XMLPipeDB: A Reusable Tool Chain for Building Relational Databases from XML Sources

John David N. Dionisio Joey Barrett Joe Boyle Adam Carasso
David Hoffman Babak Naffas Jeffrey Nicholas Roberto Ruiz
Scott Spicer Kam D. Dahlquist

Loyola Marymount University

May 5, 2006

Abstract

We set out to create a method of importing data from large genetic databases into the format used by GenMAPP. We limited the scope of our work to Uniprot and GO databases and used PostgreSQL as our intermediate database. We were determined to use as many existing, open source tools as possible to facilitate this. We used Hyperjaxb2 to facilitate generation of database schemas, Hibernate mapping files and target-specific Hibernate class files. Hibernate provides the object to relational mapping. We have built a chain of tools to go from an XML Schema Definition (XSD) to a GenMAPP database file. We also succeeded in building many reusable components that are neither specific to- nor tied solely to- bioinformatics, Uniprot, GO or GenMAPP. *xsd2db* allow users to convert any XSD to an SQL database schema and supporting Hibernate files. *xmlpipedbutils* allows users to configure a system to use Hibernate for database access, preview XML data, import XML data, and run queries on data in the database. Three more tools, uniprotdb, godb and genmapp builder, were required to meet our end goal - generation of a GenMAPP database file. This tool set is unique in that there is no open source tool set that uses Hyperjaxb2 and Hibernate to accomplish such tasks.

Contents

1 Introduction

We started out with a goal, to build a tool that will take an XML file from a gene information storage giant like Uniprot and convert it to a GenMAPP gene database. At first this appeared to be trivial to solve by using existing tools like Hyperjaxb2 and Hibernate. However, in the begining stages of putting together our first version we found many problems that are common in XML to relational database conversion. Our tool was to be as generic as possible to be able load and convert gene information from a number of different sources, but because the nuances we discovered were source specific we could not build an all encompassing tool. We broke up the task into several modules in order to facilitate both the generic conversion and source specific customizations that were needed. Each module would take part in the task of converting gene information from one form to another.

XMLPipeDB is a suite of tools for managing, querying, importing and exporting information from XML data, to a relational database, and finally to a GenMAPP gene database. This paper describes the motivations behind our original goal. We will step you through the process we experienced and show you the solution that has reached our goal. We will look at each piece that makes this tool chain a success. We will explore how our solution solves both the data transformation required in the bioinfomatics community, as well as its applicability in the more general case within computing and computer science communities. Finally, we will analyze and critique our own work and explore further goals and future directions for XMLPipeDB.

2 Background and Motivation

With the introduction of DNA micro arrays, there is a need for tools to aid in the processing of large amounts of Bioinformatic data. GenMAPP [Gen06b] is a tool used widely in the Bioinformatics community to visualize the changes in known pathways using data collected from DNA micro arrays. In addition, GenMAPP is available to download for free. However, GenMAPP has one severe limitation. It can only be used on an organism for which a GenMAPP gene database already exists. The current process for GenMAPP gene database creation is to use Ensembl [Ens06] as the base database. The problem with this approach is that the Ensembl database only contains mammals and thus it is very limited. By creating a process through which GenMAPP databases for any organism may be created, we have pushed the door to the world of micro array analysis wide open.

3 Overall Design and Approach

To understand the overall design of XMLPipeDB we must first state our initial goal: To create a reusable tool set that given genomic sequencing data for an organism in XML and a schema for that XML document could out put a working GenMAPP gene database for that organism.

Given the many different source organizations, and consequently different schemas, for this XML data we wanted to be able to automate this process as much as possi-

ble. In the initial research phase of the project we managed to find that quite a fair amount of work had already been done by the open source community to reach our final goal. The JAXB reference implementation contained an XML schema compiler to automatically create Java objects given an XML schema. Hyberjaxb had been built using JAXB and extending its functionality to Hibernate. Thus using Hyperjaxb, we could annotate these Java objects and automatically generate Hibernate mappings for them. This allowed us to use Hibernate almost "free of charge" since the work, which would have been considerable, to annotate the Java objects and generate the mapping files, was being done for us. We used hibernate to perform the object to relational mapping. Finally, we could export the needed fields from our relational database to create GenMAPP data files.

Initially the project seemed simple and we attempted to design it as a single unit. After some experimentation, we noted that some additional post processing was necessary to massage the Java, SQL and Hibernate mapping files for Uniprot and the SQL and Hibernate mapping files for GO. Given this, it made more sense to break the project up into three different stages. The first stage, called xsd2db, described more thoroughly in 3.1, is a general purpose JAXB object and hibernate mapping generator that would take an XML schema file (XSD or DTD) as input.

In the second stage, post-processor applications, UniprotDb and GoDb were written for the Uniprot and GO databases. This takes the output from *xsd2db*, makes the kinds of changes described above and then, through an ANT build file, allows the user to Jar the output for use a library with GenMAPP Builder.

The final stage, called GenMAPP Builder, uses the output of UniprotDb and GoDb and another module, XmlPipeDb Utilities, to provide the end user with an application via which she/he can:

- configure the Hibernate properties for the intermediate database being used (for us it was PostgreSQL)
- import a Uniprot and/or GO XML file to the intermediate database; run queries on that data
- export the data to a GenMAPPdata file (MS Access MDB file with the GenMAPPschema)

The approach we propose to automatically migrate existing databases to a relational database and then use genmap builder to export a GenMAPP gene database allows us to use virtually any existing bioinformatics database that provides exports of their database in XML. This allows us to expand GenMAPP to analyze plant and bacteria organisms.

XmlPipeDb Utilities consists of components to:

- Configure the database to import data and run queries via the configuration interface
- Import XML data into a database using the import interface
- Query the and view data with the query interface

The advantage of this overall design is that each component or module can be used individually, with a completely different application or, as we have done here, as part of

a chain of applications. Some of the components, like *uniprotdb* and GenMAPPBuilder are very specific in the problem domain they address. Others, however, like *xsd2db* nad XmlPipeDb Utilities are very general and can be applied to any set of XSD and XML files the user wishes to use in conjuction with a relational database.

3.1 Database Generation with *xsd2db*

3.1.1 Introduction

While manually calling the chain of open source tools described in section 3 to generate the necessary JAXB objects, hibernate mappings, and SQL DDL (an import file set) for each different type of XML source we wanted to import was possible. We decided that this would violate our pattern of reusability. It was obvious that to even build our sample application GenMAPP builder would require us to be able to import from two different XML sources, Uniprot and GO. Thus, it was clear that it would be in the best interest of the project to create a metatool for generating these import file sets. It is this tool that became *xsd2db*.

3.1.2 Design and Implementation

Initially *xsd2db* took the form of an ant build script in the original XMLPipeDB project. However, as we began to add use cases to *xsd2db*, most notably being able to download an XML schema, it was decided that an executable jar would need to be built for the project. Thus, we decided to build *xsd2db* in two parts. A command line interface, Xsd2dbCommandLine.class, and the functional component, Xsd2db.class. These components would be kept loosely coupled i.e. the only knowledge they would have of each other is that the *xsd2db* functional component would contain a run method that could be called by the command line interface. This way the *xsd2db* functional component would be easy to integrate into future application's.

Xsd2db was created as a command line tool. The reason behind this choice was that *xsd2db* was primarily intended to be a metatool to aid developers in creating there own import engines for specific XML data. Since our primary target is experienced users we felt that a command line interface was more appropriate than a graphical user interface and would allow us to spend more time working on the functionality of the program.

The use case requirements for the *xsd2db* functional component were many. From start to finish *xsd2db* would have several tasks it needed to complete. First it would need to download a schema file from a user supplied URL. It would then need to use the JAXB bindings compiler to create JAXB objects for the schema. Next it would have to invoke the Hyperjaxb2 add-on to generate hibernate mappings for the generated JAXB objects. Using these hibernate mappings a SQL DDL file would then be created by using the hibernate schema export tool. Finally, a project directory structure would need to be built to copy the generated files into, and a build file would be generated for the project. A UML use case diagram can be seen below in figure 1.

To call the JAXB bindings compiler one must first create a com.sun.tools.xjc.Options object. One can then use the Options object to set the target directory for generated JAXB object files, the schema type of the XML schema (DTD or XSD), and

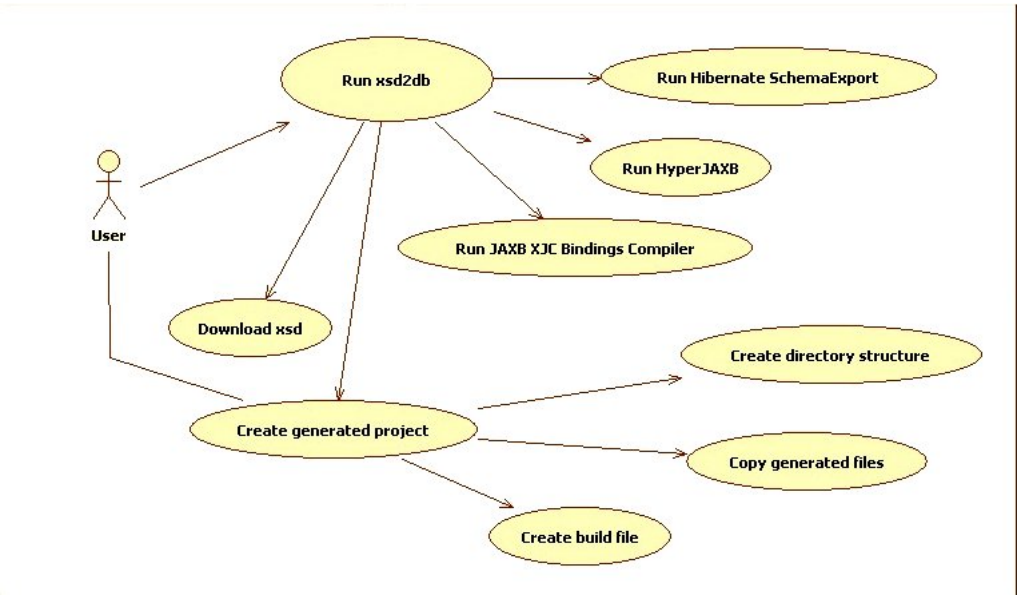

Figure 1: Use case diagram for *xsd2db*

finally the location of the schema to be parsed. Once the Options object is configured you can annotate the schema and create an annotated grammar by invoking the com.sun.tools.xjc.GrammarLoader.load() factory method on the Options object and an ErrorReceiver object. An ErrorReceiver is simply an object that implements the com.sun.tools.XJC.ErrorReceiver interface and allows a developer to decided how to handle errors; in our case we simply reported them to the command line. Once you have created an annotated grammar the XJC bindings compiler can be invoked using the static method Driver.generateCode() method on the AnnotatedGrammer, XJC Options object, and the ErrorReciver. Some sample code is shown in figure 2.

To invoke the Hyperjaxb2 add-on we take advantage of the fact that all JAXB add-ons must provide a run method since they implement the JAXB AbstractParameterizableCodeAugmenter interface. Thus to create the hibernate mapping files for the generated JAXB objects, we create a new Hyperjaxb2 add-on and call run using the AnnotatedGrammar, GeneratorContext, and XJC Options Objects that we created while invoking the XJC bindings compiler.

In order to generate the SQL DDL file we use the Hibernate SchemaExport object. Hibernate requires that the Hibernate Configuration object be created and configured before any hibernate objects can be used. To configure the Hibernate Configuration object we simply load an included Hibernate properties file and use the Hibernate Configuration object's setProperties() method to configure Hibernate. The last step of the Hibernate Configuration is to inform Hibernate of the location of the Hibernate mapping files that it will be using. In *xsd2db* we decided to do this manually using the Configuration object's addFile() method. Once hibernate is configured, generating a

```
ErrorReceiver errorReceiver = new ErrorReceiverImpl();
AnnotatedGrammar grammar = null;
try {
    // Create a annotated grammar
    grammar = GrammarLoader.load(XJCOptions, errorReceiver);
    if (grammar == null)
        System.out.println("Unable to parse schema");
} catch(Exception e) {
    System.out.println("Error loading the grammar");
    e.printStackTrace();
}
try {
    // Generate the JAXB objects
    GeneratorContext generatorContext;
    generatorContext = Driver.generateCode(
         grammar,
         XJCOptions,
         errorReceiver);
    if (generatorContext == null)
        System.out.println("failed to compile a schema");
  }
```

Figure 2: Code to invoke the JAXB XJC bindings compiler.

SQL DDL file is very simple. All we must do is create a new SchemaExporter, set the output directory for the DDL file, set the delimiter for the DDL file, and finally create the DDL file by calling the create() method on the SchemaExporter. Sample code can be seen in figure 3.

```
 // Initialize hibernate
hibernateConfig = new Configuration();
File hibPropertiesFile = new File(HIB_PROPERTIES);
Properties hibProperties = new Properties();
try {
    hibProperties.load(new FileInputStream(hibPropertiesFile));
} catch(Exception e) {
    System.out.println("Properties file failed to load.");
}
hibernateConfig.setProperties(hibProperties);

 // Add a hibernate mapping file
 hibernateConfig.addFile(hbmMappingFile);

// Produce the SQL file.
SchemaExport schemaExporter = new SchemaExport(hibernateConfig);
schemaExporter.setOutputFile(fullOutputPath);
schemaExporter.setDelimiter(";");
schemaExporter.create(true, false);
```

Figure 3: Schema export sample code

The Last phase of *xsd2db* is to create a standalone project that is ready to be compiled using Ant. To this end we first create a project directory structure. The directory structure produced is shown in figure 4. We then copied the necessary library files needed to compile the generated project from the lib directory of *xsd2db* to the lib directory of the generated project. Finally we generate a build file using a canned build file that we included as a resource in the *xsd2db* jar.

After this last phase the user can then compile the output of *xsd2db* using the generated build file. The resultant out put is a java library file that can be combined with the *xmlpipedbutils* described in section 3.2 to create a functional application capable of importing data from the user's XML sources and to a relational database and querying that database.

3.1.3 Performance Analysis

xsd2db has been tested on three different XML schema files with varying degrees of success. We preformed initial testing on the Books.xsd schema. Books.xsd is the

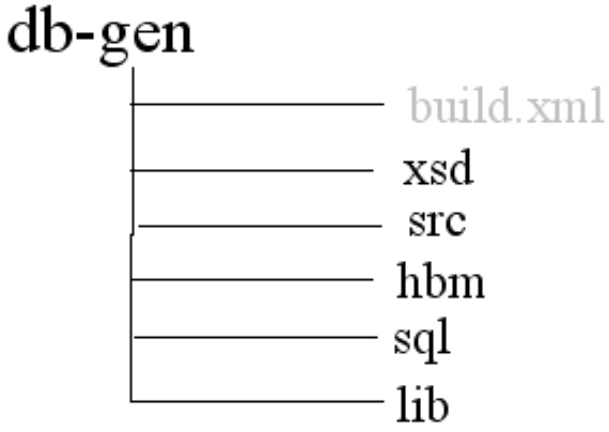

Figure 4: ***xsd2db*output project directory structure**

canonical schema used in most example applications that use XML data and thus we felt it was an appropriate choice for our initial test schema. When using *xsd2db* we were able to import XML data into a Postgresql database, and later query that database without making any modifications to our generated files. The success of this initial test lead us to try using *xsd2db* on two other schemas: Uniprot.xsd and the GO DTD schema.

After running *xsd2db* on the Uniprot.xsd we discovered that some post processing would be required on the generated project before we would be able to import XML data into a Postgresql database. This was largely due to several incompatiblities with the Postgresql database and no fault of *xsd2db*. However, one problem did arise which was a direct occurance of a failure of *xsd2db* to properly handle XSD union types. This was later attributed to a bug in Hyperjaxb2 and remains a part of *xsd2db*. We created another tool, uniprotdb, to resolve these issues by post processing the output of *xsd2db*. A more thorough examination of these problems as well as a detailed description of uniprotdb can be found in section 4.1.

The last schema file we tested *xsd2db* on was the GO schema file. The GO schema file presented a unique challenge in that it differed from the other schemas we tested *xsd2db* on since it was DTD and not an XSD schema. Again as with the Uniprot.xsd we encounter some problems with Postgresql and had to develop a post processor for GO, *godb*. *godb* and the problems we encountered are described in more detail in section 4.2.

In addition to the test we preformed with these three different schemas *xsd2db* also contains several unit test of the command line interface. Thus, we are sure that the command line interface works and is bug free.

Overall it can be said, that *xsd2db* does what is intended. It is a metatool for developers to create there own xml import engines, thus it is reasonable that it might require post processing depending on a developer's design choices. However, despite this initial

success it is clear that there remains work to be done to improve the performance and robustness of *xsd2db*.

3.1.4 Future Work

For the first round of development we initially focused on simply getting *xsd2db* working as quickly as possible. While some minor unit testing has been preformed, as well as, a good amount of maintenance to re-factor the code base, more can always be done.

One aspect we would like to work on would be to improve the compatibility of *xsd2db* with the Postgresql database. This would fix the problems with both biological databases referenced in sections 4.1 and 4.2. In addition, on the same line of thought we would like to fix the XSD union bug in Hyperjaxb2.

Another area of improvement in *xsd2db* would be to develop more unit test for the *xsd2db* command line and the *xsd2db* functional unit. When the unit test were initially written for xsd2db the functional component was not written in a state that would lend itself to be easily unit tested. It has since been re-factored and unit test should be written.

The last area of improvement, would be to possibly add a graphical interface to *xsd2db*. While *xsd2db* is primarily intended as a tool for developers, and thus a command line interface is appropriate, adding a graphical interface to *xsd2db* can only ease its use.

3.2 Performing Common Operations with *xmlpipedbutils*

From the outset of the project, it was clear that some common functions would be needed. These functions were grouped together as xmlpipedbutils (XPD Utils). XPD Utils contains tools to

- Configure the database to import data and run queries via the configuration interface
- Import XML data into a database using the import interface
- Query the and view data with the query interface

These tools not only provide behind-the-scenes work that is needed to perform these tasks, but also include GUIs. Essentially, each one is a self-contained, plugable component that performs its intended function. They are delivered bundeled together in one jar file, but they have not inherent interdependencies. The basic design of the utilities is a user interface with a back end engine. The engines perform the meaningful work and are designed to work with or with out the user interface. The next three sections will examine each component individually in terms of the reason it is needed, how it is designed, a frank evaluation of its capabilities and a look at what might still be done to enhance or expand on it.

3.2.1 Client Configuration

3.2.1.1 Introduction In order to make the tools as easy to use as possible the issue of Hibernate properties configuration had to be addressed. Hibernate supports a

wide variety of databases and data access methods. While advantageous for technical people, this can make configuration a daunting task for others. We chose an approach that would be modular, allowing the configuration component to be plugged into any application seamlessly.

3.2.1.2 Design and Implementation There were several challenges in the design of this component. First was to determine how to break down the myriad of supported database connection types and sundry other properties in a logical, straight forward and convenient way. Next was the issue of flexibility. We wanted a design that would adapt to changes in the hibernate properties structure. This is particularly important in this type of project, since Hibernate is not under our control and therefore could change how they are doing things at any time. We did not want to have to perform a complete re-write in the event of such an occurrence. The final challenge was to display all the properties in a convenient and straight forward way. To overcome these challenges, the project was broken down into tasks: redefining the hibernate properties approach; creating data objects to model the properties; creating an engine to read and write properties; and finally creating a GUI to allow users to easily manipulate the properties. This led logically to breaking the component in to four distinct objects: HibernateProperty, HibernatePropertiesModel, ConfigurationEngine and ConfigurationPanel (see Figure 5). This approach also conveniently and purposely uses the MVC (Model, View, Controller) design patter to separate the presentation from the data and the control. In this design, the HibernateProperty and HibernatePropertiesModel are the data or model component; ConfigurationEngine is the controller component; and ConfigurationPanel is the view component. It should also be mentioned that two new exceptions were created: CouldNotLoadPropertiesException and NoHibernatePropertiesException. These apply when the user of the configuration tool is trying to do something that fails. They are important to inform the caller of the issue, so appropriate action can be taken.

A thorough examination of the hibernate properties showed that properties could be easily broken into categories and types. Categories are platforms, general and connection pools. Platforms contain the different database platforms supported by hibernate via direct connection, e.g. PostgreSQL, MySQL, etc. Connection pools are the different connection pools, such as C3PO or Apache DBCP, that are supported. General are properties that apply to any setup, regardless of whether it is a direct connection or a connection pool. Types are used to identify the specific type of property within the category, e.g. PostgreSQL is a type within the platforms category and C3PO is a type within the connection pools category.

The data structures to support this breakdown consist of a Hibernate Property object to store each individual property and a HibernatePropertiesModel as a container for the HibernateProperty objects. HibernateProperty objects consist of a category, type, name and value, as well as an indicator of whether the property is saved or not. This allows both saved and unsaved properties to be stored in the same model. The ConfigurationEngine loads the default data structure for properties along with default values into a HibernatePropertiesModel. The saved properties are then read in and layered on top of the default data model, replacing the default values for those properties that were saved. The resulting model contains all the possible properties,

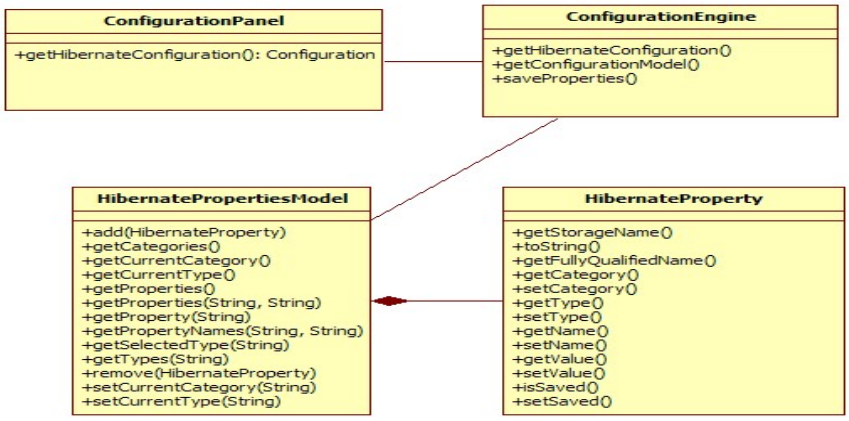

Figure 5: XmlPipeDb Utilities - Configuration Utility Class Diagram

and the values for those that were saved.

The model is used by the ConfigurationPanel to render the a GUI for the user to view and/or edit the properties. However, if a different GUI were desired, the ConfigurationEngine, HibernatePropertiesModel and HibernateProperty objects could be reused. When properties are saved, a model object is passed back to the controller, which then stores the properties in a file.

3.2.1.3 Performance Analysis Overall, the component works well. It will save the properties selected with the values entered and it will produce a Hibernate Configuration object that can be used to access a database. All this is done graphically and in case hibernate changes what databases it supports, for example, the configuration tool can be changed too, simply by updating its configuration file. From a developer and end user perspective, the tool is easy to use, in that no configuration of the configuration tool is required. It will default to picking up its properties from the jar in which it runs. Or if it is run exploded, it will pick the internal configuration up from the folder. When it persists the properties that were set, it does not require any input regarding where those properties will be saved. This makes everyone's lives easier, even if at the expense of choice. Those are the strengths.

On the other hand, it is still very much a buyer-beware type of tool. The user must know what values to input. The user must understand whether to use a connection pool or a platform and what is meant by the term platform (direct connection). The user is not given much guidance and is assumed (or required) to be something of an expert or at least experienced at database configuration. This is a draw back of the tool, from a users point of view. The GUI has an oddity, in that the combo box and the fields are displayed centered - top to bottom - rather than aligned to the top of the panel in which they are displayed. This is not very evident, unless the list of fields is very short. There is no indicator of whether something has changed, and therefore a

save is required. This is a nice to have, particularly since moving from one category to another or from one type to another causes the data that was showing to be lost.

From the point of view of a developer using the tool as a component (and possibly an end user, as well) The component does not support setting of locations for hibernate mappings. This means that the Configuration object returned cannot include this information. Otherwise, from a developer point of view it is really a plug and play component. Very little coding is required to create the panel and get the Configuration object.

3.2.1.4 Future Work The emphasis in this first round of development was to develop something that was very flexible. That goal was achieved. There are a number of things that could be done to improve the product. These range from requirements to minor improvements to major work.

As a requirement of the next version, unit tests must be created to excercise the functionality of the tool. No automated unit tests were written for this component. Adding these would ensure the integrity of the methods used and that they are performing as desired. The first thing that should be done for a new version is to create a unit testing framework.

There are some quick wins that could be attained by polishing some blemishes out of the current GUI. These would not take much work at all. The combo box and fields of the center panel should be top aligned, instead of center aligned. Enabling the revert and default functions. Revert is intended to go back to the configuration as it was when the application was started. Default loads all the properties to their default, Hibernate defined, values. Both add some flexibility for users.

The Hibernate Configuration object can hold a great variety of information. Determining what of that information is relevant to an end user and providing a mechanism for configuring that would be a dramatic enhancement. This is a larger undertaking than the previously mentioned changes. Included in this would be configuring the location of the Hibernate mapping files. However, this brings with it another set of issues. It is convenient for the caller and therefore the end user to be able to use more than one set of configuration files. Determining exactly how to incorporate this dynamism into the configuration tool would be a challenge. The utility to the developer using the configuration tool would be simply making one call to get the Configuration object and then using it straight away, without any additional handling.

Lastly, and not exclusive of the previous change, would be a complete overhaul of the GUI. The current layout works. However, that does not mean it is the best overall design for configuring these properties. In fact, some of the design might stay, but major improvements could be made in the utility of the GUI. Properties that have been set could be shown in a textarea on the top of the screen or a separate tab or even a detached window. This would provide the user with a quick overview of the properties. More guidance can be provided in setting properties correctly. Currently the default hibernate value is shown in the textfield. Once the textfield is changed, this information is lost. The hibernate default could be put into a tooltip, so that it is always available to the user. In that vein, additional information could be provided, particularly for properties that are not necessarily intuitively configured. Provide a way to test the setup would allow the user to be sure that everything was correct before

continuing. This should take the form of opening a connection to the database and, in the event of a failure, reporting as closely as possible what failed in establishing the connection (e.g. bad URL, bad login, etc.). Lastly, the way properties are set could be re-examined. The options may be layed out differently or in more of a guided format to provide the user a better feeling for what they are expected to do.

3.2.2 XML Import

3.2.2.1 Introduction The general idea with this utility is to import XML files into the a configured database. This will take any XML file or input stream convert them into Java objects and then store them into an SQL database. The interface to this utility is an intuitive user interface where you can open, preview and import files. The interface had to be intuitive enough for a non-computer science audience.

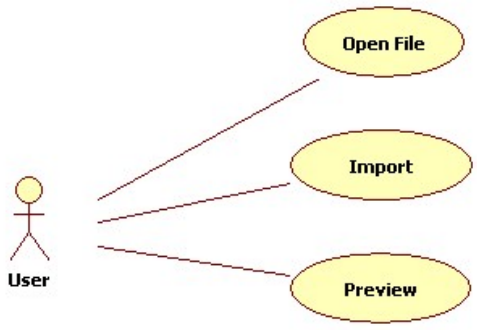

Figure 6: The Use Case For The Import Utility

3.2.2.2 Design and Implementation The first design choice we made was to use Java Swing components to develop our user interface, so we could use the standard components in the Swing library. These components have been well tested and are familiar to the user. The second design choice was to use pre-configured JaxB and Hibernate properties. The import utility assumes that hibernate has been properly configured and the JaxB classes exist. The hibernate and JaxB configurations are required upon loading. Error messages received by the utility are likely to be because the database or hibernate properties have not been configured correctly, or the utility cannot find the JaxB classes.

The XML Import utility is made up of two classes: the ImportPanel for the user interface, and the ImportEngine for the logic. We chose this design to have interface separated from functionality. We wanted the ImportEngine to be removable from the GUI so developers can take that class and use it with other user interfaces.

The user interface includes: a text area for previewing XML files, an open dialog for opening XML files, a text area to view an opened file path, an open button, an import button, and a preview button. Files can be opened via a file dialog by clicking open

button. The open dialog that has a default filter for XML files for ease of use. Once opened, the file can be previewed by pressing the preview button or imported using the import button. When a preview is performed on a large file, a progress monitor pops up to give the user feedback on how long the action will take.

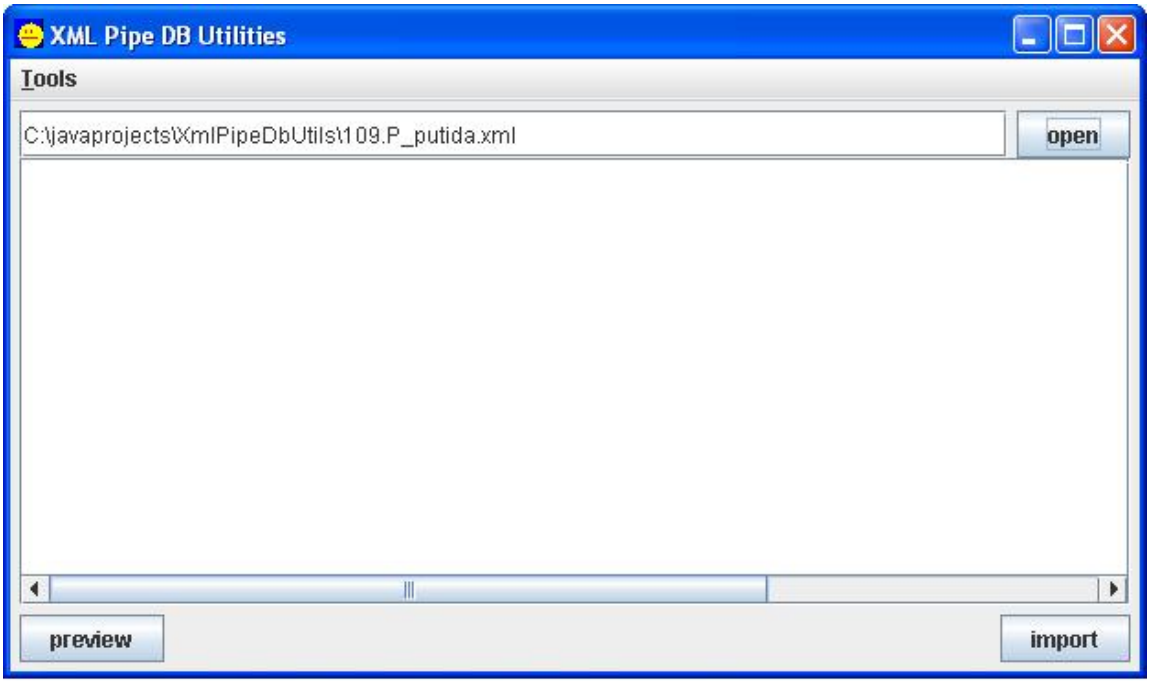

Figure 7: A screen shot of the import interface

When the import button is pressed a new ImportEngine is created. The ImportEngine takes a hibernate configuration and a JaxB context path. The loadToDB method called which converts the xml to Java objects then uses hibernate to commit the data to the SQL database. A detailed class diagram is shown below. A progress monitor opens, however, it did not perform as specified in Swing. This will be addressed in the next version

3.2.2.3 Performance Analysis The module and design worked well from the start. Most of our problems were with our build tool, ANT, and making small changes to the user interface. We went through several iterations with the panel to make it more usable and to fix bugs. We had to learn the how to save objects to the database using hibernate as the middle ware to create the ImportEngine. Hibernate performed very well, and the ImportEngine required only minor changes after the first coding iterations. We was a little disappointed with the Java components that we tried to use for user feedback, specifically the progress monitor for input streams. Though it worked well for previewing large XML files, it did not work as expected for giving

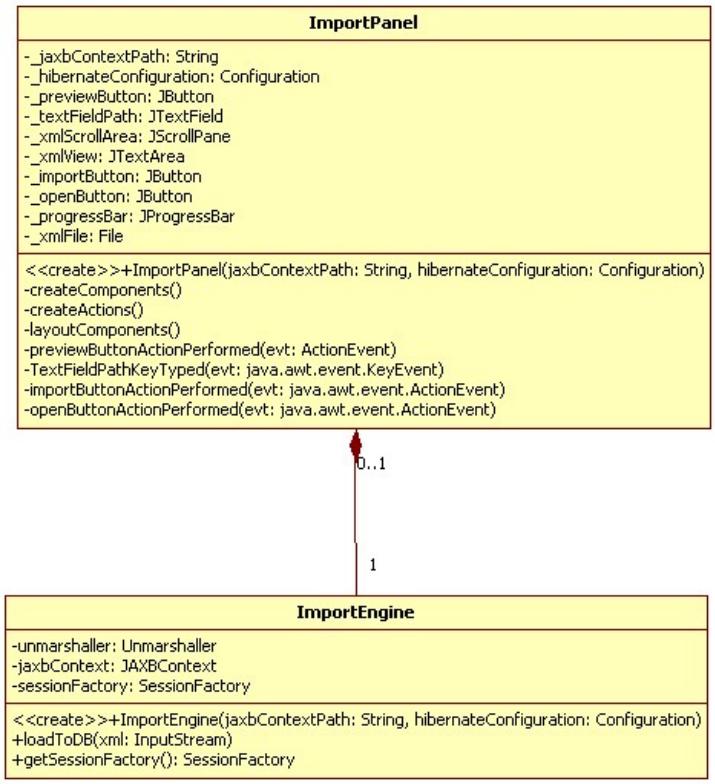

Figure 8: The ImportEngine is part of the ImportPanel but it can also run stand-alone

feedback during a large import. Overall, the utility performed well.

3.2.2.4 Future Work We plan to focus on the usability of the interface. We would like to implement a better and consistent user feedback system. There are also performance issues we would like to solve, such as allowing multiple imports to be conducted at the same time and a more seamless configuration with multiple databases. Lastly, we would like to implement tool tips and have a help menu item.

3.2.3 Ad Hoc Queries

3.2.3.1 Introduction A major factor in any research tool is data analysis. To this end, we are providing users of xmlpipedbutils with two query tools: an SQL analyzer and an HQL analyzer. Each of these views allows for a different view of the data and allows each user to utilize the query language they are most familiar with.

3.2.3.2 Design and Implementation

3.2.3.2.1 GUI Design The query panel is split into three sections; each separated by a splitter. On the left of the user interface is the query section. The radio and command buttons are on the right. The right side consists of two radio buttons and two buttons. The two buttons let the user choose between standard query language (SQL) and hibernate query language (HQL) queries.

The left side is further split into two other sections. The north section contains a text area in which the query can be typed. The bottom of the panel contains two components: a table and a tree. How each view is populated depends on whether we are executing an SQL query versus an HQL query.

The command buttons are labeled "Clear" and "Execute Query". Clicking on clear will erase the text from the query text are and clear any data from the results table and tree. Clicking the execute button will execute the query in the text area and populate either the tree or the table with the returned data. We will discuss how the data is displayed further in the following sections.

3.2.3.2.2 SQL Analyzer The SQL Analyzer allows the user to execute SQL queries and view their results. The results of the query (if any) are displayed in a table in the user interface. The table's heading is updated to match the field names of the results.

The result table is a simple JTable. The DefaultTableModel object associated with the JTable allows for all manipulations of the table's data.

Each executed query returns a ResultSet object. The ResultSetMetaData object associated with the ResultSet instance provides the names for each of the returned columns. These column names are used for the table model's header. The table model is then populated one row at a time using the ResultSet.

3.2.3.2.3 HQL Analyzer The HQL analyzer works just like the SQL analyzer in that the user can manually type in any query, click the execute query button, and view the results. Where as the SQL analyzer would display all of the results in a table, the results from the HQL queries are displayed in an object tree. The tree displays each object's hierarchy within a tree structure.

The tree is implemented as a customized JTree. Each of the objects returned by the Hibernate query are stored as the user object of a DefaultMutableTreeNode. Each of the properties of the object for which we have a getter and/or setter are further represented as children of the nodes. This recursive deepening goes until we reach the core Java classes.

3.2.3.3 End-Users Using the query analyzer is identical whether working with SQL or HQL. The query is entered in the text box on the north side of user interface. The radio buttons on the right allow the user to designate the entered query as either SQL or HQL. The results of an SQL query are displayed in a table format. Since and HQL query returns a set of objects, the results of an HQL query are displayed in an object tree in order to display the breakdown of each object.

3.2.3.4 Future Work We will now discuss some changes that are being considered for the next iteration of the utilities. The user interface and the object tree still

have plenty of modifications that can be performed and these will be addressed in the near future.

1. Currently, the object tree is populated with all of the returned data upon executing the query. While this works fine for small sets of objects, this is not efficient for large data sets. For the next iteration we will maintain a copy of the returned objects locally, but we will only populate the first level of the tree. Each sub tree will be further populated as the user expands the corresponding tree nodes.

2. The user interface currently displays both the result table and the object tree simultaneously. The next update should display only one of these components at a time. Upon selecting each of the radio buttons designating which type of query we are working with, the proper component should be made visible. Selecting the SQL radio button shall display the result table and selecting the HQL radio button shall display the object tree.

3. The result table associated with SQL query results currently only displays the data as it is returned. So modifications, manipulations, or sorting is currently allowed. For the next version, we will utilize the the customized JTable available in Dr. Dionisio's Shag package.

4 Application to Biological Databases

The GenMAPP application uses a gene database that contains species-specific libraries of gene information. The database is needed in order to link expression data with MAPPs, for creating and modifying MAPPs, and for importing new data. The gene database contains two main types of tables, Gene Tables and Relationship Tables. A Gene table is a collection of gene identifiers obtained from a cataloging system for a particular species. A Relationship Table provides a link between gene IDs of two separate gene ID databases. These tables are essential to the GenMAPP application [Gen06a].

GenMAPP gene databases function around a central system, known as the MOD system. A Model Organism Database (MOD) is a public database that contains genes and annotation for a particular organism. The purpose of using a MOD system within GenMAPP is to form a link between GenMAPP and the gene ontology (GO) hierarchy [Gen06a]. Current gene databases supplied by GenMAPP use the Ensembl database to link to GO. However, Ensembl is limited to the number of species it represents, which is only mammalian organisms. As a result, for the XMLPipeDB project, UniProt was chosen to form the link to GO based on the number of species available from it. In this case, UniProt is not actually a MOD in the strict sense of the word. But it can be used to form the needed link to the GO hierarchy.

UniProt (Universal Protein Resource) is the world's most comprehensive catalog of information on proteins [Uni06]. The European Bioinformatics (EBI) group and the Swiss Institute of Bioinformatics (SIB) produced Swiss-Prot and TrEMBL (Translated EMBL Nucleotide Sequence Data Library) whereas Protein Information Resource (PIR) produced the Protein Sequence Database (PIR-PSD). By noticing that the information contained in these separate databases was beginning to overlap, the UniProt Consortium was formed. UniProt has now become the central repository for protein sequence and function after joining the information found in Swiss-Prot, TrEMBL, and

PIR databases.

The Gene Ontology database provides consistent descriptions of gene products present in other public databases. The GO consortium is developing three structured, controlled vocabularies (ontologies) that describe gene products in terms of their associated biological processes, cellular components and molecular functions in a species-independent manner. The GO table in the Gene Database stores the individual GO terms and their relationship to each other. Since the GO ID does not refer to a gene, it is not an appropriate gene ID system to use as primary ID on a MAPP (a special file format produced by GenMAPP) or in a Gene Expression Dataset [Gen06b].

4.1 UniProt

4.1.1 Introduction

It is important to restate the goal of the XMLPipeDB project before understanding the important role *uniprotdb* played in the project; To create a reusable tool set that given genomic sequencing data for an organism in XML and a schema for that XML document could output a working GenMAPP gene database for that organism. During the beginning phases of the project, there was only one project, XMLPipeDB, and it made use of the schema provided by the UniProt Consortium [Uni06]. However, it soon became evident that the project could be designed in a manner that would allow a wider audience the use of our tool. The first phase was *xsd2db* (Section 3.1) which allowed for the use of any schema to create the necessary JAXB objects and hibernate mappings. However, we still needed to create a tool specific for the UniProt XML and XSD. At that point, *uniprotdb* was born.

4.1.2 Design

Before the project was split into a variety of subprojects, XMLPipeDB offered the ability to use the UniProt schema and create the necessary JAXB objects and Hibernate mappings. It created these files in default directories that are called out in the Hyperjaxb2 template project [Hyp06]. Customizations were used in order to get XMLPipeDB to work properly, but we hoped to further understand how to use Hyperjaxb2 customizations to overcome these issues. Section 4.1.3 discusses in further detail the Hyperjaxb2 customizations that were needed for XMLPipeDB to function.

Once it was determined that we wanted to allow groups outside the Bioinformatics community the use of our tool, *uniprotdb* became its own subproject. *uniprotdb* would be created from the output of *xsd2db*. It would contain the UniProt schema, customized external binding file, SQL DDL file that corresponds to the schema, the JAXB objects, and the Hibernate mappings. The layout shown in Figure 9 was what we hoped uniprotdb would look like.

The build file would have the following capabilities:

- *compile* compiles the JAXB objects located within the src directory with help from any necessary jars located in the lib directory
- *jar* compile and create a jar file containing the class files derived from the src directory and the hibernate mappings located within the hbm directory

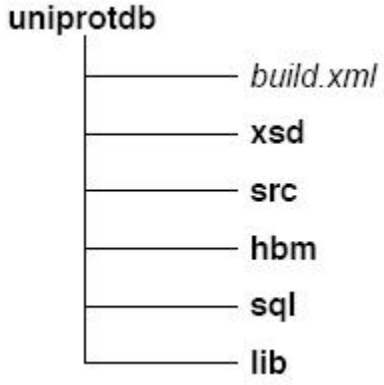

Figure 9: *The uniprotdb layout.*

- *clean* eliminates all the products of a previous build.

The jar file created as a result of a build would later be used in the gmbuilder subproject, thoroughly outlined in Section 4.3. Once uniprotdb was created, we would have been one step closer to creating a GenMAPP usable, gene database.

4.1.3 Implementation

The work we did on the original XMLPipeDB project set the stage for the creation of uniprotdb. The work on the original XMLPipeDB project brought us to a point where we were able to load the SQL DDL file that was produced from the UniProt schema. In addition, we were able to import the JAXB objects into the database. Before this point was reached though, we had to overcome three issues with the hyperjaxb2 and the UniProt schema. The issues that slowed our progress with XMLPipeDB were:

- PostgreSQL doesnt like attributes whose name is "end" (see LocationType in UniProt XSD)
- String types default to a maximum length of 255 (varchar(255)), but some strings in the UniProt XML files exceed this length
- Hyperjaxb2's handling of the xsd:union tag doesnt seem to work properly, resulting in MappingExceptions when attempting to load an XML file

The solution at the time was to manually edit the generated files. The SQL DDL file that corresponds to the UniProt schema had a need for multiple edits. The first was to change"end" to "endPosition." This solved the problem with keyword "end" in PostgreSQL. The second edit was to change "varchar(255)" to just "varchar." This solved any issues with there being a maximum length for strings within a UniProt XML file.

Since we changed the SQL DDL column named "end" to "endPosition," the same change needed to be made to the corresponding hibernate mapping file. This edit took place within LocationType.hbm.xml. Lastly, CitationType.hbm.xml needed to

be manually edited due to the xsd:union tag being an issue with hyperjaxb2. The "Date" field of CitationType allowed for any form of Date, whereas the best solution was to only allow strings. So CitationType.hbm.xml was updated to only allow strings for Citation Dates. Once all these issues were resolved, the import of a UniProt XML file into the database was no longer a problem.

At about the time XMLPipeDB was beginning to show promise, the decision was made to split the XMLPipeDB project into a group of smaller projects that would allow for groups outside the Bioinformatics community to use our tool. Hence, uniprotdb was supposed to be a subproject specific to the UniProt schema. Rather than forcing the user have to manually edit the output of xsd2db due to the issues above, we looked at the use of a customizable binding file within hyperjaxb2. Custom binding files would allow for changes to a schema and its generated files without any extra work required by the user. The use of custom bindings files were looked at for a few weeks before it was stopped due to complexity and time constraints. Refer to Section 4.1.4 for further detail regarding custom bindings files.

Once it was determined that custom bindings would not be used, we needed an alternative method to the editing of the generated files from xsd2db. We decided to develop a post processor. The post processor would reside within the uniprotdb project under a new directory named tools. Within the tools directory would be a build file, a README file, and the necessary source code to do the post processing. The build file had the ability to compile the source code and eliminate all the products of a previous build.

The post processor had the ability to be run via command line with options to specify the locations of the files to be edited or with the use of a graphical user interface (GUI). With the GUI, there are two windows shown for each file. The first window asks for the user to locate the file to post process. It is shown in Figure 10. Once the

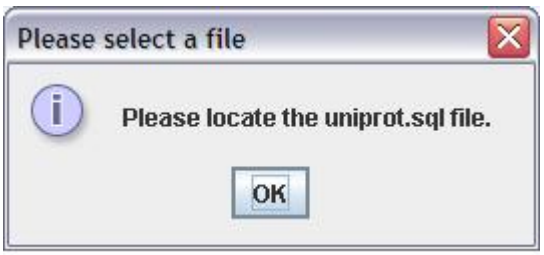

Figure 10: *The locate file dialog.*

user hits the OK button, a second window appears. The second window is a file dialog for the user to use as a navigator for the file to be processed. It is shown in Figure 11. After the last file is processed, the window in Figure 12 appears to let the user know that the operation was successful.

Further testing resulted in the finding of more bugs with the "Date" field in the schema. As a result, the post processor needed to be updated to process two more java classes. CitationType.java and CitationTypeImpl.java both needed post processing to make sure that the "Date" field used by the JAXB objects would only return a object

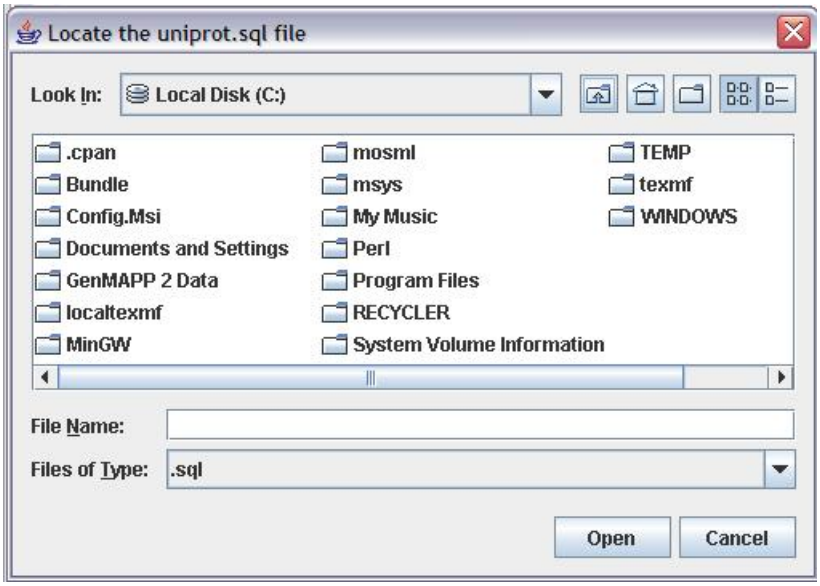

Figure 11: *The choose file dialog.*

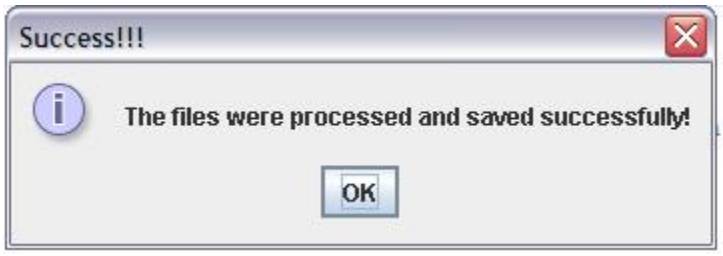

Figure 12: *The success window.*

of type string. With the addition of two more files being post processed, the decision was made to no longer make the command line version of the post processor available. We felt that the user should not be burdened with having to fill in the absolute path for a total of five files.

4.1.4 Future Work and Improvements

The only visible improvement to uniprotdb would be to use a custom mapping file rather than a post processing tool. The custom mapping file would be used with xsd2db and the UniProt schema and would allow custom mappings to be generated for classes and fields by annotating the schema. This would no longer make a post processor needed.

The future of uniprotdb is up in the air. Only a small subset of the total UniProt XML files available have been imported into a database with gmbuilder. This means that more post processing may be needed because all the post processing needs may have not been discovered. As testing continues, the Bioinformatics community will need to feel free to report any bugs and errors. In addition, a new UniProt schema could become available resulting in more post processing. As a result, it will be the responsibility of this group of developers to keep track of the ever changing Bioinformatics landscape.

4.1.5 Conclusion

uniprotdb is important in order to reach our goal of creating a gene database that can be used by the GenMAPP application. xsd2db creates the necessary JAXB objects and hibernate mappings corresponding to the UniProt schema. The uniprotdb post processing tool also appears to work with the current UniProt schema. However, a much cleaner implementation would make use of a custom bindings file. Future implementations of uniprotdb should look at the process of creating custom bindings file again. It must be noted though, as simple as uniprotdb is, it is crucial for gmbuilder application to function properly.

4.2 Gene Ontology

The Gene Ontology (henceforth GO) database is a staging database created to hold the records describing gene products according to biological processes, cellular component, and molecular functions. This database allows the user to extract structured-controlled vocabulary terms which describe gene products in any organism [Ont06]. In particular, Escherichia coli related data is queried as a part of the main objective of the xmlpipedb project.

4.2.1 Introduction

GO and Uniprot, which is described in Section 4.1, are the two data structures needed to output a working GenMAPP gene database for a particular organism. During the beginning phases of the original XMLPipeDB project, only Uniprot had been included into the build process; GO came along after the project was broken into subprojects,

which was named godb. The first step in creating a godb deliverable was to run a GO schema file through xsd2db, which is described in Section 3.1. xsd2db generates the necessary JAXB objects, hibernate mappings, SQL DDL file, and the godb build file. The build file is then executed to produce a deliverable jar file which would later be used in the gmbuilder subproject, which is described in Section 4.3.

4.2.2 Design

GO exports its data to a number of different file formats to allow flexibility for different users with different background and purposes. For this subproject, choosing a file format proved to be a difficult task. There are three main file formats to extract GO data: flat files, mySQL formats, and XML files. Flat files were eliminate because they are not compatible with JAXB classes (Java Architecture for XML Binding), which are called by xmlpipedb utilities. MySQL format were eliminated because a "one-off" tool would need to be created to port the data from mySQL to postgres. Initially, this format was considered since xsd2db required a XSD schema file (hence its name), which GO did not provide. A later version of xsd2db supported DTD schemas, which GO uses to define their schema definitions. If we used mySQL, then what happens if the schema changes? The "one-off" tool would need to be changed every time GO changed its schema, which would be costly and inefficient. But xsd2db provides the ability to auto generate any time GO changes their schema, so the mySQL format was not passed on, which left XML formats.

GO provides their data in two XML format which both contain DTD schema files: OBO.xml and RDF.xml. However, the members of the GO Consortium claimed that they will eventually generate a XSD schema for OBO.xml, and that there are no plans to generate an XSD schema for the RDF file. In fact, they only reason why they currently have a DTD file for RDF.xml is for historical reasons. Thus, the OBO.xml file format was chosen for the project, and therefore the OBO.dtd file was fed into xsd2db.

As it turned out, the OBO.dtd file produced a SQL schema that contained a tabled name *To*, which happens to be a postgresSQL keyword. Furthermore, since the DTD file did not specify a maximum character length for all GO tags-value pairs, the SQL schema defaulted those tags to type *varchar(255)*. Unfortunately, though, some of the data exceeds the character limit of 255. Due to the variety of problems, a godb postprocessor was created.

4.2.2.1 Postprocessor The godb postprocessor is a GUI based tool to fix the errors in section 4.2.2. Two files needed to be modified before godb can be delivered: *schema.sql* and *To.hbm.xml*. For the *schema.sql* file, the postprocessor searched and replaced all instances of *varchar(255)* to *varchar*, which specifies an unlimited character string type. Thus, any data that exceeded 255 characters would not generate an error. Additionally, the postprocessor changed the table name *To* to *To_*, which eliminates any conflicts with postgresSQL keywords. Xsd2db, however, creates a xml hibernate mapping file that expects there to be a table named *To*, which moves us on to the second file to modify, *To.hbm.xml*. Within this XML file, there is an *table* element that defines the table name for this mapping, which had the value of *To*. Thus, the

postprocessor simply replaced this value from *To* to *To_*.

In the interest of flexibility, the postprocessor GUI asks the user to supply the file names to be modified. Once each file has been selected, the tools proceeds to fix the corresponding files and returns a success message if the changes were done correctly, otherwise the application terminates without modifying the files.

4.2.3 Performance Analysis

Once the godb ran through the postprocessor, a OBO.xml was loaded using the import features of xmlpipedb utilities. Unfortunately, the import failed; fortunately it was simply a memory heap issue: since the OBO.xml file contains alot of data, and JaxB instantiates the "entire" object graph in memory before committing it to the database, an *out of memory* error was issue. To fix this, the java *maximum heap size* was increased to 1024mb, and the subsequent import was successful. It took around 40 minutes to load the entire file on a 2.0GHz machine.

4.2.4 Future Work

The future work with the godb can be extensive. However, in order to have a good understanding of future change requests, the developers must work together with biologists to find relevant information needed to support their requests. After all, the biologist are the end users. One potential issue with godb is that it was only tested on a postgres database. In other words, other databases may require additional/other fixes to be performed by the postprocessor. If that were the case, then the postprocessor tool would need to change to allow the user to select the database type. Integration test would also have to be performed in using another database.

Another future work that can be done is documentation. Although the godb project successfully performed its work, it was not documented properly. A software requirement specification can be prepared which lists all the requirements on the database and the postprocessor tool. The delivery of this document can give an idea to users and developers on how to proceed when using or modifying the godb applications.

4.2.5 Conclusion

The godb served its purpose because all the data from gene ontology was successfully uploaded. The database access was also successfully executed by gmbuilder without any major problems. The development of this part of the project was facilitated in part because the uniprot database. The functionality of the postprocessor was mirrored from the same postprocessor developed by uniprot db. By using the same code design as the uniprot postprocessor, the future maintenance of the godb code can be done in parallel with any changes done on the uniprot db side.

4.3 GenMAPP Builder

4.3.1 Introduction

GenMAPP Builder is a GUI application for loading, querying, and exporting data used by GenMAPP. The application is built from components in *xmlpipedbutils* and contains the customized output from *uniprotdb* and *godb*. GenMAPP Builder is our final product for converting new gene sequence data from XML into GenMAPP.

4.3.2 End-Users

GenMAPP Builder was created to be easy enough for a not-technical end user. The interface is simple and provides access to the basic functionality required to load an XML file into a GenMAPP gene database. GenMAPP Builder does however assume that a database administrator has prepared an accessible database server and loaded the UniProt and GO SQL database schemas. There are four main functionalities provided by GenMAPP Builder.

- Configure the database.
- Import a UniProt XML file.
- Import a GO XML file.
- Export to GenMAPP.

The process of loading an XML file into a GenMAPP gene database starts with configuring the database connection followed by the loading of an XML file into the relational database. The database connection must be configured to access a relational database setup by the database administrator. Once the database connection is configured the XML files can be imported and exported without effort. There is also an interface for HQL queries to the relational database provided by *xmlpipedbutils*. The screen shots of the application are below.

4.3.3 Design

GenMAPP Builder must create and fill the tables required in a GenMAPP gene database. Using a template Microsoft Access (MDB) file, GenMAPP Builder fills in the system information tables: Systems, Info and Relations. Two more stages create and fill in five more tables.

The first stage creates and fills in the Uniprot table using SQL queries. The table is populated with the needed information extracted from the relational database. The second stage requires four GO tables to be populated with data: GeneOntology, GeneOntologyTree, GeneOntologyCount, and Uniprot-GeneOntology. At runtime, each table is created in the MDB file prior data is inserted. The following sections will discuss each table.

4.3.3.1 UniProt Table The UniProt table consists of entries which have an ID, name, species and date (comments and remarks are optional). The data to populate this table comes originally from each unique entry in a given XML file. This data

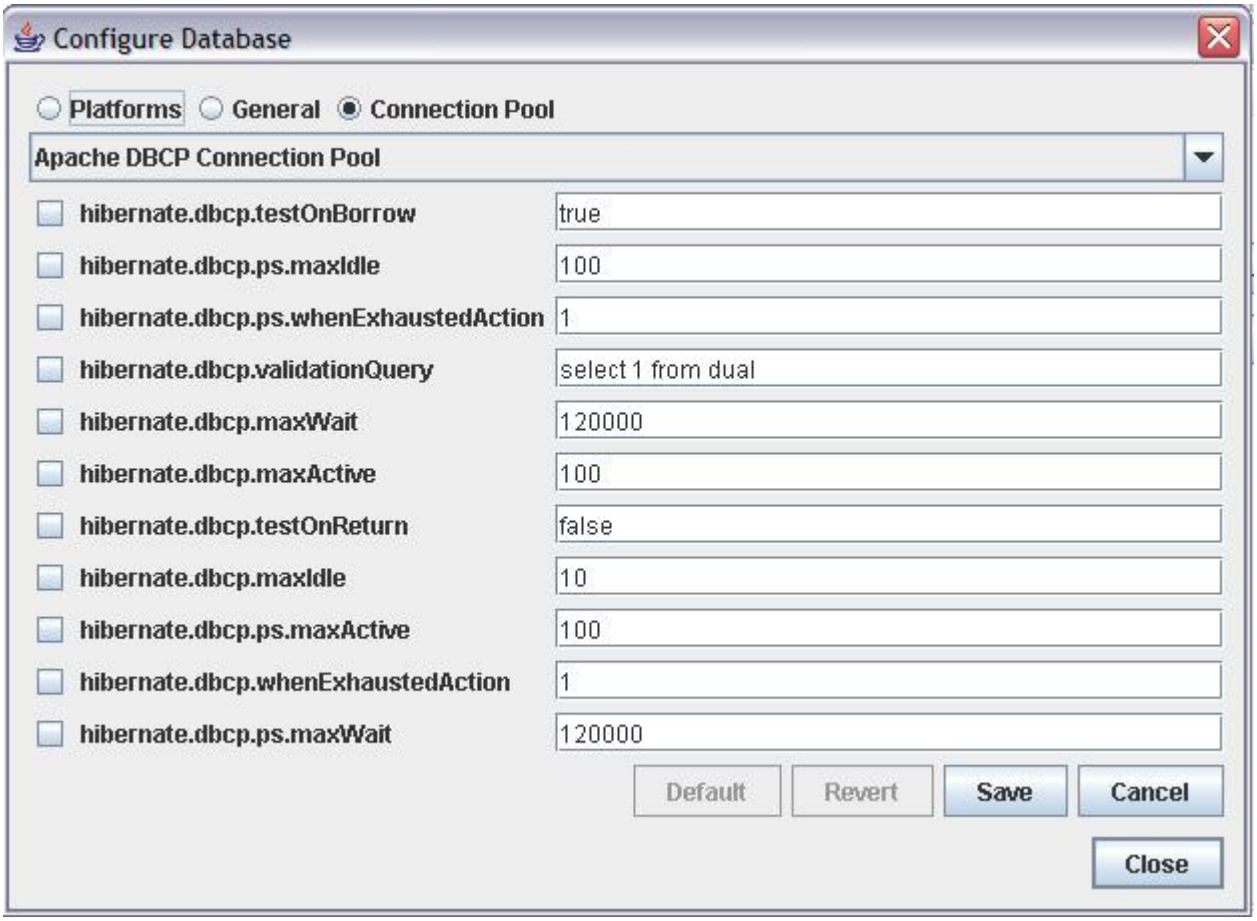

Figure 13: *The database configuration tool.*

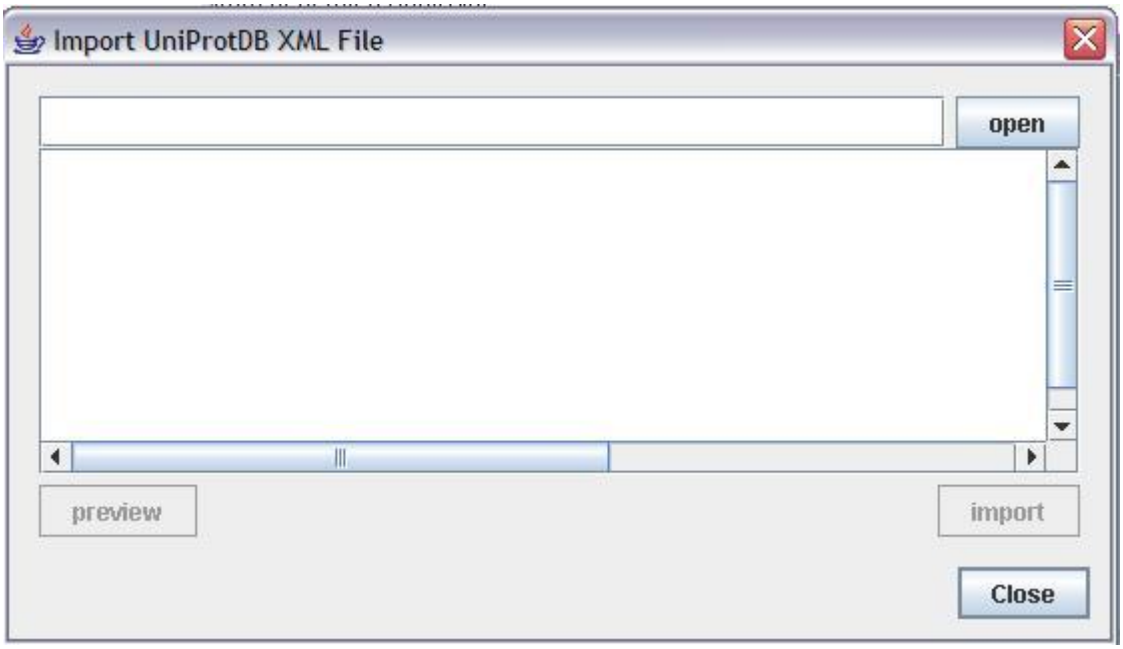

Figure 14: *The import window for UniProt XML files.*

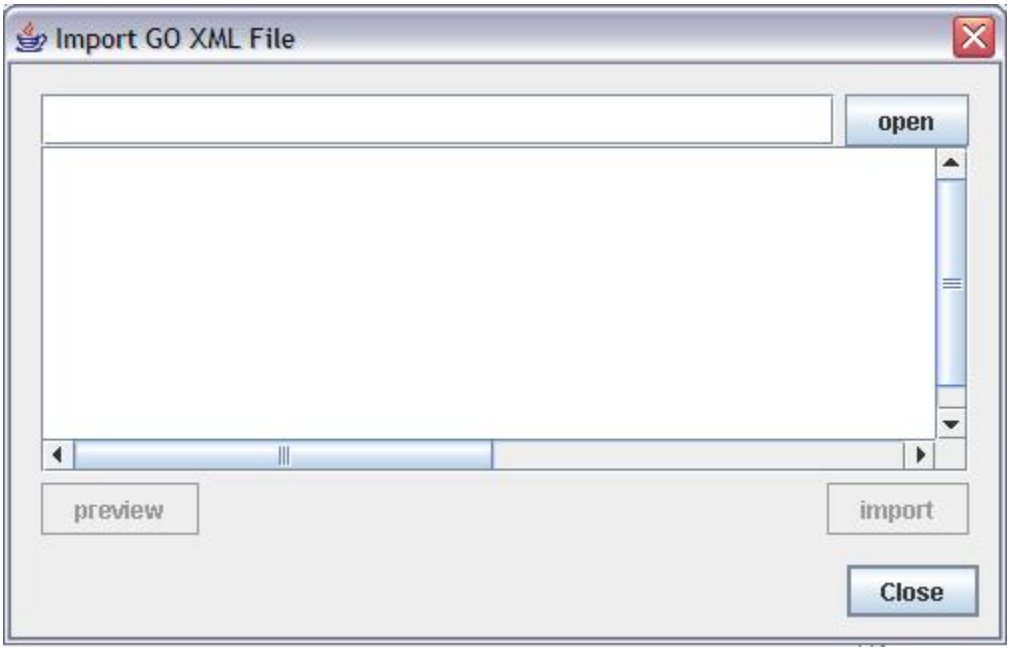

Figure 15: *The import window for GO XML files.*

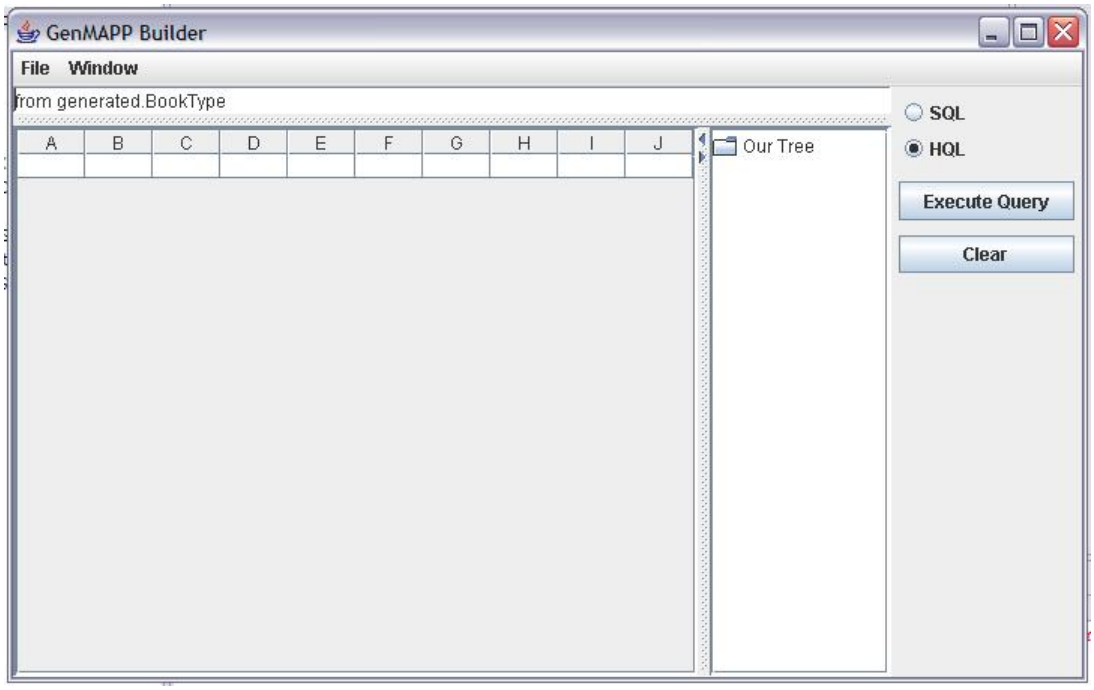

Figure 16: *The relational database query interface.*

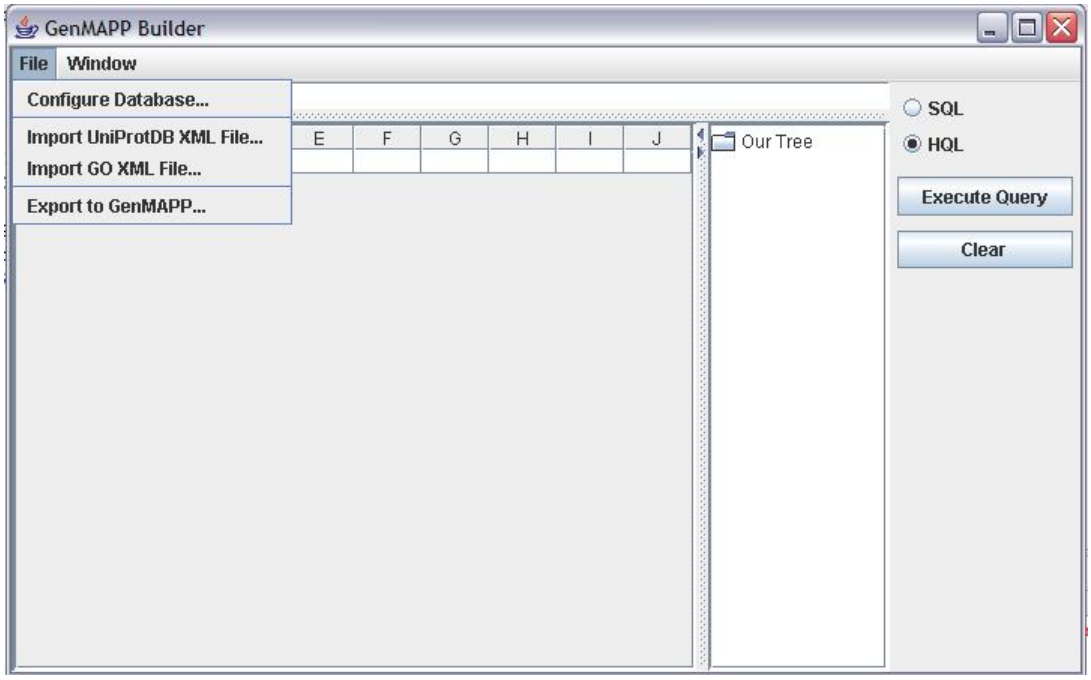

Figure 17: *The main window with the simple drop down menu.*

must be extracted from the relational database using a pattern of SQL queries. The extracted data is then filled into UniProt table in the MDB file.

4.3.3.2 GeneOntology Table The GeneOntology table consists of a list of GO terms, which is created by extracting all the GO terms from the postgres database by executing HQL statements. Each term has required tags, *Id* and *Name*. Each term may contain optional tags, one of which is an *is_a* tag. An *is_a* tag describes a subclassing relationship between one term and another. Since GO terms are organized in structures called directed acyclic graphs (DAG), each term may have many *is_a* relationships. Terms with no *is_a* relationships are roots. Each term represents an entry in the GeneOntology table. If a term has no *is_a* relationships, then one row is inserted into the table. If a term has one or more *is_a* relationships, a row is inserted for each *is_a* relationships using the same term Id.

4.3.3.3 GeneOntologyTree Table The GeneOntologyTree table contains a tree representation of the GO DAG structure, which is used by MAPPFinder to build the GO tree in the MAPPFinder browser. The algorithm that creates this table starts by inserting the root node for each ontology. After inserting the root node, it then grabs all the children of that node and performs an insert. For each child, it grabs its children and performs an insert. This process continues recursively, and stops when no more child nodes exist. Unfortunately, this table is not created from postgres, but

rather from the newly created table in Section 4.3.3.2. The OBO.dtd file creates a SQL DDL schema which does not support the ability to extract child nodes from a parent node. The GeneOntology table provides this functionality, which is why it must be created first. Thus, the algorithm actually queries the MDB Access file using SQL statements.

4.3.3.4 GeneOntologyCount Table The GeneOntologyCount table contains the of number of times a GO ID appears in the GeneOntologyTree table, which is also used by MAPPFinder to build the GO tree in the MAPPFinder browser. During the creating of the GeneOntologyTree table, a count of each GO ID is stored in a data structure. In other words, when the algorithm grabs all child nodes from a parent, each child node's count is increment by 1. When the GeneOntologyTree table is complete, the count for each GO ID is extracted from the data structure and inserted into the GeneOntologyCount table. This method was used to "kill two birds with one stone". Otherwise, the algorithm would have to wait until the GeneOntologyTree table was complete. Then for each GO ID, the algorithm would have to query the GeneOntologyTree to extract the number of times it showed up in the table, which is costly and inefficient.

4.3.3.5 Uniprot-GeneOntology Table The Uniprot-GeneOntology table provides a mapping between a Uniprot ID and a GO ID, which is a many to many relationship. Currently this information is not found in godb, but rather in an associations text file which is part of the GenMAPP Builder resource library. Each line in the text file may contain a Uniprot ID. If it does, then there will be one to many GO ID's. Thus, for each GO ID, a new row is created in the Uniprot-GeneOntology table with the Uniprot ID as one field and the GO ID as another field. For example, if the line contained a Uniprot ID of *00001* and three GO ID's *10001,10002, and 10003*, then three row would be created with values "00001 10001", "00001 10002", and "00001 10003."

4.3.4 Performance Analysis

GenMAPP Builder requires quite a bit of processing power and memory because of the large amount of data processed. Initial tests of importing large XML files (40MB) took just over an hour. Our current implementation requires a significant amount of memory because the entire file is read in as Java objects before it is pushed out to the database. Initial tests of exporting the same large XML files required nearly three hours. Memory is not as crucial for exporting as is it for importing because only a small portion of the original data is extracted to build the GenMAPP gene database.

A couple of GO errors occurred during the first run of gmbuilder: two table fields are named *Primary* and *Level*, which Microsoft Access interpreted as keywords. Additionally, the hyphen in the table *Uniprot-GeneOntology* was recognized as a continuation line, and therefore a syntax error was generated. Fortunately, the same fix was used in all three cases: enclose each name in double quotes. After the fix was implemented, gmbuilder successfully built the GO tables in the MDB file, which took considerable time.

The time to build the four GO tables ran in the neighborhood of five hours (2.0GHz single processor), and granted, this time can be shorten with better hardware. A big chunk of that time falls on the creation of the GeneOntologyTree, although this table contains roughly seven time more entries than the table with the second most entries. Still, it seemed that the rate of insertions per second was larger. A good test would be to time, call it T, the creation of the GeneOntology table by itself, then take the number of entries that were created and divide it by T. Then do the same for the GeneOntologyTree, and compare the rate of insertions per second. If they are approximately the same, then the increased time is due to the increase in entries. But, if the GeneOntologyTree insertion rate was much higher, than further investigation would be needed to why the rate is slower. Perhaps it's due to the recursive nature of the algorithm, since recursion increases overhead.

4.3.5 Future Work

Future plans involve updating the SQL queries to HQL and creating the remaining tables that aren't part of the core table set in the MDB file. The large memory requirements could be reduced by importing the XML files into memory in portions, flushing to the database when a limit is reached. Regarding the GO tables, the algorithm that populates the GeneOntologyTree table should be investigated. If it turns out that the GeneOntologyTree insertion rate in considerable larger that the other tables, then a couple of solutions can be analyzed: 1) create an equivalent non-recursive function to build the tree. This eliminates the added overhead of recursion; 2) the code can be cleaned up to close *Statement* objects before the next call to the recursive function is made. Also, the code can be modified to use *PreparedStatement* object as opposed to *Statement* objects, which may be more efficient according the java API.

As of right now, Uniprot-GeneOntology associations file is defined at a specific location in the build tree, and the code reflects that by hardcoding the path to the file. This is error prone: every time the file changes location, the code would also have to change, which supports poor coding practices. Also, what if the user wanted to use a different associations file? Then the file would have to be checked in as a new resource file, and a new version of the software would have to be released. The solution is to make this file user selectable from the genMAPP Builder GUI. This eliminate the two problems above, while add flexibility to the user.

5 Conclusion

Despite the pitfalls encountered, incredibly tight timeframes and demands of assimilating an entirely new domain of knowledge, the XMLPipeDB team has produced an initial release of outstanding quality and utility. In each area, we have identified additional work to be done, whether to add polish, tweak bugs or add new functionality. The real test will come, however, when the biological community begins to use the tool chain and reap the rewards of our effort. Our since hope is that it will meet their needs and allow them to make more interesting discoveries about the building blocks of life. Of course, there will be issues and we will rise to meet the challenge. There will also be additional databases to support and format changes. These, too, we will strive

to accomodate. Of course, one of the great advantages of this tool chain being open source is that it is not left only to us to do this work, but to anyone with the vision and the will to make it happen.

References

[Ens06] Ensembl. Ensembl genome browser, 2006. http://www.ensembl.org.

[Gen06a] GenMAPP. Creating a genmapp database for a non-supported species, 2006. http://www.genmapp.org/tutorials/Creating_a_GenMAPP_database_for_a_non-supported_species.pdf.

[Gen06b] GenMAPP. Genmapp home page, 2006. http://www.genmapp.org.

[Hyp06] Hyperjaxb2. Hyperjaxb2 - relational persistence for jaxb objects, 2006. https://hyperjaxb2.dev.java.net.

[Ont06] Gene Ontology. Gene ontology [the gene ontology], 2006. http://www.geneontology.org.

[Uni06] UniProt. Uniprot [the universal protein resource], 2006. http://www.uniprot.org.