Integrating Swissprot and GenBank Protein Data Sources using Transformation-based Approach

Sasivimol Kittivoravitkul

June 3rd, 2003

1 Introduction

In [Kit03], we describe our approach on semistructured data integration using transformation-based approach. We propose a semistructured data model, called **YATTA model**, to represent semistructured data for the integration and define a set of primitive transformation rules on the model. We also illustrate how our approach works using a toy example.

In this report, we presents how our approach [Kit03] can be applied to a real world example, semistructured biological data sources.

2 Examples of Biological Data Sources

The semistructured biological data sources that we use in this paper are **Swissprot** [BA00] and **GenBank/GenPept** [BKML+02]. Swissprot is an annotated protein data sources maintained by Swiss Institute of Bioinformatics(SIB) and the European Bioinformatics Institute(EBI). Proteins sequence entries in Swissprot are represented in EMBL format [EMB01] which uses two-character line type prefixes to indicate the type of information. GenBank/GenPept is also an annotated protein datasources. It is maintained by National Center for Biotechnology Information(NCBI). GenBank/GenPept uses GenBank flat file format(GBFF) to represent protein sequence entries. The GBFF format is one of the most commonly used formats in the representation of biological sequence data. Figure 1 and 2 show examples of the contents of a fragment of Swissprot and GenBank protein entry.

```
ID   FTSZ_ANASP     STANDARD;      PRT;   428 AA.
AC   P45482;
 ...
GN   FTSZ OR ALR3858.
OS   Anabaena sp. (strain PCC 7120).
 ...
OX   NCBI_TaxID=103690;
RN   [1]
RP   SEQUENCE FROM N.A.
RX   MEDLINE=96099685; PubMed=8525061;
RA   Zhang C.C., Huguenin S., Friry A.;
 ...
DR   EMBL; Z31371; CAA83241.1; -.
```

Figure 1: Example of Swissprot protein sequence entry.

Each Swissprot entry as shown in Figure 1 begins with an ID (IDentification) line giving the entry name, data class, type of molecule and sequence length. In the example, the entry

'FTSZ_ANASP' is of protein molecule type with length 428 amino acids. It is of standard class which means it is complete up to the standard of Swissprot. The AC (ACcession number) line tells the accession numbers associated with the entry. The GN (Gene Name) line contains the gene names that code for the protein entry. The OS (Organism Species) line specifies the organisms which were the source of the entry whereas the OX (Organism taxonomy cross-reference) line is used to indicate the identifier to a specific organism. The sources from which the data has been abstracted are indicated in the RN, RP, RX and RA lines. The DR (Database cross-Reference) line is used as pointers to information related to the entry. The format of the DR line depends on cross-references. In the example, the DR line provides information about the relationship between the entry and the EMBL/GenBank/DDBJ nucleotide sequence entry followed by the protein and status identifier of the entry in protein database of EMBL/GenBank/DDBJ.

```
LOCUS   CAA83241   428 aa   linear   BCT 30-NOV-1995
 ...
FEATURES       Location/Qualifiers
     source    1..428
               /organism="Anabaena sp."
               /db_xref="taxon:103690"
...
     Protein   1..428
               /product="FtsZ"
     CDS       1..428
               /coded_by="Z31371.1:385..1671"
               /transl_table=11
               /db_xref="SWISS-PROT:P45482"
 ...
```

Figure 2: Example of GenBank protein sequence entry.

Each GenBank entry consists of a number of keywords, associated sub-keywords, feature table and sequence. As can be seen in Figure 2, the GenBank entry starts with a LOCUS line. The first element on this line is the LOCUS name followed by the length of the sequence, molecule type, the GenBank division code and the date the entry was last made public. The line prefixed with a word 'Features' is the feature table, which contains the most important information in the entry. In the example, the feature table contains three parts which are source feature, protein feature and Coding Regions (CDS) feature. In the source feature part, mandatory feature in the entry is summarised such as the start and end position of the sequence, scientific name of the source organism and the unique identification number for the taxon of the source organism. The protein feature part describes a product of protein. The CDs part is an instruction to translate a nucleotide into protein via a genetic code. A coding region serves a link between the two sequences. The cross-reference to other protein databases is also provided. The example shows the cross-reference to the Swissprot.

Here we only describe some conventions used in Swissprot and GenBank of the example. The complete details about the EMBL and GBFF format can be found in [EMB01] and [NCB03].

3 YATTA Models for Swissprot and GenBank data sources

The **YATTA model** consists of a **YATTA schema** and **YATTA data**, representing the structure and content of semistructured data respectively. The models are rooted trees with labelled nodes. In a YATTA schema, each node is described by a tuple $\langle label, type \rangle$ where *label* is a string describing what a node represents, *type* represents data type of a node, and *value* represents the value associated with the node. The *type* can be atomic *e.g.* string, integer *etc.* or compound *i.e.* set (marked '{}'), list (marked '[]') or bag (marked '⟨⟩'). The edges of

a YATTA schema are labelled with cardinality constraints, where '∗' indicates zero or more occurrences, '+' indicates one or more occurrences, and '?' indicates zero or one occurrence. No label and '-' indicate exactly one occurrence. '-' is also used to identify a key node in a YATTA schema. Examples of YATTA schemas for Swissprot and GenBank data sources are illustrated in Figure 3 and 4.

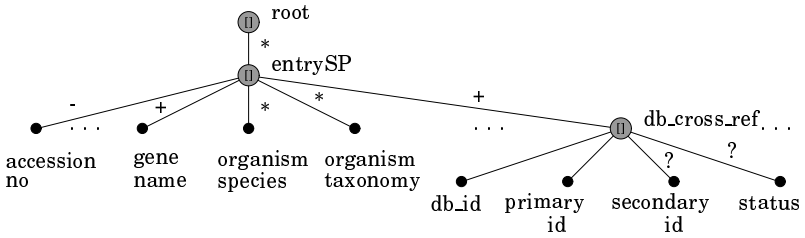

Figure 3: Example of YATTA schema for SWISSPROT protein sequence entry.

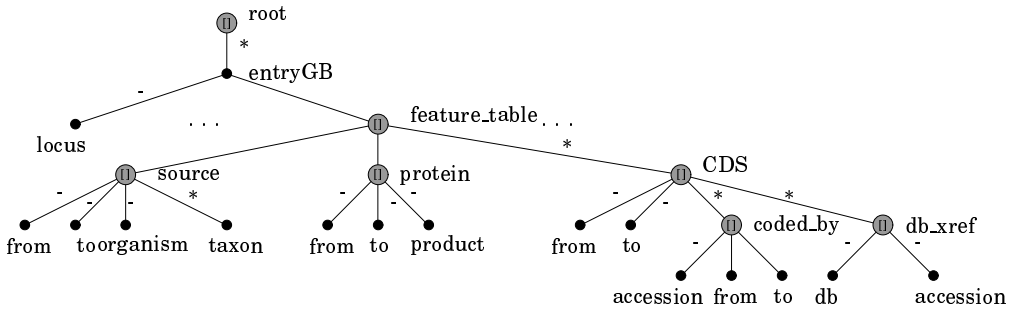

Figure 4: Example of YATTA schema for GENBANK protein sequence entry.

A YATTA data conforms to the YATTA schema. In a YATTA data, each node is described by a triple ⟨*label, type, value*⟩, where *label* is a string describing what a node represents, *type* represents data type of a node and *value* represents the value associated with the node. For a node with compound type, *value* is an integer identifer *id*. The edges in a YATTA data has no label. The outgoing edges of list nodes are ordered from left to right whereas those of set and bag nodes are unordered. The edge is repeated the number of times data occurs, and data values are shown under leaf nodes. Examples of YATTA data for Swissprot and GenBank data sources are shown in Figure 5 and 6. The leaf node which contains actual data is represented by a white circle in the model.

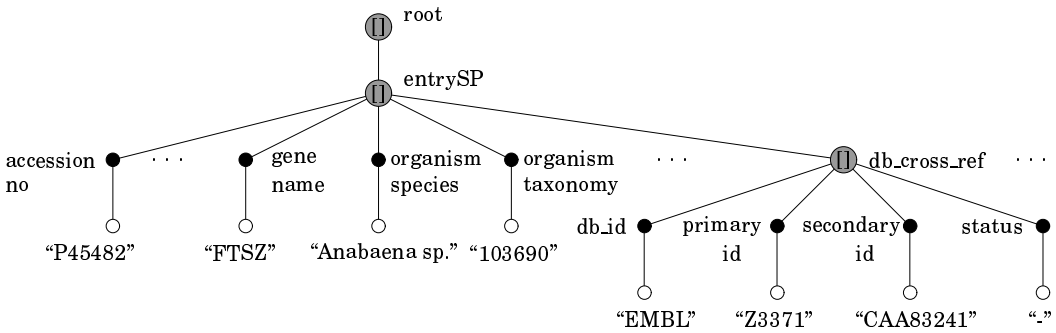

Figure 5: Example of YATTA data for SWISSPROT protein sequence entry.

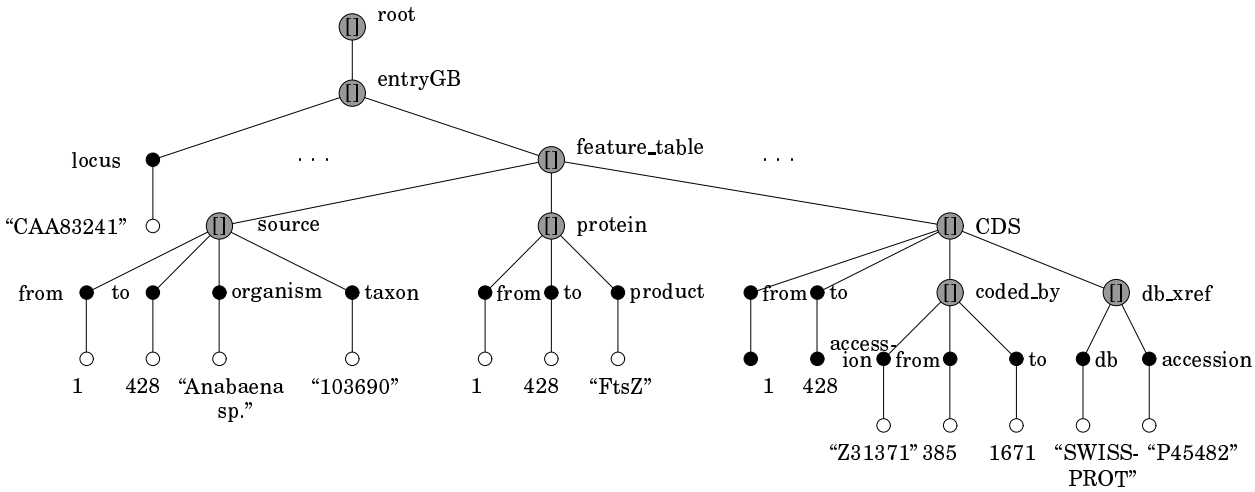

Figure 6: Example of YATTA data for GENBANK protein sequence entry.

4 Integration Approach

In our approach, data integration is done by defining associations between constructs in different schema using primitive transformation rules. In other words, the primitive transformation rules are applied to source schemas so that the schemas are in the equivalent form and are ready for merging into the global view. The primitive transformation rules on YATTA model are summarised as follow. The details can be found in [Kit03].

- renameYattaNode($\langle\langle a, n_1, t, c\rangle\rangle, \langle\langle a, n_2, t, c\rangle\rangle$) renames a node n_1 to have a new name n_2.
- addYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) adds a new node n linked to an existing node a_p, where a_p is the last node in a. The value of a is the value of a_p. The query q specifies the extent of the binary relationship between a_p and n. A position query p stating the position n relative to other sibling nodes of a_p. This position query may either be before(n'), after(n'), or any, where n' is some other node linked to a_p and any means in any place relative to other siblings.
- deleteYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) deletes an existing node n and its relationship with a_p where a_p is the last node in a. The query q tells how the extent of the node can be restored from the remaining constructs. A proviso associated with the transformation is that the node n does not have any child nodes.

In data integration, it is often the case that source schemas may not be precisely equivalent. Two more transformation rules which allow transformations between overlapping YATTA schema are provided. These rules can be defined in terms of the add and delete rule above where the query q can not return all values associated with the constructs.

- extendYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) adds a new node n. The extent of n is only partially defined by the transformation. If no values can be determined by any query, then the special value void may be used for q, or the query field omitted completely.
- contractYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) deletes an existing node n. Only some values of n may be restored by the transformation. The value void may be used where no values can be determined, or the query field omitted. A proviso associated with the transformation is that the n does not have any child nodes.

For every primitive transformation, there exists a **reverse primitive transformation**. The reverse of transformation addYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) is deleteYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$), and

vice versa. The reverse of extendYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$) is contractYattaNode($\langle\langle a, n, t, c\rangle\rangle, q, p$), and *vice versa*.

Figure 7 shows a framework of our approach to integrating two protein data sources, Swissprot and GenBank/GenPept. The protein data sources are represented in the YATTA model and transformed into the equivalent form using a sequence of transformations, called a pathway. The pathway from the YATTA model of Swissprot to that of GenBank/GenPept , **Swissprot → GenBank/GenPept**, is illustrated as below.

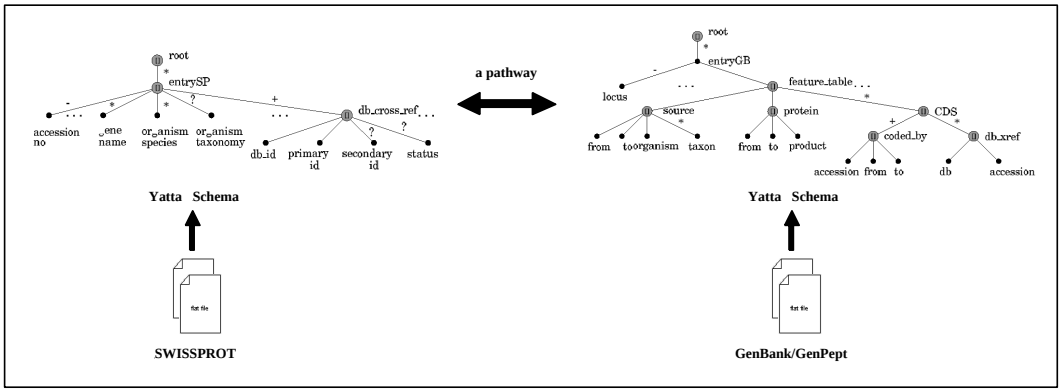

Figure 7: A framework for integrating Swissport and GenBank/GenPept data sources

In the transformation rules, the query q tells the correspondence of instances of one model to another as mentioned earlier. To simplify the presentation of rules, we use functions to represent queries over the source schemas. For example, the instances of the locus node in the semistructured GenBank/GenPept model of Figure 4 correspond to instances of secondary_id in semistructured Swissprot model of Figure 3 if that node is a child of a db_cross_ref node that also has a child db_id node equal to 'EMBL'. We may find these secondary_id by the function f below:

$$
\begin{aligned}
\text{Swissprot}: \quad f \;=\; & \{\langle x, y, z\rangle \mid \langle x, v\rangle \in \langle\langle \text{root, entrySP}\rangle\rangle, \\
& \langle v, w\rangle \in \langle\langle \text{root.entrySP, db_cross_ref}\rangle\rangle, \\
& \langle w, u\rangle \in \langle\langle \text{root.entrySP.db_cross_ref, db_id}\rangle\rangle, \\
& u = \textit{'EMBL'}, \\
& \langle u, y\rangle \in \langle\langle \text{root.entrySP.db_cross_ref, primary_id}\rangle\rangle\} \\
& \langle u, z\rangle \in \langle\langle \text{root.entrySP.db_cross_ref, secondary_id}\rangle\rangle\} \\
g \;=\; & \{\langle x, y, z, w, v\rangle \mid \langle x, u\rangle \in \langle\langle \text{root, entrySP}\rangle\rangle, \\
& \langle u, y\rangle \in \langle\langle \text{root.entrySP, accession}\rangle\rangle, \\
& \langle u, z\rangle \in \langle\langle \text{root.entrySP, organism species}\rangle\rangle, \\
& \langle u, w\rangle \in \langle\langle \text{root.entrySP, organism taxonomy}\rangle\rangle, \\
& \langle u, v\rangle \in \langle\langle \text{root.entrySP, gene name}\rangle\rangle\}
\end{aligned}
$$

GenBank :

$$
\begin{aligned}
h &= \{\langle x,y\rangle \mid \langle x,z\rangle \in \langle\!\langle \text{root, entryGB}\rangle\!\rangle,\\
&\quad \langle z,w\rangle \in \langle\!\langle \text{root.entryGB, feature_table}\rangle\!\rangle\},\\
&\quad \langle w,u\rangle \in \langle\!\langle \text{root.entryGB.feature_table, CDS}\rangle\!\rangle,\\
&\quad \langle u,v\rangle \in \langle\!\langle \text{root.entryGB.feature_table.CDS, db_xref}\rangle\!\rangle,\\
&\quad \langle v,t\rangle \in \langle\!\langle \text{root.entryGB.feature_table.CDS.db_xref, db}\rangle\!\rangle,\\
&\quad t = \textit{'SWISS-PROT'},\\
&\quad \langle t,y\rangle \in \langle\!\langle \text{root.entryGB.feature_table.CDS.db_xref, accession}\rangle\!\rangle\}\\
i &= \{\langle x,y\rangle \mid \langle x,z\rangle \in \langle\!\langle \text{root,entryGB}\rangle\!\rangle,\\
&\quad \langle z,w\rangle \in \langle\!\langle \text{root.entryGB, feature_table}\rangle\!\rangle,\\
&\quad \langle w,u\rangle \in \langle\!\langle \text{root.entryGB.feature_table, CDS}\rangle\!\rangle,\\
&\quad \langle u,v\rangle \in \langle\!\langle \text{root.entryGB.feature_table.CDS, coded_by}\rangle\!\rangle,\\
&\quad \langle v,y\rangle \in \langle\!\langle \text{root.entryGB.feature_table.CDS.coded_by, accession}\rangle\!\rangle\}\\
j &= \{\langle x,y\rangle \mid \langle x,z\rangle \in \langle\!\langle \text{root, entryGB}\rangle\!\rangle,\\
&\quad \langle z,w\rangle \in \langle\!\langle \text{root.entryGB, feature_table}\rangle\!\rangle,\\
&\quad \langle w,u\rangle \in \langle\!\langle \text{root.entryGB.feature_table, protein}\rangle\!\rangle,\\
&\quad \langle u,y\rangle \in \langle\!\langle \text{root.entryGB.feature_table.protein, product}\rangle\!\rangle\}\\
k &= \{\langle x,y\rangle \mid \langle x,z\rangle \in \langle\!\langle \text{root, entryGB}\rangle\!\rangle,\\
&\quad \langle z,w\rangle \in \langle\!\langle \text{root.entryGB, feature_table}\rangle\!\rangle,\\
&\quad \langle w,u\rangle \in \langle\!\langle \text{root.entryGB.feature_table, source}\rangle\!\rangle,\\
&\quad \langle u,y\rangle \in \langle\!\langle \text{root.entryGB.feature_table.source, organism}\rangle\!\rangle\}\\
l &= \{\langle x,y\rangle \mid \langle x,z\rangle \in \langle\!\langle \text{root, entryGB}\rangle\!\rangle,\\
&\quad \langle z,w\rangle \in \langle\!\langle \text{root.entryGB, feature_table}\rangle\!\rangle,\\
&\quad \langle w,u\rangle \in \langle\!\langle \text{root.entryGB.feature_table, source}\rangle\!\rangle,\\
&\quad \langle u,y\rangle \in \langle\!\langle \text{root.entryGB.feature_table.source, taxon}\rangle\!\rangle\}\\
m &= \{\langle x,y\rangle \mid \langle x,z\rangle \in \langle\!\langle \text{root, entryGB}\rangle\!\rangle,\\
&\quad \langle z,y\rangle \in \langle\!\langle \text{root.entryGB, locus}\rangle\!\rangle\}
\end{aligned}
$$

Transform Swissprot → GenBank

① addYattaNode(⟪root, entryGB, list, $*$⟫,
$\{\langle x,y\rangle \mid \langle x,_,z\rangle \in f \wedge y = \text{generateId}(\langle x,z\rangle)\}$, before(⟪root, entrySP⟫))

② addYattaNode(⟪root.entryGB, locus, string, -⟫,
$\{\langle y,z\rangle \mid \langle x,_,z\rangle \in f \wedge y = \text{generateId}(\langle x,z\rangle)\}$, any)

③ addYattaNode(⟪root.entryGB, feature_table, list, ⟫,
$\{\langle x,x\rangle \mid \langle y,_,z\rangle \in f \wedge x = \text{generateId}(\langle y,z\rangle)$, after(⟪root.entryGB, locus⟫)$\}$)

④ addYattaNode(⟪root.entryGB.feature_table, source, string, +⟫,
$\{\langle x,y\rangle \mid \langle v,u,w,_,_\rangle \in g \wedge\ x = \text{generateId}(\langle v,u\rangle) \wedge\ \text{getOrder}(\langle x,w\rangle) = 1$
$\wedge\ y = \text{generateId}(\langle v,w\rangle)\}$, any)

⑤ extendYattaNode(⟪root.entryGB.feature_table.source, from, integer, -⟫, $void$, any)

⑥ extendYattaNode(⟪root.entryGB.feature_table.source, to, integer, -⟫, $void$,
after(⟪root.entryGB.feature_table.source, from⟫))

⑦ addYattaNode(⟪root.entryGB.feature_table.source, organism, string, -⟫,
$\{\langle x,y\rangle \mid \langle v,u,w,t,_\rangle \in g \wedge\ z = \text{generateId}(\langle v,u\rangle) \wedge x = \text{generateId}(\langle v,w\rangle)$
$\wedge\ \text{getOrder}(\langle x,w\rangle) = 1 \wedge y = w\}$, after(⟪root.entryGB.feature_table.source, from⟫))

⑧ addYattaNode(⟪root.entryGB.feature_table.source, taxon, string, $*$⟫,
$\{\langle x,y\rangle \mid \langle v,u,w,t,_\rangle \in g \wedge\ z = \text{generateId}(\langle v,u\rangle) \wedge x = \text{generateId}(\langle v,w\rangle)$
$\wedge\ \text{getOrder}(\langle x,w\rangle) = 1 \wedge \text{getOrder}(\langle x,t\rangle) = 1 \wedge y = t\}$,
after(⟪root.entryGB.feature_table.source, organism⟫))

⑨ addYattaNode(⟪root.entryGB.feature_table, protein, list, ⟫,
$\{\langle x, y\rangle \mid \langle v, u, _, _, w\rangle \in g \wedge\ x = \mathrm{generateId}(\langle v, u\rangle)$
$\wedge\ \mathrm{getOrder}(\langle x, w\rangle) = 1 \wedge y = \mathrm{generateId}(\langle v, w\rangle)\}, \mathrm{after}(⟪$root.entryGB.feature_table, source$⟫))$

⑩ extendYattaNode(⟪root.entryGB.feature_table.protein, from, integer, -⟫, $void, \mathrm{any})$

⑪ extendYattaNode(⟪root.entryGB.feature_table.protein, to, integer, -⟫, $void$,
$\mathrm{after}(⟪$root.entryGB.feature_table.protein, from$⟫))$

⑫ addYattaNode(⟪root.entryGB.feature_table.protein, product, string, -⟫,
$\{\langle x, y\rangle \mid \langle v, u, _, _, w\rangle \in g \wedge\ x = \mathrm{generateId}(\langle v, w\rangle) \wedge \mathrm{getOrder}(\langle x, w\rangle) = 1 \wedge y = w\}$,
$\mathrm{after}(⟪$root.entryGB.feature_table.protein, to$⟫))$

⑬ extendYattaNode(⟪root.entryGB.feature_table, CDS, list, $*$⟫, $void$,
$\mathrm{after}(⟪$root.entryGB.feature_table, protein$⟫))$

⑭ extendYattaNode(⟪root.entryGB.feature_table.CDS, from, integer, -⟫, $void, \mathrm{any})$

⑮ extendYattaNode(⟪root.entryGB.feature_table.CDS, to, integer, -⟫, $void$,
$\mathrm{after}(⟪$root.entryGB.feature_table.CDS, from$⟫))$

⑯ extendYattaNode(⟪root.entryGB.feature_table.CDS, coded_by, list, $*$⟫,
$\{\langle _, y\rangle \mid \langle v, _, __, z\rangle \in g \wedge\ y = \mathrm{generateId}(\langle v, z\rangle)\}, \mathrm{any})$

⑰ addYattaNode(⟪root.entryGB.feature_table.CDS.coded_by, accession, string, -⟫,
$\{\langle x, z\rangle \mid \langle v, _, __, z\rangle \in g \wedge x = \mathrm{generateId}(\langle v, z\rangle)\}, \mathrm{any})$

⑱ extendYattaNode(⟪root.entryGB.feature_table.CDS.coded_by, from, integer, ⟫, $void$,
$\mathrm{after}(⟪$root.entryGB.feature_table.CDS.coded_by, accession$⟫))$

⑲ extendYattaNode(⟪root.entryGB.feature_table.CDS.coded_by, to, integer, ⟫, $void$,
$\mathrm{after}(⟪$root.entryGB.feature_table.CDS.coded_by, from$⟫))$

⑳ extendYattaNode(⟪root.entryGB.feature_table.CDS, db_xref, list, $*$⟫,
$\{\langle _, x\rangle \mid \langle v, y, _, _, _\rangle \in g \wedge\ x = \mathrm{generateId}(\langle v, y\rangle)\}$,
$\mathrm{after}(⟪$root.entryGB.feature_table.CDS, coded_by$⟫))$

㉑ addYattaNode(⟪root.entryGB.feature_table.CDS.db_xref, db, string, -⟫,
$\{\langle x, z\rangle \mid \langle v, y, _, _, _\rangle \in g \wedge\ x = \mathrm{generateId}(\langle v, y\rangle)\} \wedge z = \text{`}SWISSPROT\text{'}, \mathrm{any})$

㉒ addYattaNode(⟪root.entryGB.feature_table.CDS.db_xref, accession, string, -⟫,
$\{\langle x, y\rangle \mid \langle v, y, _, _, _\rangle \in g \wedge\ x = \mathrm{generateId}(\langle v, y\rangle)\}$,
$\mathrm{after}(⟪$root.entryGB.feature_table.CDS.db_xref, db$⟫))$

㉓ contractYattaNode(⟪root.entrySP.db_cross_ref, status, string, ?⟫,
$\{\langle x, _\rangle \mid \langle z, w\rangle \in l \wedge\ x = \mathrm{generateId}(\langle \mathrm{z}, \mathrm{w}\rangle)\}, \mathrm{after}(⟪$root.entrySP.db_cross_ref, secondary id$⟫))$

㉔ deleteYattaNode(⟪root.entrySP.db_cross_ref, secondary id, string, ?⟫,
$\{\langle x, y\rangle \mid \langle z, w\rangle \in i \wedge\ \langle z, y\rangle \in m \wedge x = \mathrm{generateId}(\langle z, w\rangle)\}$,
$\mathrm{after}(⟪$root.entrySP.db_cross_ref, primary id$⟫))$

㉕ deleteYattaNode(⟪root.entrySP.db_cross_ref, primary id, string, -⟫,
$\{\langle x, y\rangle \mid \langle z, y\rangle \in i \wedge\ x = \mathrm{generateId}(\langle z, y\rangle)\}, \mathrm{after}(⟪$root.entrySP.db_cross_ref, db_id$⟫))$

㉖ deleteYattaNode(⟪root.entrySP.db_cross_ref, db_id, string, ⟫,
$\{\langle x, y\rangle \mid \langle z, w\rangle \in i \wedge\ x = \mathrm{generateId}(\langle z, w\rangle) \wedge y = \text{`}EMBL\text{'}\}, \mathrm{any})$

㉗ deleteYattaNode(⟪root.entrySP, db_cross_ref, list, +⟫,
$\{\langle x, y\rangle \mid \langle z, w\rangle \in i \wedge y = \mathrm{generateId}(\langle z, w\rangle) \wedge\ \langle z, u\rangle \in m \wedge x = \mathrm{generateId}(\langle z, u\rangle)\}$,
$\mathrm{after}(⟪$root.entrySP, organism taxonomy$⟫))$

㉘ deleteYattaNode(⟪root.entrySP, organism taxonomy, string, $*$⟫,
$\{\langle x, y\rangle \mid \langle z, w\rangle \in m \wedge\ x = \mathrm{generateId}(\langle z, w\rangle) \wedge \langle z, y\rangle \in l\}$,
$\mathrm{after}(⟪$root.entrySP, organism species$⟫))$

㉙ deleteYattaNode(⟪root.entrySP, organism species, string, $*$⟫,
$\{\langle x, y\rangle \mid \langle z, w\rangle \in m \wedge\ x = \text{generateId}(\langle z, w\rangle) \wedge \langle z, y\rangle \in k\}$,
after(⟪root.entrySP, gene name⟫))

㉚ deleteYattaNode(⟪root.entrySP, gene name, string, +⟫,
$\{\langle x, y\rangle \mid \langle z, w\rangle \in m \wedge\ x = \text{generateId}(\langle z, w\rangle) \wedge \langle z, y\rangle \in j\}$,
after(⟪root.entrySP, accession no⟫))

㉛ deleteYattaNode(⟪root.entrySP, accession no, string, -⟫,
$\{\langle x, y\rangle \mid \langle z, y\rangle \in m \wedge\ x = \text{generateId}(\langle z, y\rangle)\}$, any)

㉜ deleteYattaNode(⟪root, entrySP, list, $*$⟫,
$\{\langle _, x\rangle \mid \langle z, y\rangle \in m \wedge\ x = \text{generateId}(\langle z, y\rangle)\}$, after(⟪root, entryGB⟫))

In transformation ①, a new entryGB node in the GenBank/GenPept model is added for every appropriate entry in the Swissprot model. The special function 'generateId' ensured the content of the data sources is preserved. It takes a set of tuples and returns a set of new nodes which are related to each tuple. Each new node is assigned a new integer identifier *id* for each distinct tuple. An empty set is returned if node representing the relation of the tuple already exists. The new entryGB nodes are associated to the root of the Swissprot model, and their position is before the entrySP node.

Transformation ② and ③ perform a similar process to add locus and feature_table nodes. Since locus is the first sibling of entryGB node, its order is set to any. Note that in the transformations, adding a new node, which is of compound type, requires the function 'generateId' as shown in transformation ①. The node of compound type cannot exists if none of its child exits, particularly the key nodes. The values of the compound node are the integer identifiers which are returned by 'generateId' function for distinct key nodes. However, in the case of feature_table node, there exists feature_table node for every protein entry. Its values therefore depends on the existing of the entry, identifed by locus node.

In transformation ④, the extents of source node depend on the first instance of organism species for each entry in the Swissprot model. The function 'getOrder' is used to get the order of organism species for every distinct feature_table node.

Similar to ④, transformation ⑤ adds a new node organism whose values is the first value of organism species in the protein entry in Swissprot.

Transformation ⑤ and ⑥ add from and to nodes. Since the instances of these new nodes cannot be determined from the Swissprot model, the extendYattaNode with the value void is applied.

Once every node in the GenBank/GenPept that does not exist in the Swissprot is added, a growing phrase (transformation ①-㉒), the redundant nodes of the Swissprot that do not appear in the GenBank/GenPept are removed, a shrinking phrase (transformation ㉓-㉜).

For all deleted nodes in the Swissprot model, except the status node, their instances can be restored from the GenBank/GenPept model. For example, in transformation ㉔, instances of the deleted secondary id node can be restored from locus node in the GenBank/GenPept model. Also, in transformation ㉕ the values of primary id can be recovered from accession (⟪root.entryGB.feature_table.CDS.coded_by, accession⟫) node.

Since every primitive transformation is automatically reversible, the transformation pathway **GenBank** → **Swissprot** therefore can be automatically derived by reversing the tranformation ①-㉜. Data, queries and updates can then be automatically migrated in either directions between two schemas link by a pathway. Double-headed arrows in Figure 7 is used to indicate this process. The details of the process in migrating data sources can be found in [MP99].

5 Conclusions

Integrating biological data sources such as Swissprot and GenBank/GenPept protein data sources is a challege since the data is complex *i.e.* for any give record, there could be many levels of nesting, each with many levels of logical structure such as bags, sets, lists and so on as shown in Section 2.

In this report, we have illustrated how our approach to integrating semistructured data using transformation-based approach can be applied to such data sources, Swissprot and GenBank/GenPept. The integration of the protein data sources has shown that our approach applies well with biological data which is often held in semistructured data format. The main advantage of our approach according to the BAV, demonstrated in [Kit03] but not in this report, is the ability to handle evolution of the sources and global schema/view by incrementally extending and modifying a sequence of transformations. This also shows that our approach is suitable to apply on biological sources whose structure and content are often evolved.

So far, our approach has consisted purely of the virtual view definition where no actual data is stored in the global schema. The data movement and translation are only required at runtime. Since biological data sources are extremely large, the virtual view approach may impose CPU loads. We plan to extend our approach to also support the materialised approach where integrated data is stored in a warehouse. This is a trivial task since the data from a YATTA data can be easily migrated to the relational databases using tranformations between the YATTA model and the relational model as described in [Kit03]. But the issue concerns in supporting the materialised approach is to keep the warehouse consistent as the underlying souces. This needs to be investigated.

Our approach is currently being implemented within the ongoing AutoMed project (http://www.doc.ic.ac.uk/automed/).

References

[BA00] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. *Nucleic Acids Research*, 28(1):45–48, 2000.

[BKML+02] D.A. Benson, L. Karsch-Mizrachi, D.J. Lipman, J. Ostell, B.A. Rapp, and D.L. Wheeler. Genbank. *Nucleic Acids Research*, 20(1):17–20, 2002.

[EMB01] EMBL outstation, European Bioinformatics Institute, http://www.ebi.ac.uk/embl. *EMBL Nucleotide Sequence Database User Manual*, 2001.

[Kit03] S. Kittivoravitkul. Transformation-based approach for integrating semistructured data. Technical Report 14, Imperial College London, UK, http://www.doc.ic.ac.uk/automed/techreports/, January 2003.

[MP99] P.J. Mcbrien and A. Poulovassilis. Automatic migration and wrapping of database applications - a schema transformation approach. In *Proceedings of ER99*, volume 1728 of LNCS, pages 96–113. Springer-Verlag, 1999.

[NCB03] NCBI, National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html. *GenBank Flat File Format*, 2003.