XML Object Relational Translation (XORT)

Pinglei Zhou*, Stan Letovsky, Chris Mungall, David Emmert, Peili Zhang, William Gelbart

* Correspondence: Pinglei Zhou, Email: zhou@morgan.harvard.edu Fax: 617-496-1354

Abstract
XML Object Relational Translation (XORT) is mapping specification which map objects in relational database into XML structure, besides the mapping specification, XORT package come with several tool services. Although XORT is designed to be generic enough to with any database schemas which satisfy certain criteria, XORT project is sparked by the Chado schema design, therefore, most of our documentation and examples will refer to Chado schema, and the corresponding XML will denote as Chado XML.

Source
http://www.gmod.org

Content

1. Criteria for database schema design

All primary keys are serial numbers, and each table has unique key(s). Unique key is the only way to represent a unique record out of database environment since primary key is meaningless if outside of the database context. Besides, it is highly recommended that no table and columns share the same name.

2. Chado XML Specification

2.1 . XML Elements

All the table names from database schema will become XML elements. All the table columns also will become XML elements, except those database internal columns, say all the primary keys in Chado schema.

The primary-foreign relationship is reflected as nesting in XML. From data structure's point of view, XML is tree. A *chado* database becomes a tree when converted into XML, with the root node being *chado*, the non-leaf node being either table node, which means the label of node is one of table names, or column node, which means the label of node is one of the column names of the corresponding database table. Direct children of root node are table nodes. Nodes of a table node could be either column node (which must appear before any other table element), or table node if this child table node have foreign

key referring to the parent node's primary key. For example, node of feature could be uniquename which is a column of feature, or featureprop because of the fact that featureprop has feature_id column referring to the primary key of feature:feature_id. Child of column node could be either leaf node with no label but only content of the column, or table node whose children node represent the content of this column node if this column node is also foreign key node.

2.2. XML Attributes

There are two attributes: op and id, op for operations and id for identifier

id will use for macro- mechanism, more specifically, you name some object, then call it.

There are five types of operations: lookup, insert, delete, update, and force. The first four are analogs to typical SQL operation: select, insert, delete and update, respectively. Force is the combination of lookup, insert and update, which means first look up to see if the record already in database or not, if answer is 'no', apply insert, otherwise, apply update if necessary. One thing need to be pointed out, while lookup insert, delete and force only apply to table elements, update need to specify to both table and column element.

2.3. Object reference

Similar as specifying a SQL statement, in order to uniquely refer to some object in XML, we only want to specify the least information which will be unique enough to identify the object, this is denoted object reference. There are fours types of object reference in Chado XML:

2.3.1. First, global accession for something well-known, or well-defined object, say GenBank accession or FlyBase gene identifier.

2.3.2. Secondly, for pre-defined local_id (also call macro), give some identifier for something you want to re-use in SAME file.

2.3.3. Third one is to specify all unique key value(s).

2.3.4. Implicitly as join

Last one is implicitly, which use foreign-primary key relationship. For instance, if you nest featureloc with feature element, omit the featureloc's feature_id means it refer to its parent object-feature in this example. Here will retrieve the value from context.

The question is what type of columns are allowed the context retrieval.

2.3.5. Order of object reference

If required elements are missed (limit to one), it first look for up-level (assume up-level is table element), throw error if fails.

For exist column element, if sub-element is table element, assume to be foreign-primary key mechanism. If sub-element is text value, first check if it is pre-defined as local_id, if not, then check if it is global accession using some customized decode method.

Examples of object reference as below:

Global accession example:

```
<chado>
 ….
  <feature>
  <uniquename>CG3123</uniquename>
 <type_id>gene</type_id>
 <feature_relationship>
    <subject_id>FlyBase:CG3123-RA</subject_id>
    <type_id>producedby</type_id>
 <feature_relationship>
 ……

</feature>
….
</chado>
```

Here '**FlyBase:CG3123-RA'** refers to one transcript from FlyBase.

Local ID example:

```
<cv id="SO">
   <name>Sequence Ontology</name>
</cv>

<cvterm id="exon">
        <cv_id>SO</cv_id>
        <name>exon</name>
</cvterm>

<feature>
         <type_id>exon</type_id>
        ….
</feature>
```

This is an example to show how to use local ID reference mechanism. As shown above, The cv record 'Sequence Ontology' is named as 'SO', so it can be called as 'SO' whenever need to represent this cv record.
Two points need to mention here, first, need to avoid ambiguity, never give same name to two different object, second, those identifier only valid within same file, even though we plan to implement external id mechanism in the future.

Example for unique key mechanism.

```
<feature>
   <type_id>
      <cvterm>
         <cv_id>
```

```
                    <cv>
                      <name>Sequence Ontology</name>
                    </cv>
                </cv_id>
                <name>exon</name>
            </cvterm>
        </type_id>
```

here in order to identify the type_id for this feature, we need to specify all unique keys value for cvterm record, which include cvterm.name and cvterm.cv_id.

Example of Implicitly using foreign key to identify information in the related link table

```
<feature op="lookup" id="CG3312">
    <uniquename>CG3312</uniquename>
    <type_id>
        <cvterm>
            <name>gene</name>
            <cv_id>SO</cv_id>
        </cvterm>
    <type_id>
    <organism_id>
    <feature_relationship>
        <subject_id>Gadfly:CG3312-RA</subject_id>
         <type_id>producedby</type_id>
     </feature_relationship>
 </feature>
```

As shown in the above example, for feature_relationship element, it miss the unique key: object_id, which implicitly refer to its parent object, feature.

Transactional Chado XML

Typical Chado-XML documents are assumed to be *static* or *snapshot*. They are atemporal - they contain the state of the data at one particular instance in time.

Another variant Chado-XML is Transactional-Chado-XML. This represents data manipulation operations (transactions) between two instants in time.

Transactional-Chado-XML uses the same XML elements as static Chado-XML. Additional attributes are used to represent insert/lookup/delete/store operations which are isomorphic to SQL insert/select/delete/select+(update|insert) statements.

Notice that all operation will be atomic, even though they nest like joining together.

Chado XML define 5 operation: lookup, insert, delete, update and force.

```
<cvterm  op=”lookup”>
  <name>gene</cvterm>
  <cv_id>
     <cv>
       <name>SO</name>
     </cv>
 </cv_id>
</cvterm>
```

The logic in here is that, first parse the cvterm record, first parse the name value, since it is text value, will store, then check cv_id, since its child is table element, then first parse this atomic block as following: select cv_id from cv where name=’SO’, return the value for cv_id, say here return with value as 999(same for the rest examples), then come back to finish the cvterm: select * from cvterm where name=’gene’ and cv_id=999.

Here it will throw as error if this cvterm not exist in DB yet. This is particular useful for any well defined record, such as cvterm, pub, dbxref, you don’t want to mess up with it.

```
<cvterm  op=”insert”>
  <name>gene</cvterm>
  <cv_id>SO</cv_id>
</cvterm>
```

will force to insert this record into: insert into cvterm (name, cv_id) values (‘gene’, 999); Certainly it will throw duplicate error message if this record already in the DB.

```
<cvterm  op=”delete”>
  <name>gene</cvterm>
  <cv_id>SO</cv_id>
</cvterm>
```

The corresponding SQL will be: delete from cvterm where name=’gene’ and cv_id=999. and will throw a warning if it does not exist.

```
<cvterm  op=”force”>
  <name>gene</cvterm>
  <cv_id>SO</cv_id>
</cvterm>
```

Default op attribute will be “force”. It will first look up in DB, insert if not exist, and otherwise update if necessary.

```
<synonym  op=”update”>
  <name>dpp</cvterm>
  <synonym_sgml>dpp</synonym_sgml>
  <synonym_sgml op=”update”>app</synonym_sgml>
</synonym>
```

will convert into SQL: update synonym set synonym_sgml=’app’ where name=’dpp’ and synonym_sgml=’dpp’

2.4. Policy for non-anscii character
2.5. Empty string vs null value
Empty string: <name/>
Null value: <name null="true"/>
Actual value as 'null' <name>null</name>

3. Tools

3.1 DTD generator
which automatically generate the DTD based on the database schema

3.2. Validator
Which valid the Chado XML against the Chado Database schema, it includes:
Syntax verification: legal XML, correct element nesting
Some Semantic verfication: NULLness, cardinality, local ID reference
If connect real database for validation (-v 1 option), it also will do the reference Validation

3.2.1 Typical Chado XML errors XORT Validator to detect:
3.2.1.1: Nest error:

A. Incorrect element of root

```
<chado>
  <uniquename>CG3321</uniquename>
  ….
</chado>
```

here "uniquename" is not database table and nest as child of root element

B. Link table element appear before column table elemnt

```
<feature>
  <featureloc>
    ….
  </featureloc>
  ….
</feature>
```

Here featureloc should appear AFTER all columns of table feature.

C. nest without primary/foreign or table/column relationship

```
<feature>
  <uniquename>CG33110</uniquename>
  <accession>…</accession>
  ….
</feature>
```

here accession is not column of table feature

<feature>
<uniquename>…</uniquename>
<type_id>
<dbxref>…</dbxref>
</type_id>
…
</feature>
type_id refer to cvterm insert to dbxref

D. wrong order of nesting
<feature_id>5<feature>...

E. <feature_id><feature>5
Here 5 will never be send to characters since the order of SAX header will be start_element, chararcters, end_element

F. <feature_id><feature>5
here 5 will never be send to characters since the order of SAX header will be start_element,chararcters, end_element

3.2.1.2. Object reference error

A. multiple returns for object reference

B. refer to not defined object

<chado>
<cvterm>
<name>gene</name>
<cv_id>SO</cv_id>
</cvterm>
…
</chado>
Here try to use “SO” without pre-defined.

C. more than one columns try to refer to same one
<feature>
…
<feature_relationship>
<type_id>..</type_id>
<feature_relationship>
</feature>
Here both feature_relationship.object_id and feature_relationship.subject_id are missed, and both try to take up-level (feature.feature_id). It is wrong to leave ambiguity.

3.2.1.3. Attribute error

A. Look up record which NOT in DB yet.

B. mis-op for update record

C. update which will cause database unique constraint error

3.2.1.4. Missing require element

```
<feature>
   <uniquename>…</uniquename>
   <organism_id>…</organism>
</feature>
```

3.2.15. Others

dumplicate cols for same table:

```
<feature>
  <uniquename>CG12345</uniquename>
  <uniquename>CG00000</uniquename>
```

3.3 Loader

For 99% Chado XML pass the validation, it can be loaded into database using loader. One thing need to point out, here loader is more efficient for update loading. For bulk load, XORT loader may not be best options due to various checkup during loading.

3.4 Dumper

Which export database object into XML file. By default, **Default behavior: given an object class and ID, dump all direct values and link tables, with refs to foreign keys.** For more complicated use case, it can be driven by dumpspec, a XORT XMLized query, to guide the behavior of dumper. See dumpspec section for more details.

3.5.XORTDiff

4. Dumpspec Specification:

Here is a proposal for a dumpspec syntax (DTD) and semantics. (Could also be extensions to main chado DTD; however I don't see a need to mix dumpspecs and other kinds of transaction info in same file right now.) The elements and nesting structure are same as current schema-driven DTD with a few extensions as described below.

The default dumper behavior when dumping an object (ie a main table row) is to dump all the columns and all linktable rows that join to the primary key. Foreign key columns in the maintable or linktables will generate object references (global ID, local ID or lookup). This default behavior can be modified by providing an XML dumpspec.

There are three new attributes:
* test controls when expansion occurs, and applies to column and linktable elements
* dump controls what is included in an expansion, and applies to table elements.
The new element OR can be nested inside any column element to create sets of acceptable values.
* limit will control how many rows to be returned.

<cvterm dump=”cols” limit =”2”/>
will be interpreted as:
select * from cvterm limit 2

Test operation value could be ‘eq|ne|gt|ge|lt|le|like|nl|yes|no|in|nti’ (means =, <>, >, >=, <, <=, like, not like, =, <>, in, not in, respectively), those should apply to column element. For link table, two operations apply to link-table column: yes, no

ms/mi/ns/ni (match, case sensitive/match, case insensitive/not match, case sensitive/not match, case insensitive). This will be DBMS specific
<feature>
<uniquename test=”ms”>^CG12345</uniquename>
</feature>
will be:
select … from feature where uniquename ~ ‘^CG12345’

<feature>
<uniquename test=”eq”>CG12345</uniquename>
<featureloc test=”yes”>
<srcfeature_id test=”eq”><feature><uniquename test=”in”>(‘2R’, ‘2L’,’3R’, ‘3L’, ‘4’, ‘X’)</uniquename></feature>
</featureloc>
</feature>
will be select … from feature feature_0, featureloc featureloc_0 where feature_0.uniquename=’CG12345’ and exists (select * from featureloc featureloc_0 where feature_0.feature_id=featureloc_0.feature_id)
here featureloc have Two foreign keys(feature_id, srcfeature_id) refer to parent table feature. If MORE than ONE keys WITHOUT explicit constraint, then throw error. For instance:

<feature>
<uniquename test=”eq”>CG12345</uniquename>
<feature_relationship test=”exist”/>
</feature>
here both feature_relationship.object/subject are foreign key WITHOUT explicit constraint and both refer to parent table feature. So throw out as error.

Another example is cvtermprop
<cvterm dump=”cols”>
<cvtermprop dump=”cols”/>
</cvterm>

here it may pickup cvtermprop.type_id as join key , it should explicitly exclude type_id for joining as followings:
<cvterm dump=”cols”>
<cvtermprop dump=”cols”>

```
      <type_id/>
  </cvtermprop>
</cvterm>


<feature>
   <uniquename test=”eq”>CG12345</uniquename>
   <feature_relationship test=”exist”>
    <object_id test=”ne”>
     <feature>
       <uniquename></uniquename>
     </feature>
    </object_id>
  </feature_relationship>
  </feature>
```

Nearest inner parent operation rule:

```
   <feature>
     <type_id test=”eq”>
       <cvterm>
         <name >gene</name>
       </cvterm>
     </type_id>
   </feature>
```

here <name> without operation, by default, it will inherit test=”eq” from <type_id> so it will be equivalent be:

```
<feature>
      <type_id test=”eq”>
        <cvterm>
          <name test=”eq”>gene</name>
        </cvterm>
     </type_id>
   </feature>
```

select … from feature feature_0, cvterm cvterm_0 where (cvterm_0.name=’gene’) and feature_o.type_id=cverm_0.cvterm_id.

However,

```
<feature>
      <type_id >
        <cvterm>
          <name test=”ne”>gene</name>
        </cvterm>
     </type_id>
```

</feature>

will NOT join feature/cvterm since here the join (type_id) miss test condition.

Certainly you can override the default operation as following:

```
<feature>
     <type_id test="eq">
       <cvterm>
         <name test="ne">gene</name>
       </cvterm>
     </type_id>
</feature>
```

will be:

select … from feature feature_0, cvterm cvterm_0 where (cvterm_0.name<>'gene')
and feature_o.type_id=cverm_0.cvterm_id

Remind that, here negation operation apply to <name> as whole, in other word, the following operation will be:

```
<feature>
     <type_id test="eq">
       <cvterm>
         <name test="ne"><or>gene</or><or>mRNA</or></name>
       </cvterm>
     </type_id>
</feature>
```

select … from feature feature_0, cvterm cvterm_0 where feature_0.type_id=cvterm_0.cvterm_id and NOT (cvterm_0.name='gene' OR cvterm_0.name='mRNA')

which will be the same as:

select … from feature feature_0, cvterm cvterm_0 where feature_0.type_id=cvterm_0.cvterm_id and (cvterm_0.name<>'gene' AND cvterm_0.name<>'mRNA')

Warning, if you set foreign key operation to "ne|no", you may end up with out-join(?):

```
<feature>
     <type_id test="ne">
       <cvterm>
         <name test="eq">gene</name>
       </cvterm>
     </type_id>
</feature>
```

will be
select .. from feature feature_0, cvterm cvterm_0 where feature_0.type_id <>cvterm_0.cvterm_id and cvterm_0.name='gene'

this will return thousand duplicate records …..

```
<feature>
  <name test="ne">mia:1</name>
  <name test="like"><or>mia%</or><or>pia%</or></name>
</feature>
```

will be:
select … from feature feature_0 where (name like 'mia%' or name like 'pia%') AND name ne 'mia:1'

It will be trick for multiple-foreign test constraint:

```
<feature>
    <type_id test="eq">
      <cvterm>
        <name test="ne"><or>gene</or><or>mRNA</or></name>
      </cvterm>
    </type_id>
    <type_id test="eq">
      <cvterm>
        <name test="eq">protein</name>
      </cvterm>
    </type_id>
</feature>
```

will be:
select … from feature feature_0, cvterm cvterm_0, cvterm cvterm_1 where feature_0.type_id=cvterm_0.cvterm_id and (cvterm_0.name <> 'gene' and cvterm_0.name <>'mRNA') and feature_0.type_id=cvterm_1.cvterm_id and

in constraint: the value should be properly set based on the data type

```
<feature>
  <organism_id test="in">(1, 214)</organism>
</feature>
```

will be:
select … from feature feature_0 where feature_0.organism_id in (1, 214)

```
<feature>
  <uniquename test="in">('FBgn000123, 'FBgn000124')</uniquename>
</feature>
```

will be:
select … from feature where uniquename in ('FBgn000123, 'FBgn000124')

A more complicated example:

```
<feature dump="cols">
   <type_id test="eq">
      <cvterm>
         <name test="eq"><or>exon</or><or>protein</or></name>
      </cvterm>
    </type_id>
    <seqlen test="ge">100</seqlen>
    <name test="like">mia-P%</name>
    <organism_id test="in">(1, 214)</organism_id>
    <uniquename test="in">('FBpp0088355', 'FBpp0088351')</uniquename>
    <featureloc test="yes">
      <fmin test="gt">100</fmin>
      <srcfeature_id test="eq">
        <feature>
          <uniquename test="in">('2L', '2R', '3L',  '3R','4','X')</uniquename>
        </feature>
      </srcfeature_id>
    </featureloc>
  </feature>
```

will be:

```
select feature_0.name, … feature_0.is_obsolete
from cvterm cvterm_0 , feature feature_0
where
exists (select * from feature feature_1 , featureloc featureloc_0
   where
    featureloc_0.fmin>100
    and feature_1.uniquename in ('2L', '2R', '3L', '3R','4', 'X')
    and feature_0.feature_id=featureloc_0.feature_id
    and featureloc_0.srcfeature_id=feature_1.feature_id)
and feature_0.organism_id in (1, 214)
and (cvterm_0.name='exon' or cvterm_0.name='protein')
and feature_0.seqlen>=100
and feature_0.name like 'mia-P%'
and feature_0.uniquename in ('FBpp0088355', 'FBpp0088351')
and feature_0.type_id=cvterm_0.cvterm_id
```

Inside a column element, the attribute “test” means that the applicability of the next outer table element is conditional on a successful match of this field. The contents of the test column element may be:
-for primitive datatype column, a primitive data value.
(Allow wildcards for strings)
-for foreign key column, an object reference (global ID, local ID or lookup).

-more than 1 of the above, each wrapped in an <or>v1</or> element, to indicate a list of acceptable values.
-an empty element to indicate null
Test=no|ne is used to indicate negation of the test, i.e., the value must not match.
If a table has more than one test element inside it, all of them must succeed - implicit AND of column constraints.

Example: The SQL generated for a table dumpspec of the form:
<table>
  <col1 test=yes><or>v1.1</or><or>v1.2</or></col1>
  <col2 test=no>v2.1</col2>
</table>
would be something like
  select primary_key
  from table
  where (col1 = v1.1 OR col1 = v1.2) AND not col2 = v2.1
or maybe
  where col1 in (v1.1,col1 = v1.2) AND col2 = v2.1
which should be equivalent, though maybe not in performance.

In addition, tests are allowed on linktables, where again they control the applicability of the table element that contains them. Within a table element test can take the following values:
  Test=any succeeds if there exists an instance of the linktable that satisfies the constraints.
  Test=all succeeds if all instances of the linktable satisfy the constraint. Not clear if this is needed. (True if there aren't any.)
  Test=none succeeds if no instances of the linktable satisfy the constraint.

The dump attribute is permitted within table elements; it determines whether and how much of an object to dump.
  Dump=all: like select *. Dump all cols of table, and all linktables; generate refs for all foreign keys in columns or linktables. This is the default dumper behavior for a root object. Detailed behavior can still be modified by nested dumpspec elements.
  Dump=cols: like all, except don't dump linktables.
Dump=ref: generate a ref for an object. Only tests can be nested inside; anything else makes no sense.
Dump=select: dump only what is specified by nested elements. As in SQL, if you do not use select * you must ask for what you want explicitly. To specify a column to be dumped, include it as an empty element. Linktables can also be specified as empty elements, in which case you get the default behavior: column data plus foreign key refs. Test columns should not be dumped by default in a select; if you want them dumped list them twice, once with test and once without. Eg.
<feature dump=select><name/></feature>
would dump just the name of a feature. Note that using select it is possible to dump stuff that the loader would not be able to read, because keys could be incomplete.

Dump=no: don't dump what matches. This might be used inside a Dump=yes object to prevent dumping of specific link tables which would otherwise be dumped.

If a table is specified without a dump value, the default is all.

Example: Central Dogma. To dump a gene along with its transcripts, their CDSs, and their proteins, you would do something like this:

```
<feature>
  <type> ref to term "gene"</type>
  <feature_relationship>
      <subjfeature_id>
        <feature>
            <type test=yes> ref to term "transcript"</type>
            <feature_relationship>
               <subjfeature_id>
                  <feature>
                       <type test=yes>ref CDS</type>
                       <feature_relationship>
                        <subjfeature_id>
                           <feature>
                               <type test=yes>ref protein</type>
                           </feature>
<close everything>
```

The semantics here is that all the default information for a gene is printed, but transcripts are expanded because they match the outermost feature_relationship dumpspec. Other feature relationships will be dumped according to the default behavior, but not expanded. If we did not want to dump other feature relations, we could either put dump=cols in the top level gene feature, in which case linktables are not dumped by default, or else we could add

```
   <feature_relationship dump=no>
      <subjfeature_id>
<feature>
     <type test=no> ref to term "transcript"</type>
```

inside the gene to match all other feature_relationships and prevent them being dumped.

To dump all annotations on a contig, using the central dogma model for genes and just dumping the feature for everything else, do:

```
<feature dump=cols>
    <featureloc>
        <feature_id>
            <feature>
                 <type test=yes> ref to gene term</type>
              etc. - rest of central dogma dumpspec, as above
            </feature>
```

```
            </feature_id>
          </featureloc>
          <featureloc>
               <feature_id>
                    <feature dump=cols>
                         <type test=no> <or>ref to gene term</or>
                           <or>ref transcript</or>
                          <or>exon</or>
                          <or>CDS</or>
                        </type>
                     </feature>
               </feature_id>
          </featureloc>
     </feature>
```

5. Use Case
5.1 Central Dogma of gene annotation in Chado XML (without macro)
5.2. Alignment evidence in Chado XML format (without macro)
5.3. Typical dumpspec to dump Drosophila sequence data in Chado XML for Apollo
5.4. Typical dumpspec to dump Drosophila sequence data in Chado XML for Indiana web
5.4.1. db ddl and view ddl setup
5.4.2. generate ddl.properties
5.4.3. how to exclude the view/function from dumpspec
5.4.4. _sql: invalid sql,

```
<feature dump="cols">
     <_appdata name="genus">$1</_appdata>
     <_appdata name="species">$2</_appdata>
     <_sql>
       select distinct on ( feature_0.uniquename )
            feature_0.timeaccessioned, feature_0.name,
            feature_0.timelastmodified, feature_0.residues,
            feature_0.dbxref_id, feature_0.is_obsolete,
            feature_0.feature_id, feature_0.uniquename, feature_0.seqlen,
            feature_0.md5checksum, feature_0.organism_id,
            feature_0.type_id, feature_0.is_analysis
         from feature feature_0, feature feature_1, feature feature_2, cvterm cvterm_0,
cvterm cvterm_1,
             analysisfeature analysisfeature_0, analysis analysis_0, organism organism_0,
             featureloc featureloc_0, feature_relationship feature_relationship_0
        where cvterm_0.name = 'match' and cvterm_1.name = 'match'
          and organism_0.genus='$1' and organism_0.species='$2'
          and featureloc_0.rank=0
          and analysis_0.program in ('sim4','sim4tandem','splign')
          and feature_0.is_obsolete='f' and feature_0.is_analysis='t'
```

```
        and feature_1.is_obsolete='f' and feature_1.is_analysis='t'
        and feature_0.feature_id=feature_relationship_0.object_id
        and feature_relationship_0.subject_id=feature_1.feature_id
        and feature_0.feature_id=analysisfeature_0.feature_id
        and analysisfeature_0.analysis_id=analysis_0.analysis_id
        and feature_0.type_id=cvterm_0.cvterm_id
        and feature_1.type_id=cvterm_1.cvterm_id
        and feature_1.feature_id=featureloc_0.feature_id
        and featureloc_0.srcfeature_id=feature_2.feature_id
        and feature_2.organism_id=organism_0.organism_id limit 5
    </_sql>
</feature></chado>
```

5.5. use XORT to identify all related information. For example, to see if an gene is orphan or not(the only record is feature selt ?)

```
<chado>
<uniquename test="eq"> gene_uniquename_here</uniquename>
  <type_id test="eq"><cvterm><name>gene</name></cvterm></type_id>
</feature>
</chado>
```

then use xort_dump.pl to dump it from database, then look at the output. If only feature selt (no other table refer to feature, like feature_relationship, featureprop etc), than it is orphan.

On the other hand, we can merge two genes together if we can find out all related information for this gene, as below.

5.6 use XORT to merge to two genes

for example, if we want to merge gene 1 and gene2 (with uniquename as FBgn0000001 and FBgn0000002, respectively) into gene1.

Create this dumpspec:

```
<feature dump="all">
  <uniquename test="eq"><or> FBgn0000001</or><or>
FBgn0000002</or></uniquename>
  <type_id test="eq"><cvterm><name>gene</name></cvterm></type_id>
</feature>
```

use XORT xort_dumper.pl to dump out all related information for those two genes, then edit the output, simple change the second gene of this line:

<uniquename> FBgn0000002</uniquename> to
<uniquename> FBgn0000001</uniquename>

load it back to database, delete gene FBgn0000002 from database. That's it.

5.6. use XORT to identify all related information.

6. Encoding and entity declaration

Three types of data involve our discussion:

a. 5 XML predefined name character entities:
   < <
   > >
   & &
   " “
   ' ‘

b. greeks
   alpha for α
   …..

c. <up></up> vs <down></down> for superscript vs subscript

The XML::Parser model, derived from the expat model, is that no matter what the original document encoding is, the data forwarded to the calling software will be in UTF-8(Unicode). (The first 128 characters of the ASCII encoding happen to be identical to UTF-8, ASCII is a subset of ISO-8859-1 which is subset of Unicode).

It would be easiest in our database/data if all entities would be in plain text, the only problem is plain text can’t eliminate 100% ambiguity, say ‘dialpha’ should be combination of ‘dial’ and ‘pha’, or ‘di’ with Greek ‘alpha’. On the other hand, it will be trouble to deal with Greek for lots of software, for instance, how to store Greek in plain text file ?

Here we propose that, all feature.name, synonym.name (typical place to store symbols) will be in plain text, synonym.synonym_sgml will store ‘SGML’ format. (how about featureprop.value ? This is another typical place to have tons of Greeks ?). So some typical data will look like this:

Feature.name	s.name	s.synonym_sgml
&	&	&
dialpha	dialpha	di&agr; (or raw data ‘α’)
Tub85D[n]	Tub85D[n]	Tub85D<up>n</up>

Special situation for FlyBase before we start the discussion of entity declaration/encoding:

1. all chado instance created with encoding as default (ASCII) (remember ASCII is subset of Unicode).
2. chadoXML from Aubrey use ‘double entities’ for Greeks, say ‘&agr;’ for ‘α’.
3. XORT load ‘&agr;’ into chado and store as ‘&agr;’

4. XORT convert back to ‘double entities’ as ‘&agr;’ when dump out to Indiana This cause trouble for Indian since they need to filter out and do some conversion.

In the world of entity declaration/Unicode:

For all data transfer between different sites, we use XML format, the entity declaration + encoding provide us a simple method to deal with those Greek characters. So all the XMLs will have header as:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FBchado [
<!ENTITY lt "&#60;">
<!ENTITY gt "&#62;">
<!ENTITY amp "&#38;">
<!ENTITY apos "&#39;">
<!ENTITY quot "&#34;">
<!ENTITY Agr "&#913;">
<!ENTITY Bgr "&#914;">
<!ENTITY Ggr "&#915;">
<!ENTITY Dgr "&#916;">
<!ENTITY Egr "&#917;">
<!ENTITY Zgr "&#918;">
…..
```

so

```
<feature>
  <name>A&agr;</name>
```

will properly convert into ‘Aα’

A few problems here:

1. Currently all data from Cambridge still use ‘double entity’ for Greek characters as:

```
<feature id=”A&amp;agr;”>     ---- notice here also use entity name in attribute
  <name>A&amp;agr;</name>
  …..
</feature>
<feature_relationship>
  <object_id>A&amp;agr;</object_id>
…..
```

with those ‘double entities’, the XML entities declaration will essentially have NO effect at all.

So XORT will convert it and store in chado as ‘A&agr;’
And dump out as:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE FBchado [
<!ENTITY lt "&#60;">
<!ENTITY gt "&#62;">
<!ENTITY amp "&#38;">
<!ENTITY apos "&#39;">
<!ENTITY quot "&#34;">
<!ENTITY Agr "&#913;">

     <feature id=”A&agr;”>     ---- notice here also use entity name in attribute
      <name>A&agr;</name>
      …..
   </feature>
  <feature_relationship>
    <object_id>A&agr;</object_id>
```