its terminal one because of simplicity. The following discussions can naturally be extended to the general instance graphs.

**Definition 11** Two instance graphs \(g_1\) and \(g_2\) are independent each other, if \(V(g_1) \cap V(g_2) = \emptyset\) and \(E(g_1) \cap E(g_2) = \emptyset\). Two instance graphs \(g_1\) and \(g_2\) are connected, if \(\exists e \in E(g_1) \ (i(e) \subseteq V(g_2) \lor t(e) \subseteq V(g_2))\), or \(\exists e \in E(g_2) \ (i(e) \subseteq V(g_1) \lor t(e) \subseteq V(g_1))\). □

**Example 8** Consider the instance graphs:

\[
g_a = (\{v_1, v_2, g_a\}, \{e_1\}, L_v, L_e, \phi_v, \phi_e, \phi_{\text{connect}_a}, \phi_{\text{comp}_a})
\]

where \(\phi_{\text{connect}_a}(\{v_1\}, \{v_2\}) = e_1\), \(\phi_{\text{comp}_a}(\{v_1, v_2, e_1\}) = g_a\), and \(g_b = (\{v_2, v_3, g_b\}, \{e_2\}, L_v, L_e, \phi_v, \phi_e, \phi_{\text{connect}_b}, \phi_{\text{comp}_b})\), where \(\phi_{\text{connect}_b}(\{v_2\}, \{v_3\}) = e_2\), and \(\phi_{\text{comp}_b}(\{v_2, v_3, e_2\}) = g_b\). These instance graphs are connected because \(t(e_1) = i(e_2) = \{v_2\}\), and the conditions described above are satisfied.

When any elements outside an instance graph are not directly connected with any elements in the instance graph by any edge, the instance graph could be represented separately from the other elements outside it. That is, the instance graph is represented as if it has no elements, and it and its elements are represented apart from the whole instance graph. An instance graph could be decomposed into a set of smaller instance graphs. This may enable us to handle instance graphs easily similar to the

\(^1\)The notation \(i(e)\) (\(t(e)\), respectively) represents the set of the initial (terminal) elements of the edge \(e\) as described in Table 1.
decomposition approaches in storing complex objects [31]. We will say that an instance graph \( g \) is said to be well-organized if any elements outside it are not directly connected with any elements in it by any edge.

**Definition 12** Let \( VC \) be the set of constructing instance graphs of an instance graph \( g \). The instance graph \( g \) is called well-organized if the set of the initial elements of an edge is a subset of \( VC \), and the set of the terminal elements of the edge is a subset of \( VC \) for all of the edges that have an element in \( VC \) as an element in their initial elements or terminal ones.

**Example 9** Let consider an instance graph \( g = (\{v_1, v_2, v_3, v_4, g_1, g_2, g\}, \{e_1, e_2, e_3\}, L_v, L_e, \phi_v, \phi_e, \phi_{conn} \phi_{comp}) \), where \( \phi_{conn}(\{v_1\}, \{v_2\}) = e_1, \phi_{conn}(\{v_3\}, \{v_4\}) = e_2, \phi_{conn}(\{g_1\}, \{v_3\}) = e_3, \phi_{comp}(\{v_1, v_2, e_1\}) = g_1, \phi_{comp}(\{v_3, v_4, e_2\}) = g_2, \phi_{comp}(\{g_1, g_2, e_3\}) = g \). The set of the constructing instance graphs of \( g_1 \) is \( \{v_1, v_2\} \). It is the edge \( e_1 \) that has the instance graphs \( v_1 \) and \( v_2 \) in the set of the initial elements or that of the terminal ones. The set of the initial elements of \( e_1 \) is \( \{v_1\} \), and it is a subset of the set of constructing instance graphs of \( g_1 \) \( \{v_1, v_2\} \). The set of the terminal elements of \( e_1 \) is \( \{v_2\} \), and it is a subset of the set of constructing instance graphs of \( g_1 \) \( \{v_1, v_2\} \). Therefore, \( g_1 \) is well-organized.

On the other hand, The set of the constructing instance graphs of \( g_2 \) is \( \{v_3, v_4\} \). The edges that has the instance graphs \( v_3 \) and \( v_4 \) in the set of the initial elements or that of the terminal ones are \( e_2 \) and \( e_3 \). The set of the initial elements of \( e_3 \) is \( \{g_1\} \). It is not a subset of the set of constructing instance graphs of \( g_2 \) \( \{v_3, v_4\} \). Therefore, \( g_2 \) is not well-organized.

Next, the depth of the edges in an instance graph is defined in order to decide whether the instance graph is well-organized, or not.

**Definition 13** Let \( g \) be an instance graph that has an edge \( e \) as a direct constructing edge. the smallest number \( n_{init} \), where \( i(e) \subseteq \bigcup_{m=1}^{n_{init}+1} V^m_{ce}(g) \), is called the depth of the initial element of the edge \( e \) in \( g \). Similarly, the smallest number \( n_{term} \), where \( t(e) \subseteq \bigcup_{m=1}^{n_{term}+1} V^m_{ce}(g) \), is called the depth of the terminal element of the edge \( e \) in \( g \). The largest depth of the initial and the terminal ones is called the depth of the edge \( e \) in \( g \).

**Example 10** The edge \( e_{11} \) in the instance graph \( g_1 \) shown in Fig. 1 is a direct constructing instance graph of the instance graph \( g_1 \). The set of initial elements of \( e_{11} \) is \( \{n_{111}, n_{112}\} \). This set is a subset of the set of the direct constructing instance graphs of \( g_1 \). That is, \( i(e_{11}) \subseteq V^1_{ce} \). Therefore, the depth of the initial element of \( e_{11} \) is equal to zero. Similarly, the depth of the terminal element of \( e_{11} \) is equal to zero. As the result, the depth of the edge \( e_{11} \) is equal to zero. On the other hand, \( e_{13} \) is a direct constructing edge of \( g_1 \). The set of the initial elements of \( e_{13} \) is a subset of \( V^1_{ce} \cup V^2_{ce} \) because \( i(e_{13}) = \{n_{114}, n_{115}\} \). Therefore, the depth of the initial element of \( e_{13} \) is equal to one. Similarly, the depth of the terminal element of \( e_{13} \) is equal to one. As the result, the depth of the edge \( e_{13} \) is equal to one.

The fact that the depth of an initial (terminal, respectively) element of an edge is equal to zero means that the constructing elements of the initial (terminal) element of the edge are neither the initial elements nor the terminal ones of the edge. The edge does not cross the boundary of the instance graph that is an initial or terminal element of the edge. On the other hand, the fact that the depth of an initial (terminal, respectively) element of an edge \( e \) is equal to \( k \) (\( k > 0 \)) means that the \( k + 1 \)th constructing instance graph of the instance graph \( v \) of which direct constructing edge is the edge
e is the initial or terminal element of the edge e. There is at least one edge that crosses k boundaries of the instance graphs. Based on these considerations, the depth of the initial or terminal element of an edge can be used in order to judge whether an instance graph is well-organized, or not.

**Theorem 1** The nth constructing instance graph \( g_{ni} \) of the instance graph \( g \) that is well-organized is well-organized if one of the following conditions is satisfied for every \( k (1 \leq k \leq n) \).

1. \( g_{ni} \) and its constructing elements are neither the initial element nor the terminal one of the kth constructing edge of \( g \).

2. if \( g_{ni} \) or one of its constructing elements is the initial element or the terminal one of the kth constructing edge of \( g \), its depth is at most \( n - k \).

(Proof.) First, the case that all of the constructing edges of \( g \) are the kth constructing edges is studied. When \( g_{ni} \) and its constructing elements are neither the initial element nor the terminal one of the kth constructing edge of \( g \), there is apparently no kth constructing edge that has \( g_{ni} \) or its constructing elements as the initial or terminal elements. As the kth constructing edge is not the edge that has \( g_{ni} \) or its constructing elements as the initial or terminal elements in this case, \( g_{ni} \) is well-organized.

When \( g_{ni} \) or one of its constructing elements is the initial (terminal, respectively) element of the kth constructing edge of \( g \), and its depth is at most \( n - k \), the set of the initial (terminal) elements of the edge is a subset of \( \bigcup_{i=1}^{n-k+1} V_i \) based on the definition of the depth of the initial (terminal) element. The mth \((m \geq 0)\) constructing elements of \( g_{ni} \) are the \((n+m)\)th constructing elements of \( g \). Here, the condition \( n - k + 1 \leq n + m \) is always hold because \( n - k + 1 \leq n \) and \( n \leq n + m \). The equality is held under the condition \( k = 1 \) and \( m = 0 \). In this case, \( g_{ni} \) is the direct (first) constructing element of \( g \). \( g_{ni} \) is the initial or terminal element of the edge. In the other cases, the inequality is held. When \( m > 1 \), the instance graphs are the constructing elements of \( g_{ni} \). Therefore, there is no kth constructing edge that has the constructing elements of \( g_{ni} \) as the initial or terminal elements. From these discussions, \( g_{ni} \) is well organized.

When the conditions described above are held for every \( k (1 \leq k \leq n) \), no constructing edge of \( g \) has the constructing elements of \( g_{ni} \) as the initial or terminal elements. As \( g \) is well-organized, \( g \) and its constructing elements are neither the initial nor the terminal elements of the edges outside \( g \). Therefore, \( g_{ni} \) is well-organized.

The theorem is proved.

When a representative instance graph \(^1\) is given, the constructing instance graphs of \( g \) are obtained by analyzing the constructing elements of \( g \) according to the order of the constructing elements.

Though we have studied the characteristics of the instance graphs, of which edges are limited to have an instance graph as the initial element, and the one as the terminal one, these characteristics can be applied to the general instance graphs.

### 4.2 Shape graph management

Here, the management of shape graphs is studied. The information on the shape graphs is stored into the collection graph named \( Shape \). This collection graph has only one normal instance graph. Its name is \( shape \). This normal instance graph holds four kinds of atomic instance graphs, of which

\(^1\)A representative instance graph is well-organized.
names are cg, ig, connect, and datatype. The atomic instance graph named cg has the name of a collection graph as its value. The one named ig has the name of an instance graph as its value. The connect atomic instance graph has the name of an edge as its value. The instance atomic graph named datatype has the data type name as its value. The normal instance graph shape has four kinds of edges. The name of the first kind of edge is contain. This kind of edge represents the inclusion relationships between a normal instance graph and its direct components, and those between a collection graph and representative instance graphs. The second and the third kinds of edges are for representing the initial elements and the terminal elements of edges. The fourth kind of edge represents the relationship between the atomic instance graph named datatype and the other kinds of atomic instance graphs. The name of this kind of edge is dt.

The information of the shape graph shown in Fig. 3 is managed as the collection graph as shown in Fig. 7.

![Figure 7: The collection graph for managing the information of shape graphs](image)

As the information of the shape graphs are stored as a collection graph, shape graphs can be manipulated by using the rewrite operation. Please note the followings. Shape graphs may change when instance graphs are inserted, or updated. Deletion of instance graphs of the collection graph shape can be permitted when the corresponding instance graphs do not exist. The consistency constraint mechanism must be supported by the database management system.

4.3 Querying power

Here, we present the next theorem in order to show the querying power of the rewrite operation.
Theorem 2  The rewrite operation can be described with datalog program extended with complex values\([30}\).

(Proof.) Let assume that collection graphs are stored in the following relations.

\[
\begin{align*}
\text{cg}(\text{name}, \text{id}) & \text{ of the sort } (\text{dom}, \{\text{dom}\}) \\
\text{ig}(\text{id}, \text{name}, \text{value}) & \text{ of the sort } (\text{dom}, \text{dom}, \text{dom}) \\
\text{edge}(\text{id}, \text{name}, \text{value}) & \text{ of the sort } (\text{dom}, \text{dom}) \\
\text{contain}(\text{parent}, \text{child}) & \text{ of the sort } (\text{dom}, \{\text{dom}\}) \\
\text{connect}(\text{id}, \text{init}, \text{term}) & \text{ of the sort } (\text{dom}, \{\text{dom}\}, \{\text{dom}\})
\end{align*}
\]

where dom means a domain, the pair \{\}\ represents the set constructor, and the pair (\()\ represents the tuple constructor[30]. The relation contain holds the information on the mapping \(\phi_{\text{comp}} : 2^V \cup E \rightarrow V\).

The rewrite operation is represented with \(\text{rewrite}((qg_1, va), (qg_{q1}, rc_{q1}) \cdots (qg_{qn}, rc_{qn}))\). We will show that \(qg_i, rc_i\), and \(va\) can be described with datalog program extended with complex values.

(1) The regular query graphs \(qg_1, qg_{q1}, \cdots qg_{qn}\) are first converted as follows.

(1-1) The regular query graph \(qg_i\) with a variable \(X\)

The predicate \(\text{cg}(\text{name}, X)\) is added to the head of a rule, where \(\text{name}\) is the name of the collection graph that is attached to \(qg_i\).

(1-2) A regular query graph \(qg_i\) with a variable \(X\)

The predicate \(\text{cg}(\text{name}, X)\) is added to the body of a rule, where \(\text{name}\) is the name of the collection graph that is attached to \(qg_i\).

(2) Query graphs are converted as follows.

(2-1) A regular query graph \(qg\) with a variable \(X\)

A variable \(Q\) is adopted for binding the set of the direct constructing elements of \(qg\). The predicate \(\text{contain}(X, Q)\) is added to the body of a rule. The variables \(Y_i(i = 1, \cdots, n)\) that represent the direct constructing elements of \(qg\) are introduced. For each variable \(Y_i\), the predicate \(Q \ni Y_i\) is also added to the body of a rule.

(2-2) An edge \(e\) with a variable \(Y\)

A variable \(S\) is adopted for binding the set of the initial elements of \(e\). A variable \(T\) is also adopted for binding the set of the terminal elements of \(e\). The predicate \(\text{connect}(Y, S, T)\) is added to the body of a rule. The variables \(U_i(i = 1, \cdots, m)\) that represent the sets of the initial elements of \(e\) are introduced. For each variable \(U_i\), the predicate \(S \ni U_i\) is added to the body of a rule. Similarly, The variables \(W_j(j = 1, \cdots, k)\) that represent the sets of the terminal elements of \(e\) are introduced. For each variable \(W_j\), the predicate \(T \ni W_j\) is also added to the body of a rule.

(2-3) An atomic query graph with a variable \(X\)

The variable \(X\) appears as the element of a regular query graph or an edge. Therefore, we have nothing particular to convert an atomic query graph.

(2-4) Concatenation of an edge and a query path

First, we study the concatenation of two edges \(e_1\) and \(e_2\). The edge \(e_1\) is converted into the predicate \(\text{connect}(Y_1, S_1, T_1)\) by using the variable \(Y_1, S_1\) and \(T_1\) as described in (2-2). The
edge \( e_2 \) is similarly converted into the predicate \( \text{connect}(Y_2, S_2, T_2) \). When the terminal element of \( e_1 \) is an initial one of \( e_2 \), the predicates \( T_1 \ni \alpha_1 \) and \( S_2 \ni \alpha_1 \) are added to the body of a rule. This means that a query path, which is a concatenation of two edges, is converted into the predicates described above. Concatenation of an edge and a query path can similarly be converted. In general, the concatenation of edges \( e_1, e_2, \ldots, e_n \) is converted into the predicates as follows.

\[
\text{connect}(Y_1, S_1, T_1), \ldots, \text{connect}(Y_n, S_n, T_n), T_1 \ni \alpha_1, S_2 \ni \alpha_1, \ldots, T_{n-1} \ni \alpha_{n-1}, S_n \ni \alpha_{n-1}.
\]

(2-5) Positive closure (\( qp^+ \))

In treating positive closure, we use the additional rules as used for the recursive rules. We study the positive closure of an edge \( e_1 \) for simplicity. The edge \( e_1 \) is converted into the predicate \( \text{connect}(Y_1, S_1, T_1) \) as described in (2-2). The first rule is as follows.

\[
p_{p_1}(Y, S, T) \leftarrow \text{connect}(Y, S, T).
\]

The second rule is as the following.

\[
p_{p_1}(Y, S, T) \leftarrow p_{p_1}(Y, S, T_2), \text{connect}(Y_x, S_x, T), T_x \ni \alpha, S_x \ni \alpha.
\]

(2-6) Union (\( qp_1 \mid qp_2 \))

Union of query paths \( qp_1 \) and \( qp_2 \) is converted into the separate rules as follows.

\[
p_{\text{union}} \leftarrow qp_1.
\]

\[
p_{\text{union}} \leftarrow qp_2.
\]

(2-7) Negation

Negation of a query path \( qp \) is converted the rule as follows.

\[
p_{\text{neg}} \leftarrow \neg qp.
\]

(3) A retrieval condition \( rc \) is converted as follows.

(3-1) \( X_{\text{name}} = nm \) and \( X_{\text{val}} = val \)

The conjunction of these two predicates is converted into the predicate \( ig(X, nm, val) \) in the body of a rule.

(3-2) A predicate \( X_{\text{name}} = nm \)

When the predicate of this form appears and the predicate of the form of \( X_{\text{val}} = val \) does not appear, the predicate \( ig(X, nm, Z) \) is added to the body of a rule, where \( Z \) is a variable.

(4) A value assignment \( va \) is converted into the predicates as described in (3).

The theorem is proved.

For example, the rewrite operation in Fig. 4 is converted into the following datalog program.

\[
cg("My \text{ picture}", X) \leftarrow cg("Picture", X),
\]

\[
\text{contain}(X, Q), Q \ni Y, Q \ni Z_1, Q \ni Z_2,
\]

\[
\text{contain}(Z_1, R_1), R_1 \ni V_1, \text{contain}(Z_2, R_2), R_2 \ni V_2,
\]

\[
\text{connect}(Y, S, T), S \ni V_1, T \ni V_2,
\]

\[
ig(X, "\text{picture}" , A_1), \text{edge}(Y, "pos", "on"),
\]

\[
ig(Z_1, "\text{object}" , A_2), ig(Z_2, "\text{object}" , A_3)
\]

\[
ig(V_1, "\text{parts}" , A_4), ig(V_2, "\text{parts}" , A_5).
\]
5 Related Works

Many graph-based data models are based on labeled directed graphs\[7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 20, 21\]. These data models do not have the nature of recursive graphs and hypergraphs. In DRHM, recursive graphs and hypergraphs can naturally be represented.

The concepts of recursive graphs and hypergraphs are introduced in the Directed Recursive Labelnode Hypergraphs\[24\] and the self-structured semantic relationship model\[25\]. The Directed Recursive Labelnode Hypergraphs\[24\] is proposed for the representation of knowledge. This model has introduced the concepts of labelnode graphs as well as recursive graphs and hypergraphs. A labelnode graph can be used for representing a node or an edge binding nodes. For example, a relationship (isa Fukui city), which represents that Fukui is a city, is represented as in Fig. 8(a). The rectangle in this figure denotes a labelnode. The initial node, which is isa in this figure, corresponds to the ordinary edge. The labelnodes crossing the arrow, and the terminal one are the ordinary nodes connected by the labelnode edge. Hypergraphs are represented by using this mechanism. The elements in a hypergraph can be ordered. Moreover, a LISP calculus (PLUS (TIMES 3 6 2) 4) is represented as shown in Fig. 8(b). As labelnodes can be described in a labelnode as presented in Fig. 8(b), labelnode graphs have the nature of recursive graphs. The most significant difference between the Directed Recursive Labelnode Hypergraphs and the proposed model is that the former is a knowledge representation model, and the latter is a data model. In the Directed Recursive Labelnode Hypergraphs, a data definition and instances are represented with a connected graph. On the other hand, instance graphs are separately represented with shape graphs which correspond to the data definition in the proposed model. A collection graph holds the same kind of instance graphs. The structure of a collection graph is represented with a shape graph. A shape graph can be used as the clue for finding instance graphs from a collection graph.

![Figure 8: Examples of labelnodes.](image)

The rewrite operation is a logic-based one. It enables users to specify the regular expression. This kind of operation has been proposed in G-Log\[8\], G\(^+\)[10], GraphLog[11]. The rewrite operation of the proposed model is for the recursiveness of the instance graphs as well as for their connectivity. Although the rewrite operation has been proposed in GraphDB\[17\], the regular expression can not be specified in this operation.

The shape graph is a kind of dynamic schema\[27, 19\]. It changes according to the insertion and the modification of a collection graph as in shape\[27\] and DataGuides\[19\]. This behavior is inevitable for the management of semistructured and/or unstructured data in a database. DRHM can manage the semistructured and the unstructured data as well as the structured data.
6 Concluding Remarks

This paper proposed a graph-based data model. The concepts of directed graphs, recursive graphs, and hypergraphs are introduced to the proposed data model. An instance graph is the fundamental unit in representing data. A collection graph is the graph of which components are instance graphs. A shape graph of a collection graph represents the structure of the collection graph. The rewrite operation is a logic-based one. Introducing regular query graphs enables us to specify recursive queries. It is used in updating instance graphs as well as querying on a database. The depth of an instance graph is introduced to show whether the instance graph can be decomposed or not. The shape graphs can be managed by representing them with the instance graphs. The rewrite operation can be represented with the datalog program extended for treating complex values.

The relativity of concepts[25] is the characteristics of data such that the constraints among data segments behaves as a data segment, and a data segment behaves as a constraints. This relativity is thought to be very important to represent concepts. The self-structured semantic relationship model has been extended to take into account this relativity[25]. In this model, the roles of nodes and edges can be exchanged by using the dual operation. Considering the relativity of concepts is another future work. The storage structure for the database based on the proposed data model is another subject of future work.

References


