Appl Bioinformatics 2005; 4 (1): 13-24
1175-5636/05/0001-0013/$34.95/0

© 2005 Adis Data Information BV. All rights reserved.

# A Structured Interface to the Object-Oriented Genomics Unified Schema for XML-Formatted Data

*Terry Clark*,[1] *Josef Jurek*,[2] *Gregory Kettler*[2] and *Daphne Preuss*[2]

1 Department of Electrical Engineering and Computer Science, The University of Kansas, Lawrence, Kansas, USA
2 Department of Molecular Genetics and Cell Biology, The University of Chicago, Chicago, Illinois, USA

## Abstract

Data management systems are fast becoming required components in many biology laboratories as the role of computer-based information grows. Although the need for data management systems is on the rise, their inherent complexities can deter the full and routine use of their computational capabilities. The significant undertaking to implement a capable production system can be reduced in part by adapting an established data management system. In such a way, we are leveraging the Genomics Unified Schema (GUS) developed at the Computational Biology and Informatics Laboratory at the University of Pennsylvania as a foundation for managing and analysing DNA sequence data in centromere research projects around *Arabidopsis thaliana* and related species. Because GUS provides a core schema that includes support for genome sequences, mRNA and its expression, and annotated chromosomes, it is ideal for synthesising a variety of parameters to analyse these repetitive and highly dynamic portions of the genome. Despite this, production-strength data management frameworks are complex, requiring dedicated efforts to adapt and maintain. The work reported in this article addresses one component of such an effort, namely the pivotal task of marshalling data from various sources into GUS. In order to harness GUS for our project, and motivated by efficiency needs, we developed a structured framework for transferring data into GUS from outside sources. This technology is embodied in a GUS object-layer processor, XMLGUS. XMLGUS facilitates incorporating data into GUS by (i) formulating an XML interface that includes relational database key constraint definitions, (ii) regularising traversal through that XML, (iii) realising automatic processing of the XML with database key constraints and (iv) allowing for special processing of input data within the framework for automated processing. The application of XMLGUS to production pipeline processing for a sequencing project and inputting the *Arabidopsis* genome into GUS is discussed. XMLGUS is available from the Flora website (http://flora.ittc.ku.edu/).

The pronounced rise in computational models applied to molecular biology brings with it requirements for data management systems. Data integration from sundry sources adds to the requirement for management solutions, as shown by the number of databases of molecular biology information[1] and sequence data that are accumulating at exponential rates at central distribution hubs.[2] Although national centres provide central distribution of public domain data along with analysis services, laboratories generating data have site-specific data management requirements.[3-7] In domain-specific areas such as molecular biology, a core database schema can be used to address common requirements at various sites. The Genomics Unified Schema (GUS) is one such open-source, object-oriented relational database centred on a schema for DNA and protein sequence data.[8] GUS was designed to warehouse and integrate sequence data and annotations from various heterogeneous sources under a common schema. The advanced schema and support make GUS an attractive foundation for data management in molecular biology applica-

**Table I.** Entries for the table DoTS::NASequence. Attribute values correspond directly to the XML in figure 1, except for the primary key sequence_id, which is usually generated automatically

| Attribute | Type |
|---|---|
| sequence_id | number |
| sequence_name | varchar |
| sequence | clob |
| description | varchar |
| sequence_type_id | number |

tions.[9] This article presents a software-engineering approach to input data into GUS using a tailored XML format. This work contributes to the management of molecular biology data by simplifying the complex process of input module development, and by providing a basis for automation of the schema-dependent components of the framework.

The GUS approach of importing data from an outside source uses object-layer *plugins*, programs that extend and interact with object-layer functionality. The input data are obtained from a wide variety of sources and in nearly as many formats. For example, data may be warehoused from the Protein Data Bank, TIGR XML-formatted genome annotations, BLAST® output and from National Center for Biotechnology Information (NCBI) taxonomies, to name a few. Plugins are developed around their target input-data idiosyncrasies, by individuals with various software development backgrounds, to address data management requirements. The resulting stylistic variation in plugin design can complicate use and maintenance. Moreover, separate plugins around each data source can impede the data incorporation process, a topic we discuss in some detail in this article.

The XMLGUS framework structures the input processing and nearly eliminates the need for input-specific plugins by way of a standard XML description of input data. The standard XML (called GUS XML) is processed by a generic processing module.[1] The framework also sets the stage for automatic generation of database-dependent components. Since imported data typically correspond directly to relational schema attributes, a natural development is a structured input interface to GUS using automated processing.[2] Key considerations concerning the interface design were its architecture and the requirements it placed on users. A pivotal consideration was where transformations from input data to the canonical XML occur. These matters are discussed in detail in the sections that follow.

## Genomics Unified Schema (GUS) and Data Input

This section briefly describes GUS and the strategy used to populate the database schema. The central development and management of GUS occurs at the Computational Biology and Informatics Laboratory at the University of Pennsylvania (http://www.gusdb.org/). Concepts behind GUS, data warehouses and trade-offs with respect to other data integration approaches are discussed in an article by the original developers.[8] At the core of GUS is a relational database with hundreds of tables organised into collections of logically related tables (or *namespaces*). An object layer encapsulates the relational database such that namespaces and tables are associated with object-oriented Perl packages. For example, SRes::ExternalDatabase is the class for table ExternalDatabase in namespace SRes.[3] Sequences and annotations are stored in the DoTS namespace. Other namespaces have information for (i) gene expression and regulation; (ii) shared principles that organise application data with ontologies, controlled vocabularies, metabolic pathways and the like; and (iii) workflow and data warehouse management. The Perl modules that correspond to objects are generated automatically from the schema and database key constraints. However, the input and output plugins to the GUS object layer are written manually.

Input data typically map directly into GUS objects and the schema where, in the latter step, primary-foreign relationships are resolved. For example, consider the sequence and abbreviated description presented in XML in figure 1. Assume that the sequence alpha will be inserted into the sequence table DoTS::NASequence[4] of GUS shown in table I. A plugin to input such data into GUS will create objects, read the input, assign values to appropriate object fields and commit the data to the database. In some instances it will be necessary to process the input data (see

1 In this article we refer to XMLGUS and plugins, where the term *plugin* refers to the standard hand-written approach and XMLGUS represents a framework. The XML processing engine component of the XMLGUS framework functions as a plugin.

2 The term *attribute* is overloaded because of its use in XML and relational databases. An *XML attribute* is a name-value pair within an XML element; a *relational attribute* is a component in a relational table.

3 The double colon **::** is used in two contexts in this article. In this sentence, the usage is the standard Perl package delimiter.[10] In the other context, it is used to delimit the associated GUS namespace and object. For example, in DoTS::NASequence, NASequence is a view of the GUS table NASequenceImp within the 'Database of Transcribed Sequences' namespace; in DoTS::NASequence::Description, Description is an attribute of the table NASequence.

4 The GUS relational schema can be browsed from the GUS website (http://www.gusdb.org/).

© 2005 Adis Data Information BV. All rights reserved. 

```
1  <?xml version = "1.0"?>
2  <gus>
3    <dots_nasequence depth=0>
4       <fragmentname>alpha</fragmentname>
5       <sequence>
6         gttttctkgctcgatatgtctttcaagcttcggtacatacaatttggagagagcgaaatggaagaagaca
7         tggtgaaacwccaaacccagcagctcgactaatccggtggataganaaacaaattcgaaatcaactttca
8         gcaatcaracttatgggagatagaagatatgacaagggactccaaatttggtttgcatcaagagcttaat
9       </sequence>
10      <description>random shotgun read from Arabidopsis thaliana</description>
11      <dots_sequencetype fkobj="dots::nasequence" depth=1>
12         <name>DNA</name>
13      </dots_sequencetype>
14      <sequencetypeid pkobj="dots::sequencetype" key="sequence_type_id"/>
15   </dots_nasequence>
16 </gus>
```

**Fig. 1.** XML describing the sequence alpha for insertion into the database. Element tags correspond directly to relational tables and attributes. XML attributes are used for describing relationships between tables.

the section titled Tailoring Semantics). These input operations appear straightforward; however, plugin logic can be considerably complex. The effort required to usher data into GUS is at times considerable.[5]

Input processing usually requires resolution of foreign keys from candidate keys.[6] Consequently, the plugin must determine and resolve key dependencies when incorporating data into GUS. For example, in figure 1, the sequence alpha is of type DNA. DNA in this context is a term in a controlled vocabulary accompanying the imported data and is a value for the *ad hoc* candidate key name in the table DoTS::SequenceType (table II); the table DoTS::NASequence (table I) requires a sequence type in a non-null attribute, sequence_type_id, the foreign-key reference to the table DoTS::SequenceType. (The corresponding primary key happens to have the same name.) To resolve the foreign key, the input module instantiates a DoTS::SequenceType object with the appropriate sequence type to obtain the primary key for the new DoTS::NASequence object.

## The XMLGUS Approach

XMLGUS automates the data input tasks described above with a declarative framework coupled to a processing module working as a GUS plugin. We chose XML as the standard input format because of its descriptive capabilities and the research and development surrounding it. An XML document consists of elements and attributes, along with an optional document type definition (DTD) describing the document structure. The interested reader is referred elsewhere for a review of XML concepts.[13]

### Overview

The XMLGUS plugin consists of a processor encoded as an object-oriented Perl module, a context-free grammar and optional user-defined functions. Figure 2 is a schematic of the components. XMLGUS glues the structured XML input to the GUS object layer and relational database with a correspondingly structured interface between these components. Although most data mapped into GUS are a direct assignment to relational tables, exceptions to direct mapping occur. This motivated us to develop a framework with the facility for incorporating non-default processing with the default automated processing.

For XML processing, XMLGUS uses XML::YYLex (http://home.debitel.net/user/boesswetter/xml_yylex/) with the Berkeley YACC[14] compiler generator Perl-*byacc* (http://packages.debian.org/unstable/devel/perl-byacc.html) in combination with an

**Table II.** Entries for the table DoTS::SequenceType. Attribute values correspond directly to the XML in figure 1, except for the primary key sequence_type_id, which is generated automatically. The attribute name is a candidate key used to derive the primary key. Resolution of primary keys from candidate keys in this manner is common in GUS

| Attribute | Type |
| --- | --- |
| sequence_type_id | number |
| name | varchar |

5 Examples of the issues encountered can be appreciated directly with a survey of the GUS email archives.[11]

6 Generally, a *candidate key* is one or more attributes that together uniquely identify at most one record in a given table.[12] Candidate keys that have been designated in the database are called *primary keys*.

© 2005 Adis Data Information BV. All rights reserved.

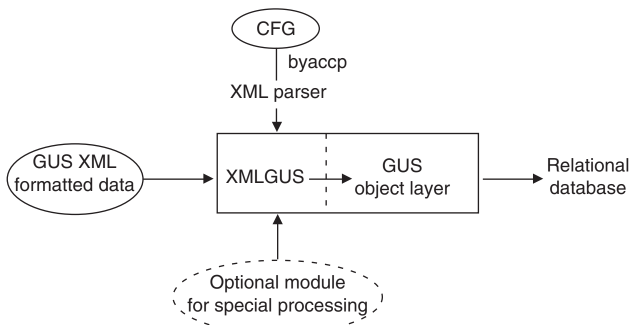


**Fig. 2.** Schematic of the XMLGUS plugin. Input data, such as chromosome sequence data, are formatted into GUS XML, as described in the text. The user of the plugin provides the circled components: context-free grammar (CFG), the XML input and an optional module for any case-specific processing.

XML::DOM processor (http://www.w3.org/DOM/). Other XML-to-relational-database tools are discussed in the section titled Related Work. XML::DOM provides the lexical analysis for the parser. The YACC processor regularises traversal over the input by way of a structured grammar and parse that triggers user-defined actions.[14] YACC with Perl actions interfaces the GUS object layer, the latter consisting of object-oriented Perl modules.

The parser encourages orderly and regular processing of the XML with depth-first, left-to-right processing of the underlying document object model (DOM) tree.[15] The grammar follows directly from the relational database schema; relational key constraints are handled during the parse. The XMLGUS framework operates as follows:

1. The user defines a grammar and the XML, with both corresponding to relational database representations of GUS objects. For example, the grammar fragment[7] in figure 3 corresponds to the XML in figure 1, and both correspond to the schema represented in table II and table I, respectively.
2. Prior to XMLGUS processing, input data are formatted into GUS XML, where the XML tag elements correspond to the terminals in the grammar, which in turn correspond to GUS table names and relational database attributes as described in step 1. Key constraints are declared where needed in the XML using XML attributes (see sections titled GUS XML, and Key Constraints).
3. The grammar definition is input to the program *byaccp*, which produces the parser; the generated parser is an input to the XMLGUS processor.
4. If special purpose processing is needed, the methods are written and named according to a convention used by the XMLGUS dispatcher method (see section titled XMLGUS Grammar). These methods are triggered into action by non-null fields in the parameter list to the dispatcher.
5. XMLGUS is executed by providing the *byacc*-generated parser on the command line with the input XML and any other arguments. Reductions during the XML parse take the action of a single call to the dispatching routine. All calls to the dispatching routine are through a simple template, for example in figure 3, lines 11−19.

In this way, the parser regularises input processing and simultaneously verifies input syntax.[8,9]

## GUS XML

The GUS XML in figure 1 corresponds in a straightforward way to the GUS object DoTS::NASequence (table I). The XML elements nested in DoTS::NASequence correspond to either a relational attribute or a relational table. The opening and closing XML tags dots_nasequence in figure 1 (line 3 and line 15) enclose the data intended for the relational table DoTS::NASequence. Other XML tag names follow the same logical naming scheme. The convention where XML element tag names correspond to the relational tables is not essential; however, it aids readability and highlights the correspondence between the schema and XML. In fact, it is the XMLGUS grammar that determines the correspondence between XML elements and relational table attributes. This occurs by string names embedded in the parameter list of the calls to a single dispatching routine on actions taken during the parse (see section titled XMLGUS Grammar).

Pairwise relationships between primary and foreign keys are expressed with the XML attributes fkobj and pkobj, which point the foreign-key object to the primary-key object. These XML attributes trigger the XMLGUS processor to resolve foreign keys by instantiating and fetching data for referenced objects. For example, the object for the primary key sequence_type_id is determined from a fetch of the object DoTS::SequenceType having attribute name equal to value DNA, as indicated by lines 11−13 in the XML of figure 1. The primary key is named in the same XML at line 14 with the attribute key. Finally, the foreign key sequence_type_id is assigned the value of the primary key by the

7 Grammars and code for the examples in this article are available on the Flora website (http://flora.ittc.ku.edu/).

8 The processing order of XML is identical for all input with the XMLGUS processor, namely, left to right, bottom to top in the XML parse tree. This regularity naturally structures control flow, even those that are authored by different people for different purposes.

9 XML DTDs can also be used to verify XML syntax.

© 2005 Adis Data Information BV. All rights reserved.

```
1  XMLDOCUMENT:  gus BODY _gus
2                { return 1; };
3
4  BODY:         BODY_ELEMENT | BODY BODY_ELEMENT;
5
6  BODY_ELEMENT: DOTS_NASEQUENCE | DOTS_NAFEATURE;
7
8  DOTS_NASEQUENCE: dots_nasequence DOTS_NASEQUENCE_SET _dots_nasequence
9                {
10                 XMLGUS::process_xml_rule(
11                       undef, "Specialized",
12                       "DoTS::NASequence",
13                       $2->getNodeValue,
14                       $1->getAttribute("pkobj"),
15                       $1->getAttribute("fkobj"),
16                       $1->getAttribute("key"),
17                       $1->getAttribute("depth")
18                       );
19               };
20
21 DOTS_NASEQUENCE_SET:  DOTS_NASEQUENCE_ATT |
22                       DOTS_NASEQUENCE_SET DOTS_NASEQUENCE_ATT;
23
24 DOTS_NASEQUENCE_ATT:  DOTS_NAFEATURE |
25               DOTS_NASEQUENCE_DESCRIPTION |
26               DOTS_NASEQUENCE_SEQUENCE_TYPE_ID;
27
28 DOTS_NASEQUENCE_DESCRIPTION: description TEXT _description
29               {
30                     XMLGUS::process_xml_rule(
31                       undef, undef,
32                       "DoTS::NASequence::description",
33                       $2->getNodeValue,
34                       $1->getAttribute("pkobj"),
35                       $1->getAttribute("fkobj"),
36                       $1->getAttribute("key"),
37                       $1->getAttribute("depth")
38                       );
39               };
40
41 DOTS_NASEQUENCE_SEQUENCE_TYPE_ID: sequencetypeid _sequencetypeid
42               {
43                     GUS::Common::Plugin::XMLGUS::process_xml_rule(
44                       undef, undef,
45                       "DoTS::NASequence::sequence_type_id",
46                       $1->getAttribute("dummy"),
47                       $1->getAttribute("pkobj"),
48                       $1->getAttribute("fkobj"),
49                       $1->getAttribute("key"),
50                       $1->getAttribute("depth")
51                       );
52               };
```

**Fig. 3.** Portion of *byaccp* grammar for table DoTS::NASequence.

description in the XML, also at line 14 in figure 1, by way of the XML element sequencetypeid. The data are shipped to the dispatcher by the production DoTS_NASEQUENCE_SEQUENCE_TYPE_ID (figure 3, lines 43−51). This example illustrates how objects and attribute relationships are fully defined in the XML.[10]

10 That primary-key to foreign-key object relations are defined in the XML allows data-driven use of the schema. The original intent of GUS foreign-key associations is at times unclear. Moreover, the ability to use attributes in problem-specific ways extends the applicability of the schema and retains the spirit of current use.

© 2005 Adis Data Information BV. All rights reserved.

Foreign- and primary-key relationships can also be expressed by nesting XML elements referring to tables. It is shown in the section titled Key Constraints that nesting alone is insufficient for expressing all key relationships, thereby calling for an alternative such as the fkobj and pkobj attributes used by XMLGUS. Persistent primary-key-containing objects can be useful when a single fetch of an object provides values for many foreign-key-containing objects. The application of fkobj and pkobj facilitates this simplification and performance gain (see section titled Performance).

For general XML, we assume some transformation to arrive at GUS XML, such as XSLT (XSL transformations).[16] Other non-XML formats can be translated to GUS XML with typically simple scripts. The plugin interface to the object layer is streamlined with this approach. Rather than one module for each format, the data supplier is responsible for formatting their data into GUS XML. Where a program generates data intended for GUS, GUS XML may be output directly. Routine transformations from other formats into GUS can be done modularly, separate from the object-layer components. This simplifies the plugin interface, and most input data can be processed by the same generic functions.

### XMLGUS Grammar

The XMLGUS grammar consists principally of variables and terminals associated with GUS XML elements. GUS XML tag names correspond to either relational *table names* or relational *table attribute names*. The XML elements determine the parse through the grammar, where XML content is mapped to GUS objects through actions taken when appropriate rules are reduced.

The *byaccp* grammar of figure 3 with 13 productions is a subset required to parse the XML in figure 1. A parse of that XML first reduces the following production:

DOTS_NASEQUENCE_DESCRIPTION →
description TEXT _description

where, by convention, *uppercase names* are variables and *lowercase names* are terminals. Terminals correspond to XML element tags with a leading *underscore* matching the closing tag. Elements are retrieved with the method getNodeValue(), and attributes with the method getAttribute(). The action for the reduction enters the XMLGUS module with the corresponding call to the dispatcher method process_xml_rule(). In figure 3, the string DoTS::NASequence::description informs XMLGUS about the namespace, object and attribute, respectively. The undef arguments indicate that special processing should not take place on either the first or second pass of processing.[11] The attribute depth informs the processor about nesting level in the XML for the purpose of committing an object hierarchy; objects are committed to the database when they occur at a depth of zero. The GUS object layer provides methods to construct, in effect, a tree of objects for the purpose of establishing primary- to foreign-key relationships.[12] Foreign-key references for objects associated in this way are automatically resolved in the object layer. This feature, as implemented in GUS, applies to primary-foreign key relationships declared in the database, where along with other restrictions, it is not a comprehensive key-resolution mechanism. For those objects defined in the object hierarchy, each object and its child objects − should any exist − are committed from the root object; by definition, the root object in GUS XML has level equal to zero. Since description is not involved in a key relationship, the attribute arguments in the process_xml_rule() parameter list would be undefined during processing.

The production DOTS_NASEQUENCE in figure 3 corresponds to the object DoTS::NASequence. The left-hand-side variable DOTS_NASEQUENCE_SET and the right-hand-side variables form a partial collection of productions for the object. The action for DOTS_NASEQUENCE is the second-to-last action taken in the parse with this grammar snippet; the return for rule XMLDOCUMENT is the last action. The XML tag dots_nasequence, a terminal in the grammar, occurs at the outermost level with a depth of zero. As noted above, zero depth triggers submission of the object hierarchy. As before, undef indicates that default processing is to be used, in this case for the first pass.[13] The second argument Specialized is more interesting. This name corresponds to the module with the special purpose routine with the default name DoTS_NASequence_02. The special purpose module might count the number of nucleotides in a sequence or manipulate version numbers, for example. An example of special purpose processing is given in the following section.

---

**11** XMLGUS makes two passes through the input XML. In the first pass, empty objects are created and queued for use in the second pass.

**12** A parent-child relationship where the parent contains a primary key referenced by the child is created by the method addChild() as in parentObj→addChild(childObj).

**13** In our work with XMLGUS, we have never required any processing other than default processing for the first pass, which simply allocates objects for second pass processing.

© 2005 Adis Data Information BV. All rights reserved.

```
1  <?xml version = "1.0"?>
2  <gus>
3    <dots_nasequence depth=0>
4       <fragmentname>alpha</fragmentname>
5       <sequence>
6         gttttctkgctcgatatgtctttcaagcttcggtacatacaatttggagagagcgaaatggaagaagaca
7         tggtgaaacwccaaacccagcagctcgactaatccggtggataganaaacaaattcgaaatcaactttca
8         gcaatcaxacttatgggagatagaagatatgacaagggactccaaatttggtttgcatcaagagcttaat
9         attttatagttttaaaakttttaaytgtatttattttyttaaaaaaacgttgcactwgttgtaaaaacgt
10        tattttttcttttgaatataatttaacattatattcaaaaaaaaaaaaaa
11      </sequence>
12      <description>fragment from Arabidopsis thaliana with LINE structure
13       Kapitonov V.V., Jurka J., Genetica 107 (1-3), pp. 27-37, 1999.
14      </description>
15      <dots_sequencetype fkobj="dots::nasequence" depth=1>
16          <name>DNA</name>
17      </dots_sequencetype>
18      <sequencetypeid pkobj="dots::sequencetype" key="sequence_type_id"/>
19   </dots_nasequence>
20 </gus>
```

**Fig. 4.** XML for second sequence insertion into database. Note the changes relative to figure 1 consisting of an extended sequence and its altered description.

### Tailoring Semantics

Specialised routines are required where non-default behaviour is desired, or where there is processing apart from direct mapping of input to schema. Although various policies can be embedded in the object layer, over time the object layer will be encumbered with site-specific (specialised) idiosyncrasies that interfere with scalability and may conflict with fundamental functionality. The ability to tailor selectively the input interface for interpretation of data isolates non-default processing from fundamental object-layer functionality.

The example in this section uses an update of a pre-existing sequence in the database. The default GUS policy for a sequence update looks for relational tables related to the updated root tuple. Related tuples are updated with new information as required, with a history retained in other tables for data warehouse maintenance.[8] The default GUS semantics are not of primary concern in this article. Rather, the existence of plausible alternatives to the default update semantics motivates the following discussion about non-default processing.

Consider an example of inserting into the database a modified sequence, alpha, with the original definition of alpha in figure 1. Later, it was determined that the original sequence was trimmed of a low-complexity tail that completed a structural element of interest, so the sequence with data in figure 4 was re-inserted with more of the sequence intact.**14** We consider three update policies for the re-insertion of the sequence:

1. *Update without history:* Overwrite a subset of the sequence entry, losing the original tuple to the modifications, but retaining the original primary key. This is the default GUS update policy for tables that are not versioned.[8]

2. *Update with history:* Create a new entry that retains the primary key of the replaced alpha thereby having a second alpha instance, with the original instance given a new primary key. Thus, foreign-key references to the original alpha will be lost, but transferred to the new sequence. The default GUS policy achieves this for tables that are versioned in the GUS sense.

3. *Revise:* Create a new instance with a new primary key. All foreign-key references to the original sequence are kept intact; the revised sequence and its annotations are rebuilt from scratch, while retaining the previous version and its annotations. In this example, an auxiliary table, DoTS::NAEntry, uses the name alpha as a candidate key to retrieve the latest version of alpha.

(A change to a primary key can impact foreign-key references in other tables, which must be managed consistently.) These seman-

**14** How the trimming came to be is not important here. Also, positional references to the sequence alpha may change with the change of sequence. These technicalities are important in practice, but not discussed further in this article.

© 2005 Adis Data Information BV. All rights reserved.

**Table III.** Table contents for three different insertion policies. The table NASequence with sequence information contains the candidate key name and sequence_version, primary key pKey and the sequence; sequences are abbreviated with the letter s and s′, where s′ is the new sequence (discussed in the section titled Genomics Unified Schema [GUS] and Data Input). The table naentry is used to track the version number of the instance of the most up-to-date sequence; name alone is a candidate key for naentry. Thus, in this sample application of GUS, a sequence is retrieved by first consulting naentry to determine the candidate-key component version for a given sequence name. Note that name is not an attribute of the standard GUS table NASequence, but is an attribute in a view used at our site of the table, NASequenceImp

| Update cases | nasequence | | | | naentry | | | |
|---|---|---|---|---|---|---|---|---|
| | pKey | name | sequence_version | sequence | pKey | version | name | sequence_id |
| Initial state | 10 | alpha | 1 | s | 1301 | 1 | alpha | 10 |
| Update without history (case 1 in text) | 10 | alpha | 2 | s′ | 1301 | 2 | alpha | 10 |
| Update with history (case 2 in text) | 10 | alpha | 2 | s′ | 1301 | 2 | alpha | 10 |
| | 11 | alpha | 1 | s | | | | |
| Revise (case 3 in text) | 10 | alpha | 1 | s | 1301 | 2 | alpha | 11 |
| | 11 | alpha | 2 | s′ | | | | |

tics are summarised in table III. In order to implement these semantics, a specialised plugin method is written for the DoTS::NASequence object. The action alerts the dispatcher to the non-default method with a Perl module name in the appropriate parameter slot (that is the string Specialized in figure 3, line 11). In turn, the module Specialized.pm contains an implementation of the method with the constructed name DoTS_NASequence_02.[15] The specialised routine is written in the context of the XMLGUS framework with objects and their attributes defined accordingly. In this example, the objects DoTS::NASequence and DoTS::NAEntry are manipulated to implement the semantics of case 3 (Revise) above.

### Key Constraints

Key constraints are represented in GUS XML by constructing pointers from the foreign-key referencing object to the primary-key defining object using XML attributes with the candidate-key name. Foreign-key references to candidate keys can arise in GUS; for example, in the guise of key resolution for controlled vocabularies such as the attribute sequencetypeid in figure 1, line 14. In this case, the primary key sequencetypeid is derived from the object DoTS::SequenceType using the candidate-key value DNA; this is also discussed in the section titled GUS XML. In figure 1, line 14, the ‘pointer’ pkobj for sequencetypeid indicates the name of the candidate-key defining object dots::sequencetype for the foreign key dots::nasequence::sequence_type_id, whereas the attribute key indicates the primary key sequence_type_id to be obtained from the referenced object, which happens to have the same name in the corresponding GUS object.[16] The ‘pointer’ fkobj=dots::nasequence for the primary-key object dots::nasequencetype informs the processor that this XML nest is for the purpose of foreign-key definition. The string value dots::nasequence is superfluous to processing; however, it aids in reading the XML.

Foreign-key references also arise where one object refers to another object, such as in the resolution of the externaldatabasereleaseid in the XML fragment in figure 5. Referential constraints of the type expressed by the XML in figure 5 commonly occur in the GUS schema. The referential path[12] in this case involves intra- and inter-table references. Specifically:

$$R_1 \text{ (name)} \rightarrow R_1 \text{ (external_db_id)}, R_2 \text{ (version)} \rightarrow R_2 \text{ (external_db_release_id)}$$

where arrows point from foreign and candidate keys referencing the candidate key contained in tables $R_1$ and $R_2$, where $R_1$ and $R_2$ are shorthand for tables SRES::ExternalDatabase and SRES::ExternalDatabaseRelease, respectively. XML element nesting alone cannot represent combinations of such references. In this case, the external database elements would be nested outside of DoTS::NASequence, which is fine, but awkward.

The object DoTS::NASequence, however, also requires resolution of the taxonomy primary key through the taxonomy name. This would require that the outer nests to DoTS::NASequence be two: one each for taxonomy and external database. As a result, one of either taxonomy or external database would be nested within the

15 See http://flora.ittc.ku.edu/ for the complete method.

16 The actual name of the object attributes are used in the grammar with the exception of the XML attribute pointers. For example, line 18 in figure 4 with XML element sequencetypeid will appear as a terminal in the corresponding grammar with the action specifying the actual variable name. This correspondence is discussed in the section titled XMLGUS Grammar.

© 2005 Adis Data Information BV. All rights reserved.

other, which is inappropriate since there are not constraints between these two tables. An alternative placement of the pkobj and fkobj block for the external-database-release key is as a non-nested block before the DoTS::NASequence block. This important option is discussed in the section titled Performance.

## Applications

There are various ways to incorporate XMLGUS into a production project. As outlined above, the central components are a grammar and GUS XML-formatted input. Illustrative production applications are presented in this section, and the grammar and XML files for this work are available at the data management website for this project (http://flora.ittc.ku.edu/).

### GenBank®-Formatted *Arabidopsis* Chromosomes with Annotations

The GUS schema captures the central dogma of biology so that genomes and annotations can naturally be represented in the database. In this application, the *Arabidopsis thaliana* genome, along with gene and miscellaneous feature annotations, was input to GUS.[17] The data were downloaded from NCBI in GenBank® flat-file format.[18] The GenBank® files were formatted into several GUS XML files using simple Perl scripts.

The first set of GUS XML files contained the gene symbols used in the annotations. One file was used per chromosome for management purposes. This first step input the gene symbols in the table DoTS::Gene for reference by feature descriptions. In the second step, the gene locations on chromosomes were stored using the GUS tables DoTS::GeneFeature, DoTS::NALocation and DoTS::GeneInstance. As in the first step, a grammar was written for the GUS XML and this part of the GUS schema. Finally, the miscellaneous features were added from another set of GUS XML-formatted files using the table DoTS::NAFeature. The order of these steps is both a consequence of style and database dependences. These multiple steps could be merged into a single step if desired, since dependences can be separated with concatenated blocks of XML processed in order in the input file.

One aspect of performance is the overhead in using a system, apart from the time for execution. (The latter is discussed in the section titled Performance.) For the genome loading application, one may consider the manually written GUS plugin LoadGeneFeaturesFromXML.pm, a suitable but single-point solution for a limited set of tables in the GUS distribution. The scope of this plugin is limited by fragility and hard-coded characteristics developed for specific GUS projects.

The structured approach embodied by XMLGUS makes the same development task repetitive and routine; accordingly, the effort shifts to planning and understanding the schema, and away from time-consuming development and debugging of hundreds of lines of object-oriented code. What is more, moving the translation of foreign formats to GUS XML with standalone scripts outside of the object layer reduces the complexity of this task by removing it from the object layer.

```
1  <gus>
2    <dots_nasequence>
3      <. . .>
4      <sres_externaldatabasereleaseid>
5         <sres_externaldatabase fkobj="dots::nasequence">
6            <name>Pfam</name>
7         </sres_externaldatabase>
8         <externaldatabaseid pkobj="sres::externaldatabase"
9                             key="external_database_id"/>
10        <version>5.1</version>
11     </sres_externaldatabasereleaseid>
12     <externaldatabasereleaseid pkobj="sres::externaldatabasereleaseid"
13                                key="external_database_release_id"/>
14     <. . .>
15   </dots_nasequence>
16 </gus>
```

**Fig. 5.** GUS XML fragment illustrating candidate-key dependencies. The framed ellipses <…> represent XML omitted for this example. See section titled Key Constraints for details.

© 2005 Adis Data Information BV. All rights reserved. 

### BLAST® XML

In another production application, we utilise XMLGUS in a sequence-processing pipeline. In an early pipeline phase, vector sequences are identified by aligning sequence reads against a reference vector-sequence database. The coordinates of the nonvector-containing sequence are noted and stored as feature annotations in GUS. In this way, the complete sequence reads are kept in GUS, with the nonvector portion identified with the feature coordinates. The local sequence alignments were performed with NCBI BLAST®,[19] which optionally outputs the results in XML format. Programs that output XML are especially attractive to the XMLGUS framework since numerous tools exist for working with XML. In this case, we used an XSL-defined mapping from BLAST® XML to GUS XML.

## Performance

In general, XMLGUS processing time will be at least as good as hand-coded plugins. The processing of an object from XML to GUS objects can be readily envisioned from the XML structure, which essentially lays bare what is the XMLGUS equivalent to control flow in the standard plugin. With a plugin, inefficiencies can be more difficult to detect in as much as control flow can be more difficult to ascertain. One can, for example, review a generated XML file for accuracy, whereas with a plugin there is no human-readable intermediate step, except possibly for the messy option of dumping the internal data structures of the plugin into a file. As in code optimisation, experts may find optimisation opportunities over XMLGUS, but in general the automated processing will be at least as good as the average manually written plugin.

The scan and parse of input GUS XML is essentially the same as that encountered in processing XML by hand-written processors. Although XMLGUS traverses the DOM tree twice, the traversal is linear in the input length with a modest constant factor such that the double pass adds an insignificant overhead. Non-XML input can be processed in – at best – linear time, so there is not significant overhead in transforming non-XML into an XML format.

Additional costs can arise from repetitive queries of the same primary-key object to resolve a foreign key. This problem is not unique to XMLGUS and can be avoided in any case. Repetitive references of invariant keys can happen when the same primary-key object is referenced in multiple objects defined in the same XML file, such as genome features. With a quiescent GUS running on Oracle® and Linux, a single such reference to a small object with less than 1024 bytes takes approximately 0.016 seconds on a computer with two 2-GHz Pentium® 4 processors with 2 gigabytes of memory.**17** Although it costs little to fetch a single modest-sized object, this time can be needlessly excessive for thousands of objects. An example of such a reference is resolution of the foreign-key reference to DoTS::SequenceType where the candidate key is the nucleotide type. This reference requires only one reference to DoTS::SequenceType to resolve the sequence-type foreign key to the controlled-vocabulary table for *nucleic acid types* to define the genome features as in the example in the section titled GenBank®-Formatted *Arabidopsis* Chromosomes with Annotations. A primary-key-containing object defined at an XML level of depth zero persists, thereby facilitating the desired behaviour of a single fetch of invariant objects.

## Related Work

In the work described in this article, XML is modelled by a context-free grammar in an interface to object-oriented middleware. Timoshkina et al.[20] studied Lex and YACC in the context of constructing a general-purpose processor for transforming XML documents into HTML.[20] They cite the bottom-up parsing action as a disadvantage, whereas in our experience this is an advantage for structuring the plugin logic and processing. The XML 1.0 specification provides a verification mechanism for document classes using grammars in the DTD.[21] The DTD is essentially a context-free grammar with right-hand sides that may contain arbitrary regular expressions.[22] DTDs do not, however, provide for key constraint specifications. As an outgrowth of this limitation, constraint specification is addressed by XML Schema.[23] Within our framework, XML that is compatible with the XMLGUS processor can describe any key constraints required for the GUS object-oriented database. Numerous systems have been developed for querying XML using relational storage techniques.[13,24] This work, however, maps XML into pre-existing relational schemata. The difficult task of arriving at relational designs from XML is the inverse of the problem addressed in this article.[24] Bourret's[25] XML-DBMS maps XML objects to relational databases. The tool was not suitable for our work, in part because of a lack of chaining of candidate keys to an arbitrary depth (see section titled Key Constraints) and the restricted capacity for arbitrarily involved processing of input.

**17** Earlier releases of XMLGUS required that every primary-key-containing object be nested in the referring object; this is no longer necessary in the current version.

© 2005 Adis Data Information BV. All rights reserved.

## Discussion and Conclusions

Clear advantages are realised with XMLGUS over imperative programs for GUS data-input tasks. The structured and descriptive programming approach coupled with automated processing clarifies input processing. The approach sets the stage for automated input handling based on a schema and meantime makes for less error and less overhead filling in framework details, in contrast with single-format plugin solutions. The strategy encourages structured plugin architecture, which is especially crucial for a diverse developer community. Code readability is enhanced and programming is reduced. That the object layer interfaces a standard XML and that the standard XML is produced apart from the object layer simplifies the interface framework.

XML-generating scripts and XML output are very readable. One can, for example, review generated XML for accuracy, which incidentally is tantamount to reviewing the corresponding grammar. With a plugin there is no human-readable intermediate step, except by exposing operations by looking into the internal data structures of the plugin during execution. The XML is error checked against the grammar and vice versa, with the processing module written once and for all. The nested XML captures parent-child relationships in an obvious way, whereas the alternative Perl plugin requires a series of parentPtr→addChild(childPtr) method calls whose program order does not necessarily reflect the relation between objects.

Standard GUS plugins and XMLGUS alike confront the problem of a plethora of input formats. With the XMLGUS approach, the translation of data formats into XML by easy-to-understand scripts parallels the similar function implemented within standard plugins. The difference is that plugins will likely conform to the data, whereas with XMLGUS the data conform to the processor. Thus, data are translated into a standard format apart from their presentation to the object layer. In contrast, standard plugins organise and translate data within the process that also serves to interface the object layer. The fusing of data formatting and object creation as practiced in standard plugins is at odds with the principle of encapsulation, a cornerstone of structured programming. The readability of XML-generating scripts and their output naturally follows from a straightforward task that is retained as such through encapsulation. In other words, GUSXML imposes a design pattern on the problem of marshalling data into the object layer where XML is a suitable choice.

It is important that structured software-engineering practices be applied where possible. Bioinformatics applications highlight this need with rapidly changing developments in methodologies along with noteworthy variety in data. Robust software-engineering practice improves reliability, reduces related overheads and frees time to pursue other activities. Beyond the software-engineering thrust in part motivating this work, XML representations of a relational schema can facilitate joint queries with relational data warehouses. Also, the XMLGUS input framework can be automatically generated, further facilitating integration of data with GUS. Although some GUS input may not be suitable for XML processing, and not all developers will be comfortable with the mechanics, XMLGUS goes a long way to facilitate assimilation of input. Future work includes automating grammar generation and XML definitions from the GUS relational schema.

## Acknowledgements

This work was supported by grants to DP from the National Science Foundation and from The Atlantic Philanthropies. The authors are pleased to acknowledge invigorating discussions with Steve Fischer at the Computational Biology and Informatics Laboratory at the University of Pennsylvania.

The authors have no conflicts of interest that are directly relevant to the content of this article.

## References

1. Database issue. Nucleic Acids Res 2004 Jan; 32: D1-594
2. National Center for Biotechnology Information. GenBank statistics: growth of GenBank (1982–2004) [online]. Available from URL: http://www.ncbi.nih.gov/Genbank/genbankstats.html [Accessed 2004 Mar 30]
3. Bernal A, Ear U, Kyrpides N. Genomes online database (GOLD): a monitor of genome projects world-wide. Nucleic Acids Res 2001; 29 (1): 126-7. Available from URL: http://www.genomesonline.org/ [Accessed 2005 May 11]
4. Galperin MY. The molecular biology database collection: 2004 update. Nucleic Acids Res 2004; 32: D3-22
5. Gardner MJ, Tettelin H, Carucci DJ, et al. Chromosome 2 sequence of the human malaria parasite *Plasmodium falciparum*. Science 1998; 282: 1126-32
6. Hall SE, Kettler G, Preuss D. Centromere satellites from *Arabidopsis* populations: maintenance of conserved and variable domains. Genome Res 2003; 13 (2): 195-205
7. Leplae R, Hebrant A, Wodak SJ, et al. ACLAME: a classification of mobile genetic elements. Nucleic Acids Res 2004; 32: D45-9
8. Davidson SB, Crabtree J, Brunk B, et al. K2, Kleisli and GUS: experiments in integrated access to genomic data sources. IBM Syst J 2001; 40 (2): 512-31
9. Hall AE, Keith KC, Hall SE, et al. The rapidly evolving field of plant centromeres. Curr Opin Plant Biol 2004; 7 (108): 108-14
10. Johnson AL. Elements of programming with Perl. Greenwich (CT): Manning Publications Co., 1999
11. Project: Genomic Unified Schema development: summary [online]. Available from URL: https://sourceforge.net/projects/gusdev/ [Accessed 2004 Jul 25]
12. Date C. An introduction to database systems. Reading (MA): Addison-Wesley Publishing Co., 2000
13. Shanmugasundaram J, Shekita E, Kiernan J, et al. A general technique for querying XML documents using a relational database system. ACM SIGMOD (Special Interest Group on Management of Data) 2001; 30 (3) Sep: 20-6
14. Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques and tools. Reading (PA): Addison-Wesley Publishing Co., 1986
15. W3C. Document object model (DOM) [online]. Available from URL: http://www.w3.org/DOM/ [Accessed 2004 Mar 30]

© 2005 Adis Data Information BV. All rights reserved.

16. W3C. XSL Transformations (XSLT) version 1.0: W3C recommendation 16 November 1999 [online]. Available from URL: http://www.w3.org/TR/xslt [Accessed 2004 Mar 30]
17. Arabidopsis Genome Initiative. Analysis of the genome sequence of the flowering plant *Arabidopsis thaliana*. Nature 2004; 408 (6814): 796-815
18. National Center for Biotechnology Information (NCBI). GenBank overview [online]. Available from URL: http://www.ncbi.nlm.nih.gov/GenBank/index.html [Accessed 2004 Jul 25]
19. Altschul SF, Gish W, Miller W, et al. Basic local alignment search tool. J Mol Biol 1990; 215: 403-10
20. Timoshkina U, Bogoyavlenskiy Y, Penttonen M, et al. Structured documents processing using Lex and YACC (report 2001) [online]. Available from URL: http://citeseer.ist.psu.edu/475903.html [Accessed 2005 May 24]
21. W3C. Extensible Markup Language (XML) 1.0 (3rd ed.): W3C recommendation 04 February 2004 [online]. Available from URL: http://www.w3.org/TR/REC-xml/ [Accessed 2004 Mar 30]
22. Neven F. Extensions of attribute grammars for structured document queries. In: Connor RCH, Mendelzon AO, editors. Research issues in structured and semi-structured database programming languages. Lecture Notes in Computer Science. Vol. 1949. Berlin: Springer 2000, 99-116
23. W3C. XML schema [online]. Available from URL: http://www.w3.org/XML/Schema [Accessed 2004 Mar 30]
24. Davidson S, Fan W, Hara C, et al. Propagating XML constraints to relations. In: Dayal U, Ramamritham K, Vijayaraman TM, editors. Proceedings of the Nineteenth International Conference on Data Engineering; 2003 Mar 5-8; Bangalore, India.
25. Bourret R. XML-DBMS: middleware for transferring data between XML documents and relational databases [online]. Available from URL: http://www.rpbourret.com/xmldbms [Accessed 2004 Mar 30]

Correspondence and offprints: Prof. *Terry Clark*, Department of Electrical Engineering and Computer Science, The University of Kansas, Lawrence, KS 66045, USA.
E-mail: tclark@ittc.ku.edu

© 2005 Adis Data Information BV. All rights reserved. 