Universiteit Leiden
Opleiding Informatica

Integrating data modeling with data analysis

in Taverna workflows

Name:	Sem Scholtes
Date:	24/07/2015
1st supervisor:	Dr. K. Wolstencroft
2nd supervisor:	Dr. Ir. F. Verbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Integrating data modeling with data analysis in Taverna workflows

Sem Scholtes

Abstract

Taverna is a workflow management system where scientific workflows can be designed and executed for in-silico experiments. Scientific workflows are used to describe, manage and share complex scientific analyses. The results of executing such workflows are often large amounts of unstructured data in strings and lists. The Data Modeling plugin has been developed to build and populate data models for managing the data during the execution of a workflow. It integrates new database services that provides users with a GUI and database metadata to make queries that create and populate tables, retrieve data and create new users. The plugin also introduces a new regex service that helps building regular expression with a GUI for building models from services that generate flat files. This thesis will describe the implementation of each of these new services. The data modeling capabilities are evaluated on how they help with workflows that have already been constructed and in the process of designing and building new workflows. The results show that the construction of a data model facilitates the comprehension of the results, usability and re-usability.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Scientific workflows are used by researchers to do in-silico experiments. These experiments use models closely reflecting the real world to conduct research via computer simulations. In a workflow different distributed services from different sources can be connected together to generate and compute new data. Different systems exists to design such workflows; a few examples are Galaxy, Kepler and Taverna. This thesis will only focus on Taverna. What Taverna is lacking are data modeling capabilities to manage and structure data. The goal is to add this to Taverna to build and populate data models during workflow execution.

1.1 Taverna

Taverna [1] is a Scientific Workflow System where scientific workflows can be designed and executed. Scientific workflows are used to describe, manage and share complex scientific analyses. They provide a high-level declarative way of specifying what it is set to achieve. Various types of tasks can be performed within a workflow where each service only performs one certain task. A service can perform this task either locally or distributed via remote web services like REST, SOAP and WSDL. This allows users to integrate different services into their workflows. By chaining these services together a sophisticated analysis pipeline can be created. An example of a simple workflow can be found in Figure 1.1. Each different kind of service has its own color. A WSDL service for example is always green. When a workflow is run services start to iterate over their given input starting from the input ports of the workflow.

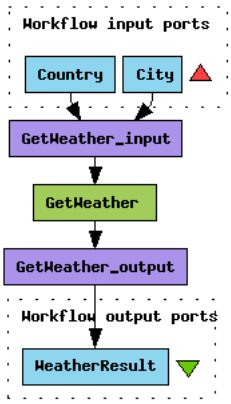

Figure 1.1: A simple workflow that retrieves a weather forecast for the specified city

Taverna has been used in several scientific domains including bioinformatics [2], astronomy [3] [4] and biodiversity [5]. In bioinformatics there is a lot of information freely available to researchers. With Taverna workflows can be created that uses this data in the form of >8000 service operations [1] for analysis and computation. For example a Taverna based toolkit helps with cancer research by connecting different services of caGrid (cancer grid) together that would otherwise be isolated. In astronomy the HELIO plugin has been developed to give access to 200 instruments from over 60 observatories to do sun related research inside Taverna. Another plugin called AstroTaverna integrated Virtual Observatory web services inside Taverna to digitally capture procedural steps that would be lost otherwise. Lastly, in biodiversity a Data Refinement Workflow for Taverna has been designed which integrates taxonomic data retrieval, data cleaning and data selection. Using a workflow makes it into a consistent and effective system that hides the complexity of underlying service infrastructures.

1.2 Problem

The result of executing workflows in Taverna is often a large amount of unstructured data in strings, lists and XML. There are workflows that produce long lists of 1000s of items. With the current Taverna setup only one list item of these lists can be viewed at once. There isn't a general overview that allows users to see all their results at once; switching between output ports and list items is required. The strings and lists a workflow produces often have relationships between each other, because they are the result of the same input data. Relationships can be found in the workflow structure itself, but are lost in the output files. This makes it hard to comprehend all the results a workflow produces.

What is is actually required is the creation and population of a database as the workflow runs. A plugin has been developed that allows users to build and populate a data model for managing the data during the execution of the workflow.

1.3 Research Question

This thesis will answer the following research question. How does the construction of a data model facilitates the comprehension of the results, usability and re-usability?

1.4 Thesis Overview

This thesis is organized as follows. This chapter explains what Taverna is and describes the problem it has that this thesis tries to solve. Chapter 2 presents the services added to Taverna and their implementation to solve this problem. Chapter 3 evaluates the use of the new services in two use cases and three criteria from the research question. Chapter 4 concludes.

Chapter 2

Methods

The Data Modeling plugin has been developed to add data modeling capabilities to Taverna. By having developed this plugin it can be compared how the construction of a data model facilitates the comprehension of the results, usability and re-usability. The plugin adds three new services that interacting with a database to create and make use of data models. They can create and populate tables, retrieve data and create new users. An extra fourth service is added to build and execute regular expressions to help modeling produced flat files.

2.1 Database

The plugin uses relational databases to build and populate data models during workflow runs. They are ideal for storing structured data and defining its relationships. Because tables can be used to represent an entity in a data model and relationships between entities can be defined by foreign keys. By using a database the models and their data persist over multiple workflow runs. This makes the data more manageable, because it is organized inside tables, can be further populated with new data and can be used again later for computations if necessary.

For database support the Data Modeling plugin uses the Java Database Connectivity (JDBC) API [6]. Taverna and its plugins are written in Java. And with JDBC it's possible to create a database-independent connectivity between the Java programming language and a wide range of SQL databases. This means the plugin isn't bound to a specific instance of a Database Management System and can connect to different systems like MySQL and PostgreSQL. The only additional requirement is a JDBC driver to handle queries and results between the client and an individual database of a specific system. When such a driver is present in Taverna's library folder, the plugin can connect to local and remote database instances of that system. The things the

plugin has to do are: creating data base connections, building the SQL queries and process the results.

This would mean that users always need to have a database setup available and provide a corresponding driver before the new services of the plugin can be used. This problem is solved by integrating a local Derby [7] database in the plugin that can be used freely without an initial setup. Taverna already has a driver for a Derby databases in its library by default. It is used to connect to its local database, e.g. to store and retrieve provenance data collected during workflow executions. The plugin uses this driver to add another local database called *tavDB*. All database services provide an option to use a this database to build and populate data models.

2.2 Database Services

Three services were created, which interact with the database:

- Create and Populate Table
- Retrieve Data
- Create User

2.2.1 Table Creation and Population

This services is used to build and populate a data model. It performs two different actions to accommodate this. Firstly, it creates a table in a database representing the instance of a data model. Secondly, it populates the table with the provided input data from the workflow. By creating multiple tables and specifying their foreign keys a data model can be realized. The table the service is set to create is configured through the interface of the service. Appendix A show how the configuration would look like. For each table its name, columns and foreign keys can be configured. Figure 2.1 shows how a service would look like inside Taverna that creates a table *Journals* with columns *journal*, *title*, *authors* and *path*. If an existing table is loaded from the interface it can be populated again without creating the table first. The input ports of the service are added depending on its configuration, where each column gets its own input port. So users can connect output ports from different services to an individual column.

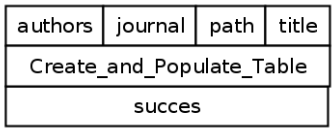

Figure 2.1: Create_and_Populate_Table-service example.

When the service is executed it starts by generating a SQL query that creates the table. The query start by listing all columns and their data type. Taverna mainly uses two data types: strings and images. This is translated to three different datatypes in the database. Varchars are used for shorter strings with a maximum length of 254. A string with a longer length here is not supported by all database systems. In case of longer strings a BLOB is used, which can store 65535 bytes maximum. For some images that Taverna produces is a BLOB too small, so they are stores within a LONGBLOB. Using a different type for each format makes it easier to distinguish types when the data is retrieved in the next service as each case needs to be handled differently. The exact data type that is going the be used, is determined by users themselves by either choosing *String*, *Text* or *Image* in the interface. After each column it specifies if the column is *notnull* and/or *unique* depending on how it is configured. At the end of the query are all the primary and foreign keys listed. Both keys are named in such a way they won't collide with other constrains. For primary keys this is a table name followed by "PK" and for foreign keys this is a combination of column and table names it involved followed by "FK". Figure 2.2 shows the structure of how the final query looks like. The service can execute this query and creates the table inside the database.

```
CREATE TABLE tablename (
columname {varchar(254)|longblob|blob} UNIQUE NOT NULL,
CONSTRAINT pkname PRIMARY KEY (pkcolummns, ...),
CONSTRAINT fkname FOREIGN KEY (column) ftable(fcolumn), ...
)
```

Figure 2.2: Create Table SQL Query Structure

After the table is created the population of it can be begin. The basic structure of the SQL query that will be used for this can seen in Figure 2.3. The number of question marks in the query depends on the number of columns that has to be populated. The query will be executed in form of a *PreparedStatement*, which adds the possibility of adding parameters. These parameters will be used to insert the actual data inside the query. This happens as the service iterates over the input lists it receives. By using a *PreparedStatement* the database knows what kind of query to expect and only has to parse it once. This might lead to a possible performance increase. The execution of each these populate queries happens in their own try Blocks. So when one query fails, the other ones will not be affected by this. This might happen when a specific constraint isn't met, like having duplicate entries with the same primary keys. If the service has finished iterating over its input list the table is fully populated with its input data.

```
INSERT INTO tablename (?, ?, ...)
```

Figure 2.3: Populate Table SQL Query Structure

A service can iterate through multiple lists when it is run inside a workflow. This causes the SQL queries to be generated for each iteration. That's why strings containing the SQL queries are stored as a global variable. This way these are generated only once per workflow run per service which saves some time.

2.2.2 Data Retrieval

Workflows should be able to use existing data models for their computations. This services is able to create queries to get data out of a database inside Taverna. The user can choose in the interface which columns from one or multiple tables need to be retrieved. Each selected column gets its own output port, so it can be connected to different services. Figure 2.4 shows how the service would look like if all columns of the *Journals* table from the previous section need to be retrieved. The names of a output port the consists of both the table and the column name. This avoids conflicts with different tables with the same column name, as port names need to be unique. There is also an option to get the results in a CSV (comma-separated values) file format. The CSV file format is commonly used in some research areas. If users check this option in the interface, a new port called *csv* is added. User can choose their own delimiter to avoid conflict with the output data. In case a column is selected that contain images it will be ignored when the file is composed.

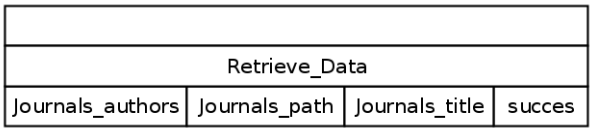

Figure 2.4: Retrieve_Data-service example.

By selecting multiple columns from different tables and specifying joins basic queries can be built to retrieve data. However, sometimes not all results are needed; only records with a specific id or property. That's why the *Retrieve Data*-service includes a basic implementation of adding condition to queries. A condition consists of three parts: a column from a table where to condition is applied to, an operation to filter the result and an input name to ask a value from outside the service. That input name is used to add a new input port to retrieve a value for the condition from the workflow itself. This makes the service more dynamic as those values are not hard coded and the service can iterate over a list of those values. A simple operation of a condition would be "=" to retrieve a record with a specific value out of the database. This value is retrieved from the input port of the condition. If multiple conditions are used they are combined with AND. This implementation of conditions doesn't cover all possible conditions that can be created, but should cover the basics needs. Sub-queries aren't supported for example, but if multiple instance of this service are used together with a Beanshell service the same result can be achieved.

```
SELECT table.columnname, ...
FROM table table, ...
WHERE table.column = jtable.jcolumn AND ...
AND conditions
```

Figure 2.5: Retrieve Data SQL Query Structure

Figure 2.5 show the final structure of how the query would look like. Executing this query gives a result set of rows. By iterating over the set each value is split into their own column list as this is the format that is needed for each output port. As mentioned earlier each data type is handled differently. The data type of each column is determined by metadata from the database. Only values with a *String* or *Text* data type are added to the csv files. Images and texts are retrieved as blobs from the database and varchars as strings.

2.2.3 User Management

There is now a service to create actual data models by populating tables inside the database and another service to create queries to get data out of the database. What is also needed is a service to create new users inside a database, because workflows are often shared with others. So when a workflow makes use of data from a database, other people should be able to do this as well. This is only the case when either a remote database is used or different users use the same local database on a shared PC. Local databases are not stored inside a workflow and thus aren't shared across PC's. To access a database credentials are needed to make a connection. Using the same credentials for ever person means the data inside the database can be tempered with. So there needs to be a way to restrict people on what they can do inside a database. This service does this by creating new users inside a database and giving them specific permissions. Figure 2.6 shows how the service looks inside Taverna.

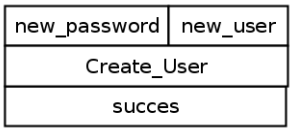

Figure 2.6: Create_User-service example.

The service has two input ports; one for usernames of the new users and one for passwords. The data it receives will be used to create new users. By default newly added users have no permissions. These permissions are added and configured through the interface of the service beforehand. Permissions can be added per table, so each table can have its own permissions. Because the usernames and passwords are provided in workflow themselves, users can be created in one batch with the same permissions in a database. If users are needed with a different set of permissions a new instance of this service needs to be used.

```
CREATE USER 'username'
IDENTIFIED BY 'password'
```

Figure 2.7: Create User SQL Query Structure

The service starts by building a SQL query that creates the user in the database. The basic structure of this can be seen in Figure 2.7. The permissions of that user are added in separate queries after this one. The basic structure of this can be seen in Figure 2.8. The available permissions to choose are: *all*, *alter*, *create*, *create user*, *delete*, *drop*, *insert*, *select* and *update*. Not all these permissions are needed if the database is solely used for data modeling inside Taverna. Only *insert* and *create* are need for building an populating a model and *select* for retrieving data from it. But to make this a complete service for creating users inside a database the other permissions were added as well.

```
GRANT permission
ON database.table
TO username
```

Figure 2.8: Add Permission SQL Query Structure

2.2.4 Configuration

The Data Modeling plugin can be installed in Taverna via the *PluginManager*. It can be found under *Updates and plugins* from the *Advanced* menu. Select 'Find New Plugins' and then 'Add update site'. Fill in the Data Modeling plugin's website [8] and select 'OK'. Check the box for the plugin and click install. All services can be found under *Data Modeling* in the *Service panel* as seen in Figure 2.9.

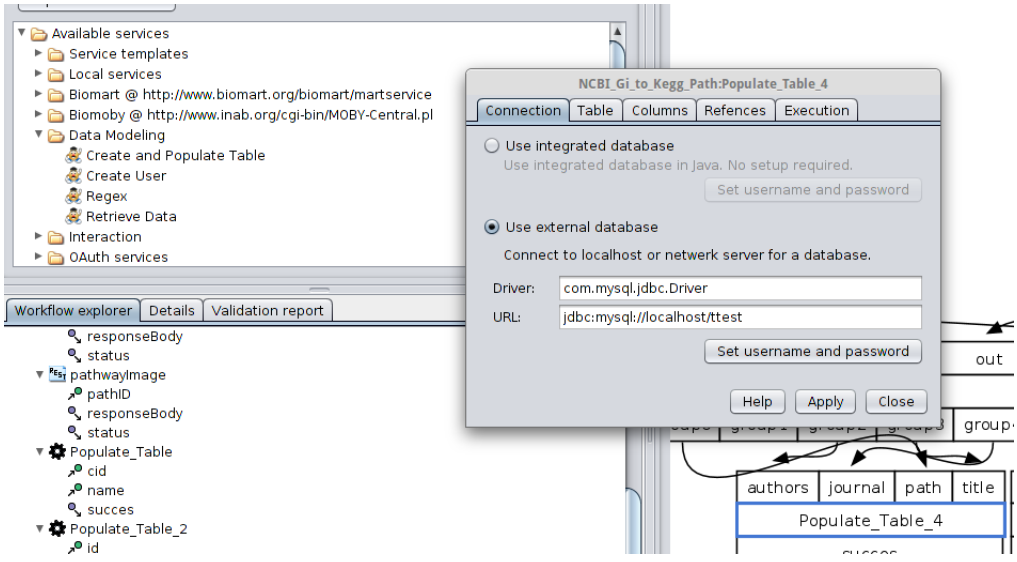

Figure 2.9: Data modeling services inside Taverna.

Each of the database services requires a database connection if they need to be executed. To establish a connection a database URL, JDBC-driver name, username and password are required. If the integrated Derby database is used only a username and password are needed. This information is added during the configuration of a service. Credentials are handled by Taverna's *Credential Manager* [9] and not by the services themselves. The manager can safely store usernames and passwords and remember for which services to use them. So if a user already has the credentials for a specific database URL inside the manager, they won't be asked again. When a user click to set his username and password, the service gives the URL to the manager which then returns the credentials back to the service. If the URL is not present in the manager, a window pops up that asks the user to fill in his credentials. This procedure is used for both the configuration and execution of a database service.

Each database service has an *succes* output port. For each execution of such a service a small report is produced to show the query that is executed and possible errors that have occured. Some possible errors that may be faced are: the username of a new user already exist, a record already exist in a table or some table constrains aren't met. This port is added for debug purposes to provide users feedback why an execution might be failed.

All database services uses database metadata to assist users during the configuring of the queries. A simple example would be configuring foreign keys were users can only select a table and column that already exists in the current database as a reference. This makes it easier for users to configure the service as specifics about the database doesn't need be remembers and typographical errors can be avoided.

2.3 Regex Service

With the database services data can be modeled that other services generate. However, not all output data from these services contains strings or list of string with single values. Some services generate flat files of multiple records with fields. In appendix B an example of such a flat file can be seen. All records with their field need to be extracted in order to create an accurate model of the data a workflow produces.

Taverna already has a XPath service to extract fields from XML data, but not all flat files are in XML format, so using it doesn't solve the problem. Something is needed that doesn't work only on a specific type of text, like XPath on XML. Regular expressions will be used to extract the data from flat files, because it is flexible as it can to extract field from different text structures. Taverna already has a few regex services, but they are designed for user who understand regex. There are a lot of scientist who don't know how to write them. This service will help them to build the regular expression they need.

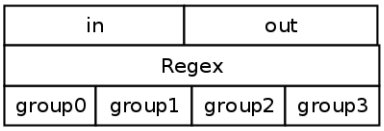

Figure 2.10: Regex-service example.

The newly developed regex service always has two input ports: *in* and *out*. The *out*-port processes the data where the regular expression is executed on. So in this case it is connected to the output of the service that generates a flat file. The *in*-port is connected to the input of that same service in order to map it to the output. This port is added, because the regex service will be used here to produce lists that goes into a table in order to model it. And often that service's input is part of a foreign key to bring over relationships into a database as it is a identifier to the flat file. By mapping it the *Create_and_Populate_Table*-service can easily take a dot-product of all lists the regex service produces. In case the regex service isn't use or data modeling the *in*-port can be left unconnected. The number of output ports the services has, depends on the configured regular expression. $group0$ is the port where the input is mapped at. Each field of a record that is extracted has its own output port, starting from $group1$.

The regular expression the service builds, is a general representation of the same kind of records of the flat file that are going to be extracted. In the regular expression each field that needs to be extracted is enclosed in parenthesis. This makes it easy to extract the field when the regular expression is executed, as parenthesis are used to capture groups. The service provides two methods to build the regular expression through the interface. Users only have to supply a flat file as an example input that the service might get. These two methods are discussed in the following sections. On the other hand this service can still be used to manually build regular expressions.

2.3.1 Interface

The interface to configure the regex service can be seen in Figure 2.11. The main elements are listed and explained below.

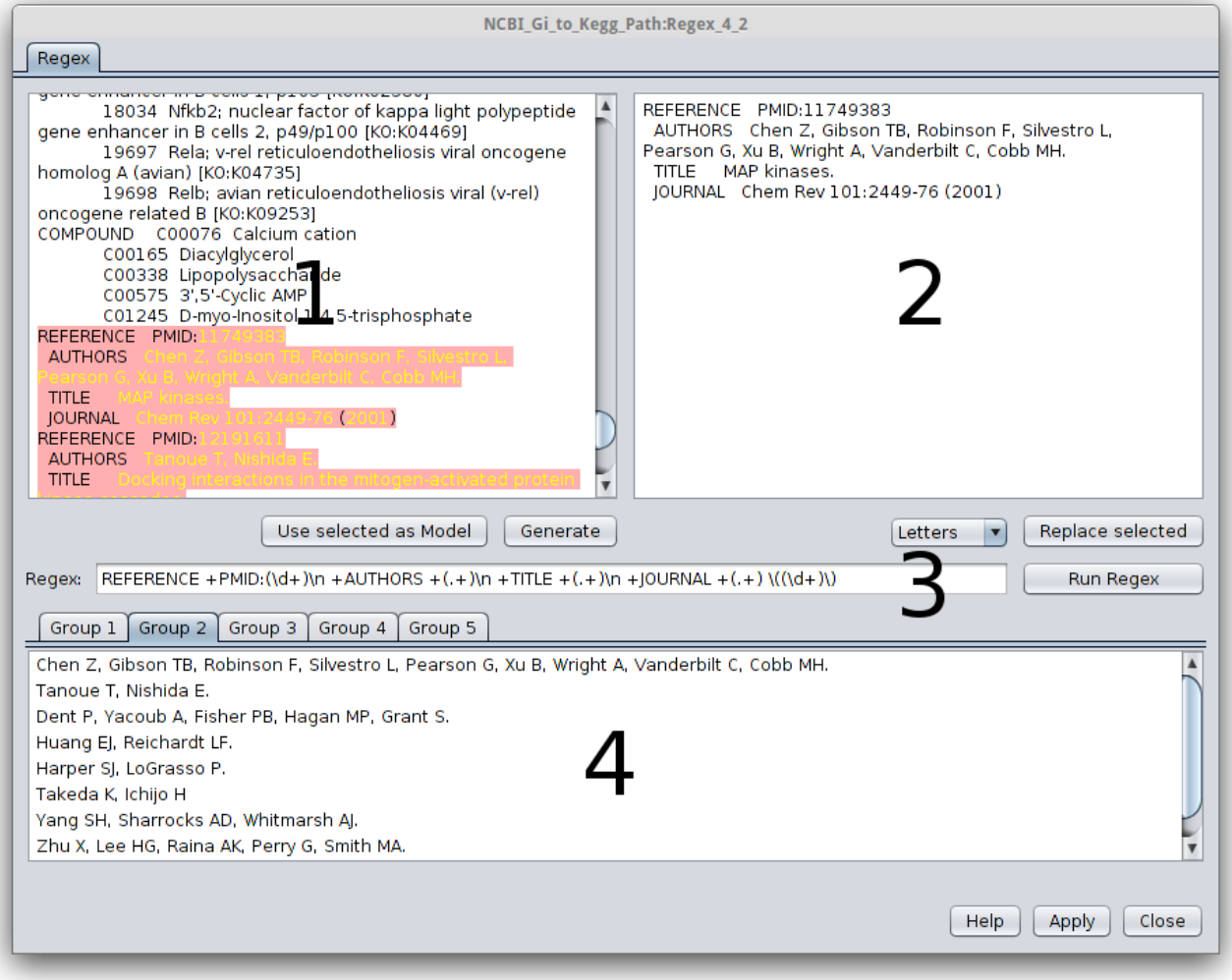

Figure 2.11: Regex configuration panel.

1. **Flat file.** Users have to provide this themselves before the service can be configured. Feedback is visually provided in this component on what the regex will extract. Text is marked pink if it is part of the full instance and colored yellow if it is a field that is going to be extracted.

2. **Data model instance.** This will be an example of an instance that the regex is going to extract. It will be used by the two methods the build the regular expression.

3. **Regular expression.** The actual regular expression that is going to be executed. It can be edited manually if needed.

4. **Preview.** The results of executing the regular expression on the provided flat files. Each field can be viewed separately.

2.3.2 Generation method

The first method to build a regular expression is by generating one automatically based on reference records. In the flat file window users select one record that is going to be use as a model instance and click on 'Use selected as model'. This string will be used as a base to build the regular expression. After this action all the records that represent the same model instance are selected. This string will be used as a reference to generate the regular expression. When the 'Generate' button is pressed, the regular expression will be automatically generated according to Algorithm 1. This regular expression can be used now on input flat files to extract all the records that have the same representation as the model instance.

```
lastAnything = false;
split model into list;
for each word in list do
    matches = countMatches(word, selected);
    if matches ≤ 0.8 ∗ instances or matches > instances then
        if word matches digitsregex then
            replace word with digitregex in regex;
        else if ... then
            ...
        else
            if lastAnything then
                remove word from regex;
            else
                replace word with anything regex (.+);
                lastAnything = true;
            end
        end
    else
        word remains in regex;
        lastAnything = false;
    end
end
```

Algorithm 1: Automatically generate regex with reference text

The algorithm starts by splitting the model text into a list of words. For each word it looks how often it occurs in the reference text. If a word does appear a regular number of time, for example in each record, it is likely a fixed part of the regular expression. If it appears irregular, then it must be a field or a part of field and must be replaced with a generalized regex part. If that is the case, the word is analyzed on what kind of pattern would match and will be replaced with a corresponding expression. A number field for example, would be replaced with "(\d+)". Some fields can consist of multiple words, which generally are some kind of sentences or descriptions. This is taken into account by looking at the next word if a irregular word is encountered that doesn't match any special pattern. All following irregular words are removed from the regular expression until a regular word is encountered. This state is remembered with the *lastAnything* boolean. The removed words are all caught in one single expression. If the algorithm has looped over all word, a regular expression remains that catches all reference records.

The advantage of this method is that doesn't require any knowledge about regular expression and is fast to build one. The service does the heavy-lifting for the user by automatically identifying the field of a record. But there are some cases where the generated regular expression is not accurate. The algorithm relies on searching how often a word of a model occurs in a part of the input flat file. When the flat file is not properly separated by spaces it can't find individual fields. An example of such a file would be an XML file and can be seen below. This is the output from the workflow pictured in Figure 1.1 in Chapter 1.1. the algorithm can't detect that in the word "<Location>Rotterdam" that "Rotterdam" is a part of field and '<Location>' is a fixed part, because they are concatenated. In this case it doesn't matter, because there is an XPath service for doing the extraction. But there are similar files that can be thought of that aren't XML, like some CSV files.

```
<?xml version="1.0" encoding="utf-16"?>
<CurrentWeather>
  <Location>Rotterdam Airport Zestienhoven, Netherlands (EHRD) 51-57N 004-27E -4M</Location>
  <Wind> from the SSW (200 degrees) at 10 MPH (9 KT) (direction variable):0</Wind>
  <SkyConditions> mostly cloudy</SkyConditions>
  <Temperature> 59 F (15 C)</Temperature>
  <RelativeHumidity> 93%</RelativeHumidity>
  <Pressure> 29.97 in. Hg (1015 hPa)</Pressure>
  <Status>Success</Status>
</CurrentWeather>
```

Another case where the generation not works is when the reference text is to small. If all reference records have the same value for a specific field, it will be treated as a fixed part of the regular expression. During execution the same kind of record can appear, but with a different value for that field. In that case the record will be ignored, because it doesn't match the generated regular expression.

2.3.3 Replacement method

As the previous method has its disadvantages a more stable method is developed and added to build a regular expression. The configuration of this method starts in the same way by selecting a model instance from the flat file. Instead of automatically identifying the fields of a record, they now have to pinpointed manually one by one. Users have to select such a field inside the model instance window and can choose what pattern that field has from a drop-down list. The service provides the following options: *Letters*, *Numbers*, *Decimals*, *Word* and *Anything*. When 'Replace selected' is pressed, the first occurrence of the selected field is replaced with the regex pattern in the regular expression. After all fields are replaced a regular expression is built that extracts all records that represents the same model instance.

This method is a little slower to configure than the previous one, but still doesn't to write the regular expressions by hand. The only pitfall is when a value of a field appears multiple times in the model instance. The algorithm works in a way were just the first occurrence is replaced with the pattern. This can be avoid by working from left to right when identifying fields.

Chapter 3

Evaluation

In the previous chapter the new services developed for Taverna were introduced and discussed. With these new services data modeling capabilities are added to Taverna. This chapter evaluates how the construction and usage of a data model can help within workflows.

3.1 Experiments

The data modeling services can be differentiated into two modes how they can be used. Either to build and populate a data model or to use an existing data model to query for results. Both of these modes will be looked into individually with a case study. The first case evaluates how the plugin helps with workflows that have already been constructed. And the second case how it helps with the process of designing and building new workflows.

One of the aspects that is going to be be looked at is the execution time of the workflows. All workflows are run in the Taverna Workbench 2.5 on the same machine with an Intel i5-3210M (2.5GHz) processor and 8GB working memory. A local MySQL database instance is used to build and use data models.

3.1.1 Use case: Existing workflow

This use case takes a look at how the newly developed services help with workflows that have already been constructed. The workflow that will be used is the *Blast Align and Tree* workflow pictured in Figure 3.1. The description of the workflow is as follows:

> This workflow accepts a protein sequence as input. This sequence is compared to others in the Uniprot database, using the NCBI BLAST Web Service from the EBI (WSDL), and the top

10 hits are returned (Nested workflow:EBI_NCBI_BLast). For each extracted hit, the Uniprot REST service returns the protein sequence in FASTA format. The workflow concatenates the 10 protein sequences and submits them as input to the EBI CLustalw service (Nested workflow EMBL_EBI_clustalw2_SOAP). These sequences are aligned and returned as results. Finally, the alignment is submitted to the EBI Clustalw_phylogeny service (Nested Workflow: clustalw_phylogeny), and a phylogenetic tree in phylip format is returned. The workflow returned a list of protein sequences in FASTA format, a Clustalw alignment, and a phylogenetic tree. [10]

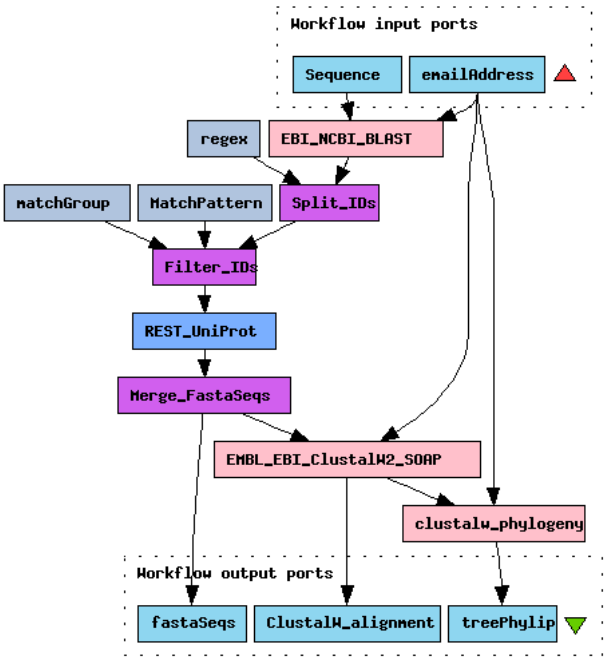

Figure 3.1: Blast Align and Tree workflow.

With the Data Modeling plugin it is possible to build a data model from the data this workflow produces. The data is worth to be modeled as it has multiple instances with underlying relationships. The *EMBL_EBI_ClustalW2_SOAP* service produces two flat files that contain five different kind of records that need to be extracted with the regex service. This results in the following workflow pictured in Figure 3.2. It builds and populates a data model with six instances including the ones from the flat files. Each instance is created by a separate *Create and Populate Table* service. The *Merge_Alignment* service creates single strings from alignment sequences that are divided over multiple lines. The strings can then be used for their corresponding field to populate a data model instance. The *ExtractSeq* service extracts a sequence identifier from a FASTA sequence in able to use it as a primary key; normal FASTA sequences are to long for this.

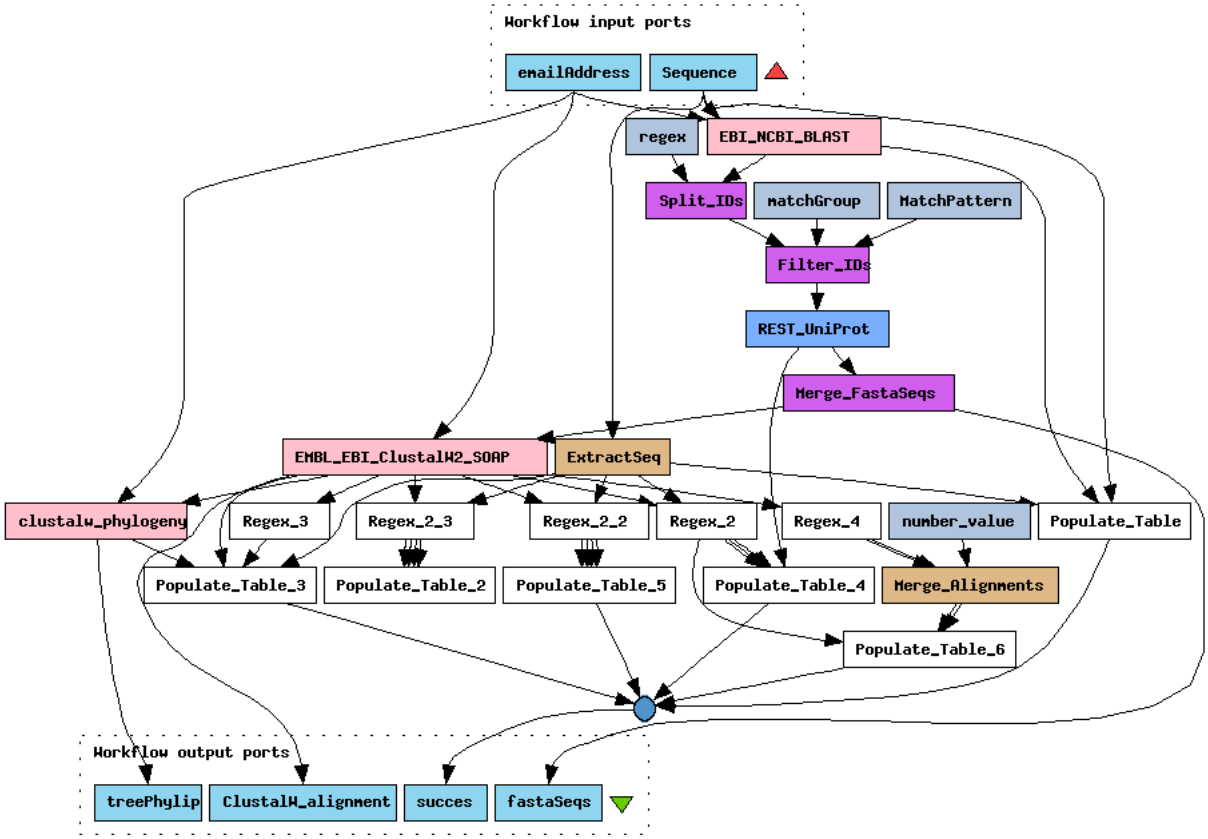

Figure 3.2: Blast Align and Tree with Data Modeling workflow.

If the two workflows are compared it is noticeable that only new services are added to the original workflow. The part that resembles the original workflow hasn't changed its form. This can be declared by the fact that the data the original workflow produces, is only processed further in order to model it. Provenance data from workflows that have already been collected only need to be analyzed to create a matching data model. After that services can be added accordingly to build a data modeling workflow. But even though new services are added, the execution time of the workflow, listed in Table 3.1, only increases by 0.1 minute. But in return the plugin helps to manage and structure data from the workflow by modeling and storing it in the database. So when data from the workflow is needed again it can be retrieved from the database instead of computing it again. The same data can be seen inside the database in form of tables and different views of it can be composed. This gives a better overview to comprehend all the results, because originally only list entries from a single port can be seen at once inside Taverna. If the modified *Blast Align and Tree* workflow is run more than once with different input sequences, the results are now stored in the same place inside the database. This makes it more convenient to compare workflow results.

Workflow	Execution time
Blast	1.2m
Blast with Data Modeling	1.3m

Table 3.1: Execution time of Blast Align and Tree workflows.

On a side note the data model from this workflow is relatively small with only six instances and simple, wide relations. If a model gets bigger and has more complex, deep relations the execution time will be larger. The main reason for this is that control links need to be added in the workflow to avoid violating the constrains from foreign keys. Control links enable to set dependencies between services; a service is only invoked if its dependency has finished. When a record added to a table, it fails if the record it is referencing to doesn't exist yet. With control links this can be avoided, but the execution takes longer because services has to wait. So in some cases the execution time might be doubled, but all other advantage of having the data modeled remain.

3.1.2 Use case: New workflow

This use case takes a look at how the newly developed services help with the process of designing and building new workflows. The workflow that will be used as a base to create these new workflows is the *NCBI Gi to Kegg Pathway Descriptions* workflow pictured in Figure 3.3. The description of the workflow is as follows:

> This workflow accepts a list of NCBI gene identifiers and returns descriptions of gene functions and a list of all pathways each gene is involved in (plus pathway image) from the KEGG database. [11]

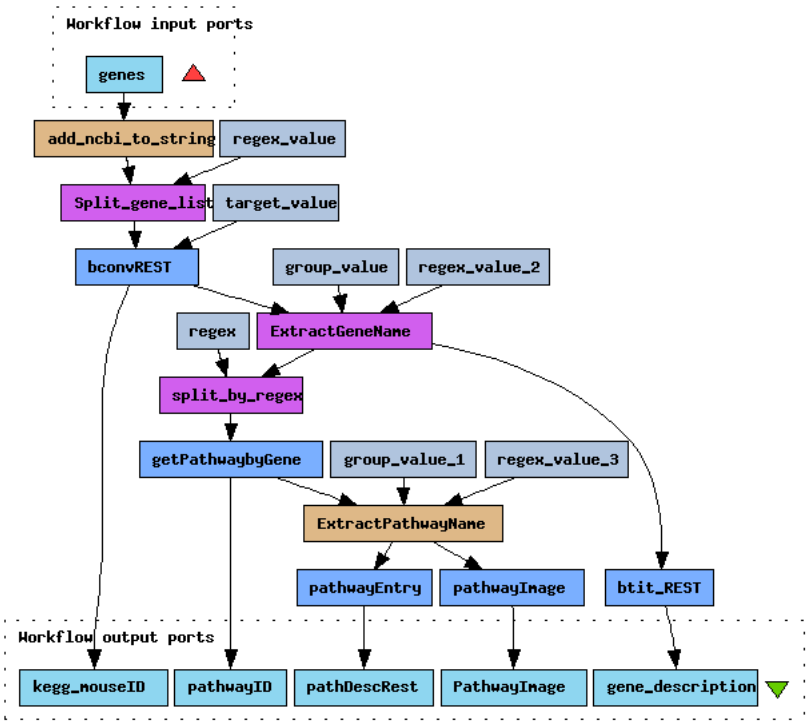

Figure 3.3: NCBI Gi to Kegg Pathway Descriptions workflow.

What often happens is that scientists have generated a list of genes that are expressed in a disease state. One thing they possibly want to know about those genes is what the functional consequences of them are. Functional properties can be assessed by looking at what functions the gene-products (proteins) have in pathways, and which are statistically over-represented. A workflow is needed that gives all those generated genes a distinctive color in their involved pathways. If multiple, different genes are present in the same pathway they should all be colored. For this use case two new workflows have been created that perform this task, but only one of them uses an existing data model. The two workflows can be compared to evaluate how the plugin helps in process and designing new workflows.

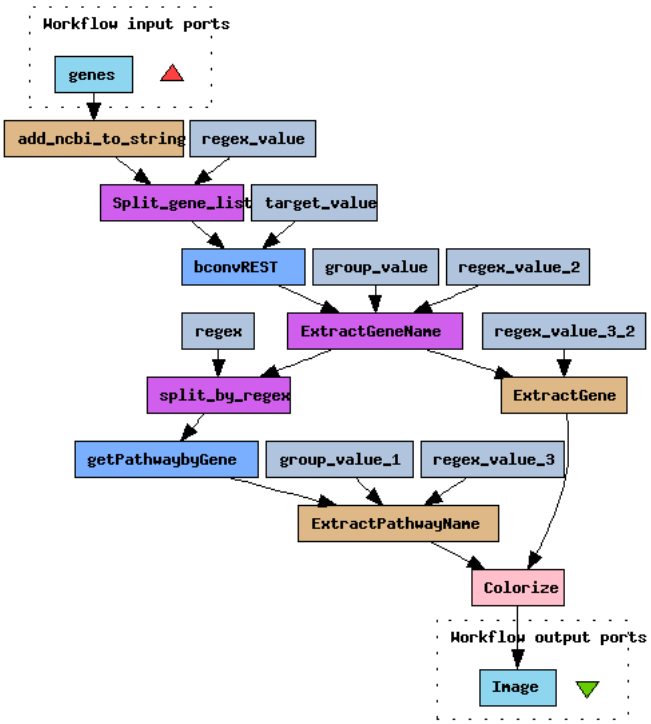

Figure 3.4: Color Pathways workflow.

The first workflow that has been created is shown in Figure 3.4 and doesn't make use of the data modeling services. The base workflow is used to create this workflow with a few additional changes. First, the *colorize* nested workflow is added to the workflow. This service produces the colored pathway images with pathways and the to be colored genes as input. The nested workflow can be seen in full detail in Appendix D. The rest of the workflow only has to supply the required data, which the base workflow is already doing. After connecting the two parts, two consecutive regex services are merged together into one and the unused services are removed to speed things up. This results in a workflow that produced the desired pathway images with colored input genes.

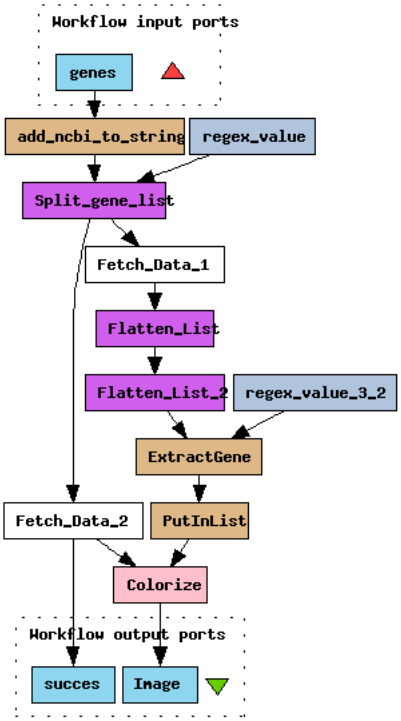

Figure 3.5: Color Pathways with Data Modeling workflow.

The second workflow, in Figure 3.5, does make use of the data modeling services. The idea here is to use an existing data model to produce the pathway images. Such a data model with information about genes and pathways doesn't exist yet, so a new one has to be created. However, this information can only be retrieved with REST services and for this workflow a data model needs to used for the evaluation. A separate workflow has been created to build and populate this new data model and will be used here. This workflow is further discussed in Appendix C.

The workflow starts by converting the input string into a list of strings with *ncbi-gi:* prefixes. The genes in the database are stored in this format. This list is used by the first retrieve data-service, *Fetch_Data_1*, to fetch the NCBI-GeneID representations of each gene. This is the format that the *Colorize* service accepts for coloring the genes. The services that follows merge all genes back in a single list to be able to process them. The other retrieve data-service, *Fetch_Data_2*, fetches for each gene its pathways they are involved in. This is essentially all data that is needed to produce the pathway images.

The two previous workflows can now be compared to evaluated how the data modeling services help with new workflows. The first thing to look at is to process of building the two workflows. In this use case the first workflow was created with one that already exists. To complete that workflow the colorize-service was added, which actually is used in both workflows. So building this workflow was fairly easy, but then again a base workflow was used. If this workflow was designed from scratch, it would be harder as all services need to be picked and the REST services need to be correctly configured. In case of the the second workflow,

which uses a data model, there is only one place to look up the data. This makes it easy to determine what information is needed as it is structured and has visible relationships. Only the *Retrieve Data* services need to be configured to build the right queries to retrieve the needed data. If the data need to be processed further some transformation services might me needed, but this is the same case when workflows are created without using the plugin. So in some cases it's worthwhile to use a data model for building new workflows as it can replace multiple different services that need to be configured. This only applies if a suitable data model exists and it is populated with sufficient data.

Using a data model to retrieve data inside a workflow brings other advantages. Workflows will be more flexible, because the *Retrieve Data* service can easily be changed if for example different input data is encountered. An example for this use case would be if the previous two workflows would have diseases as input. If this is already modeled in the same database, only *Fetch_Data_1* needs to be edited with the right columns and joins to achieve a new working workflow. Without a data model there need to be looked for specific services that produces a list of genes that involve a certain disease.

# Genes	Input string	Total Pathways
1	84579909	81
3	84579909, 224967070, 148747309	160
5	344217725, 124430578, 84579909, 224967070, 148747309	176
7	344217725, 124430578, 84579909, 224967070, 148747309, 125858479, 6677759	232

Table 3.2: Input strings used for CP-workflows

For this use case the execution times are also compared to see if there is a difference in using the plugin or not. Both workflows are tested on the same input strings that can be seen in Table 3.2. It's worth noting that not every gene has the same number of pathways they are involved in. This has as result that the execution time cannot be dependent on the number of genes the workflow receives. The total number of involved pathways of all input genes is a better indicator, because every pathway needs be colored. That's why this is listed inside the table as *Total Pathways*.

Workflow	Execution time			
Input genes	1	3	5	7
Color Pathway	6.9m	16.3m	17.7m	24.7m
Color Pathway with data modeling	7.1m	16.7m	17.7m	24.6m

Table 3.3: Execution time of CP-workflows; no parallelization

If the execution time of the two workflow are compared similar times are encountered. This can be declared by looking how each service is run in the workflow. The nested workflow service *Colorize* forms a bottle neck, because it can't keep up at the rate it receives pathways and genes. So the execution time of this workflow isn't influenced by using a data model or not. To test this statement the both workflows are also tested with parallelization added. The execution time remain similar, so it can be said that there is a bottle neck. There are still some differences between how the two different workflows run. The workflow that makes use of the data model gets *all* its pathway and gene information faster than the other workflow. But the original workflow produces its *first* result of pathways and genes earlier. This can be declared by the fact that the *Retrieve Data* service needs to iterate over the result set of the query before it produces its output lists at once. While on the other hand the first workflow makes use of a REST service that only produces one output file and can immediately work on that.

Workflow	Execution time			
# Input genes	1	3	5	7
Color Pathway	3.3m	7.7 m	8.9m	13.2m
Color Pathway with data modeling	3.2m	7.8m	8.9m	13.2m

Table 3.4: Execution time of CP-workflows; 10 parallel jobs

3.2 Criteria

The construction of data models has to be evaluated on how it facilitates the comprehension of the result, usability and re-usability. Workflows are tested on comprehension of result to see if the initial problem of having a large amount of unstructured data is solved. The new services that are used to build and populate data models should be easy to use, so usability is one of the evaluation criteria. Workflows are often shared between researchers. Re-usability will look into how workflows that use the data modeling services act on different systems.

3.2.1 Comprehension of the results

One of the main criteria that need to be evaluated is the comprehension of the result. Chapter 1.2 discussed that workflows often produce large amounts of unstructured data in strings and lists. With the developed plugin new workflows can be designed that build and populate data models to structure and store their data inside a database. When the results are stored inside a database it can be viewed in a table format. Users can see an overview of their results of multiple output records at once; for example all the outputs from a service. Different views with specific conditions can be composed from multiple tables to give a better understanding

of output data. However, this data isn't only restricted to results of one workflow run. Even if a workflow is run multiple times with different input data, the generated data will be stored in the same tables inside the database. This allows users to compare these multiple runs of the same workflow more easily as they now are conveniently stored in the same place. This is a useful addition, because workflows are hardly run ever once. Without the use of the plugin switching between list entries and ports is still required to get an overview of the result. Or all result need to merged in a huge flat file, but this is hard to manage if the workflows are run multiple times. Users will either end up with multiple files or need to extend the file, which makes it hard to compare the results.

The only downside is that to get the overview of the results a connection to the database has to be made outside of Taverna. This can be done either via the command line or with dedicated software. On the other hand before it wouldn't be possible at all. The same applies to the relations that are modeled inside the database. They are stored as foreign keys, but there is no entity-relationship diagram in Taverna that shows these relationships. This is something that can be added later to Taverna. The code is already there to retrieve actual from the database and metadata about the tables, i.e. when a existing table configuration is loaded and the data retrieval service is executed. Only the visual aspect isn't currently implemented

3.2.2 Usability

The plugin should improve the usability to make tasks more easy then they currently are. Without the new database services users have to use the native JDBC service to interact with a database. Figure 3.6 shows how that service looks. All SQL queries has to be written manually and the only way to see if they are valid is during execution. Results are only available in list of lists or XML format, which require extra services if only specific columns of the results are needed separately. There isn't an easy way to build a data model by creating and populating a table with this service. This requires changing the script that the service executes every time and additional services to redirect multiple data columns to the *params*-port.

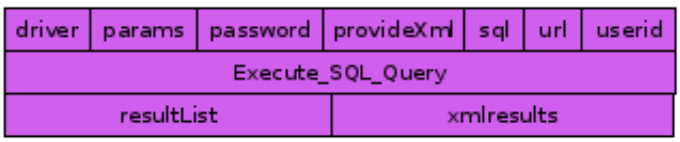

Figure 3.6: Execute SQL Query-Service.

The database services that the Data Modeling adds to Taverna don't require to write any SQL queries by hand. So users who don't know SQL can use the services as well. All queries, e.g. to create a table or get receive data from a table, are built by the services themselves. Configuring the queries happens through a graphical user interface. The interface makes use of metadata about the tables that exist in the connected

database. So when, for example, a reference from one table to another needs to be added, users can select table and column names that exist in the database from a drop-down list. This way users can easily create their references without having to remember exact column names or making typographical errors. Both the *Create and Populate Table* and *Retrieve Data* service have separate port for each column, so data can easily be connected from and to other services.

The plugin also provides users with an option to use an integrated database. So users don't need a initial database setup to start building and populating a local database. This makes the services more approachable to people who don't want to or can't setup a database themselves. If a connection to a external database has to be made, it is likely that a driver still needs to be places in Taverna's library folder.

Originally, users have to know how to write a regular expressions if they want to use the existing regex services in Taverna. The new regex service the plugin introduces provides two ways to the build a regular expression through a graphical user interface. Users are now able to build their own regular expressions even if they don't know how to write them. The interface also contains a preview of the regular expression executed on the user provided text. This gives immediate feedback if the regular is correctly configured. The service also allows the extract multiple field from a strings at once. Normally multiple services are needed to do this, now they can be managed in one single service.

3.2.3 Re-usability

A key point of scientific workflows is that they can be shared among researchers. They either use workflows on their own input data or expand workflows with new computations. If workflows with data modeling services are shared they should still work on other systems. When a researcher receives a workflow that populates a table in the local integrated database it is likely that the table doesn't exist yet. The service takes this into account by creating the table again before populating it. If it is an external database this is no problem, because the table already exists. The only problem that can occur here is if people use a local non-integrated database in their shared workflows. The workflow will look for this database during execution, but might not exists. This can be avoided by switching to the integrated database before sharing a workflow or after receiving one.

Accessing databases requires the credentials of a user to establish a database connection. These credentials are saved inside Taverna's credential manager; not in the workflows themselves. If a user doesn't have these credentials when the shared workflow is run, a window shows up during the execution where these can be filled in.

Chapter 4

Conclusions

The goal of this project was to create a plugin for Taverna to build and design data models during workflow execution. This is now possible with the Data Modeling plugin that adds four services to Taverna. The services are helpful to both existing workflows and workflows that need to be created. For existing workflows it is now possible to structure all of the data it produces and define relationship between them. This makes workflow results more comprehensible and comparable as the results can be seen organized in a database. If the data is modeled it can be used again to create new workflows. The advantage of this is that it is easier to retrieve specific data, because it is all in one place and simple queries can be created that would require multiple services otherwise. The data doesn't have to be computed again, but only retrieved from the database.

The database services provides users with a GUI and database metadata to make queries to create and populate tables, retrieve data and create new users. This makes it easier for people to build their own queries without knowing SQL well. The same has been done for the new regex service. With the help of the GUI users can use two different methods to build a regular expression, even if they don't know how to write them. For users who understand regex the service offers a preview of the results of an executed regular expression and can be used to extract multiple groups at once. The plugin also offers an integrated Derby database that can be used to build and use data models without an initial setup.

The re-usability of the workflows that uses the new data modeling services is assured in several ways. If users receive a workflow that populates certain model, tables are re-created again if these don't exist yet. Credentials for creating a database connection are stored separately in Taverna's Credential Manager. If a workflow is executed where the credentials are not present in the manager, a window will pop up to ask the users to fill them in. The workflow will continue normally and the credentials won't be asked again next time. By using a remote database researchers can collaborate on the same data model and use each others

data again.

For future work the plugin can be extended to include more controls to retrieve data from the database. A simple implementation of condition is already included, but not all possibilities are covered. To increase the usability of the plugin a perspective could be added in Taverna to view data inside the database and relationships in a entity-relationship diagram. Currently, this can be done a via the command line or with dedicated software outside Taverna.

Bibliography

[1] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P. Balcazar Vargas, Shoaib Sufi, and Carole Goble. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561, 2013.

[2] Wei Tan, Ravi Madduri, Alexandra Nenadic, Stian Soiland-Reyes, Dinanath Sulakhe, Ian Foster, and Carole A. Goble. Cagrid workflow toolkit: A taverna based workflow tool for cancer grid. *BMC Bioinformatics*, 11:542, 2010.

[3] Anja Le Blanc, John Brooke, Donal Fellows, Marco Soldati, David Prez-Surez, Alessandro Marassi, and Andrej Santin. Workflows for heliophysics. *Journal of Grid Computing*, 11(3):481–503, 2013.

[4] J.E. Ruiz et al. Astrotaverna building workflows with virtual observatory services. *Astronomy and Computing*, 7-8:3–11, November-December 2014.

[5] Cherian Mathew et al. A semi-automated workflow for biodiversity data retrieval, cleaning, and quality control. *Biodiversity Data Journal*, 2:e4221, 2014.

[6] Oracle. Jdbc overview. `http://www.oracle.com/technetwork/java/overview-141217.html`.

[7] Apache. Apache derby. `http://db.apache.org/derby/`.

[8] Sem Scholtes. Data modeling plugin. `http://liacs.leidenuniv.nl/~s1208470/plugin/`.

[9] MyGrid. Credential manager - taverna 2.5. `http://dev.mygrid.org.uk/wiki/display/tav250/Credential+Manager`.

[10] K. Wolstencroft. myexperiment - workflows - blast_align_and_tree (katy wolstencroft) [taverna 2 workflow]. `http://www.myexperiment.org/workflows/3369.html`.

[11] K. Wolstencroft. myexperiment - workflows - ncbi gi to kegg pathway descriptions (katy wolstencroft) [taverna 2 workflow]. http://www.myexperiment.org/workflows/2659.html.

Appendix A

Database Services Interfaces

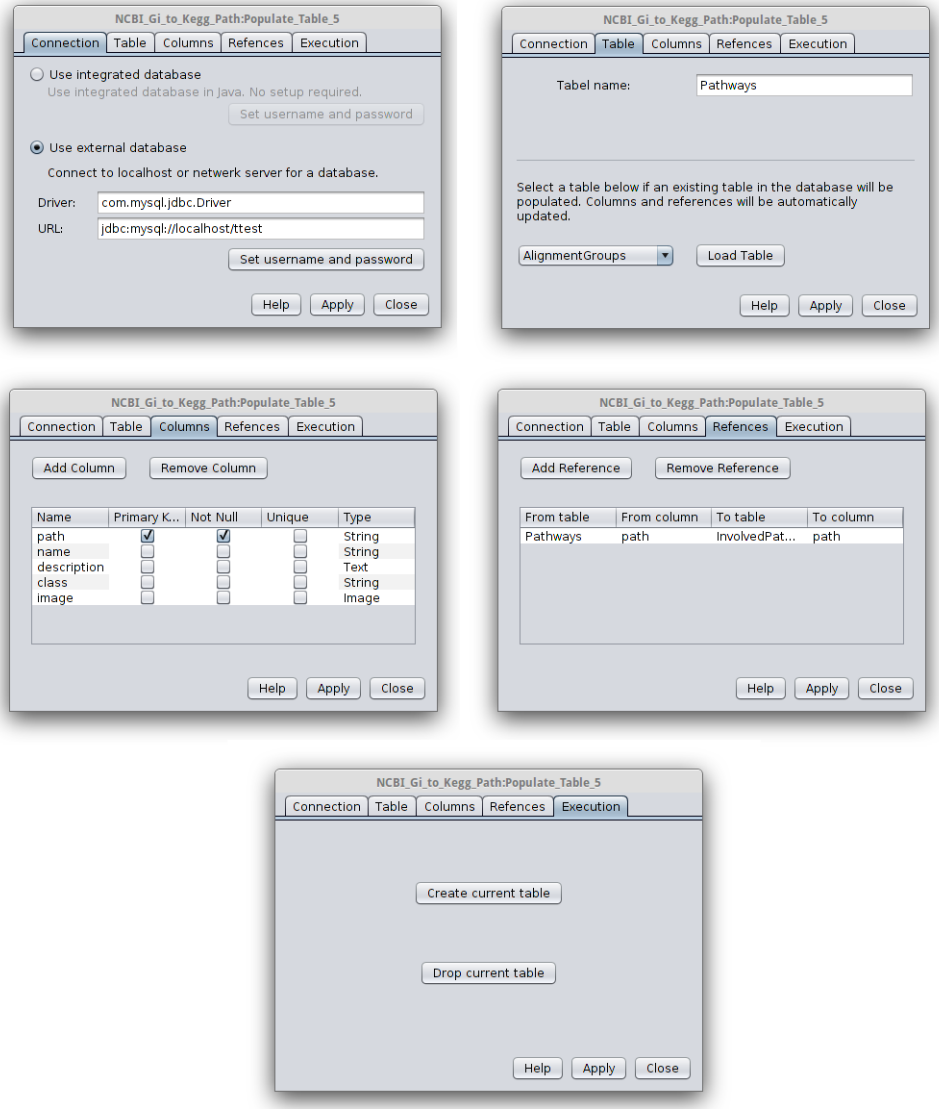

Figure A.1: Create and Populate Table Interface.

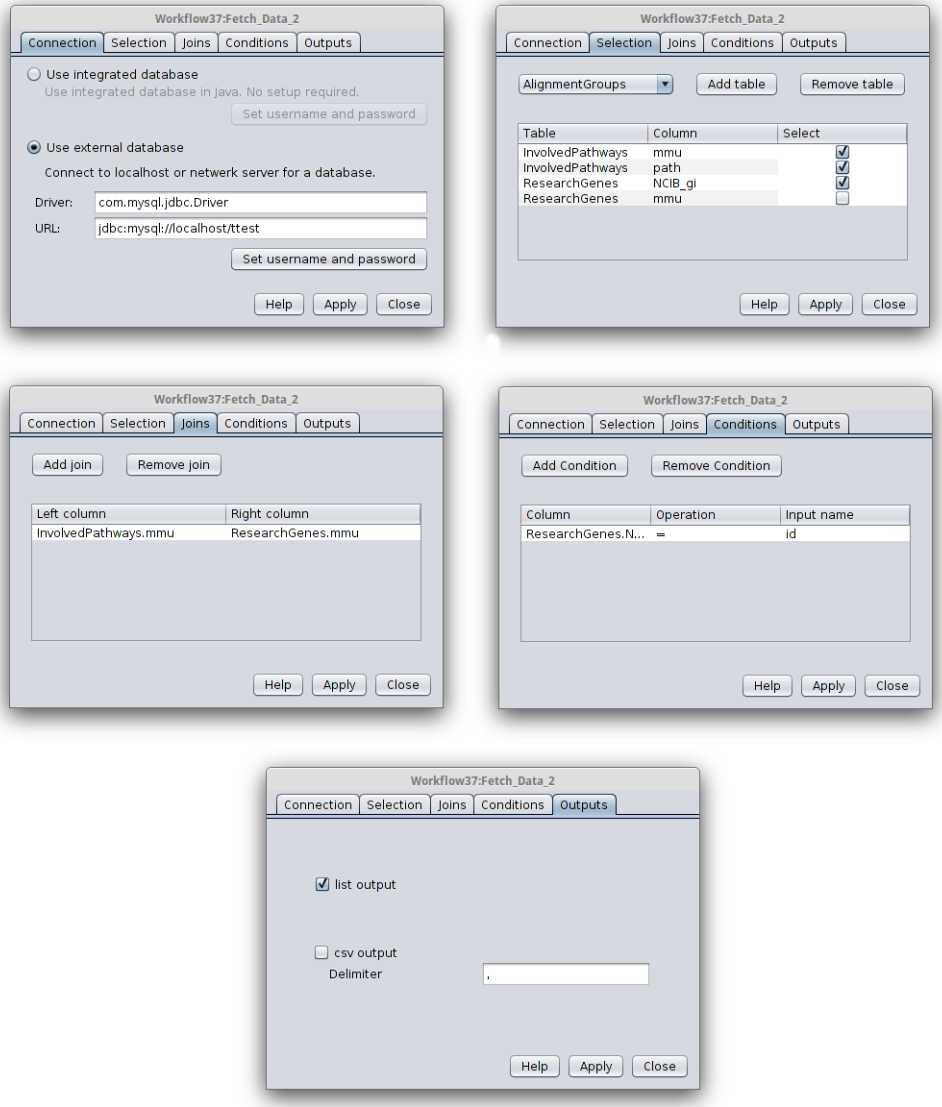

Figure A.2: Retrieve Data Interface.

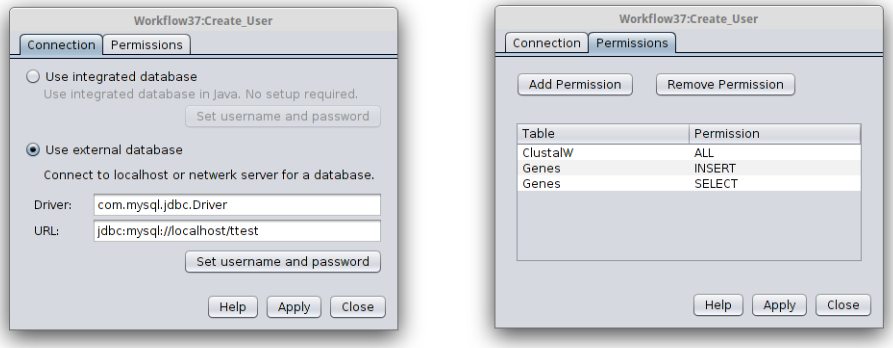

Figure A.3: Create User Interface.

Appendix B

Flat file example

This flat file is generated by the *pathwayEntry* service from the workflow in Fig 3.3. It gives a description of a pathway which including its genes, compounds and references made in journals. To building a data model from this workflow several regex services are needed to extract records with their fields from the flat file. An example of a data model instance would be a *References* with its fields *pmid*, *authors*, *title*, *journal* and *year*.

```
ENTRY       mmu04730                    Pathway
NAME        Long-term depression - Mus musculus (mouse)
DESCRIPTION Cerebellar long-term depression (LTD), thought to be a molecular and cellular basis for cer
CLASS       Organismal Systems; Nervous system
PATHWAY_MAP mmu04730  Long-term depression
DBLINKS     BSID: 83283
            GO: 0048169
ORGANISM    Mus musculus (mouse) [GN:mmu]
GENE        18125  Nos1; nitric oxide synthase 1, neuronal [KO:K13240] [EC:1.14.13.39]
            60596  Gucy1a3; guanylate cyclase 1, soluble, alpha 3 [KO:K12318] [EC:4.6.1.2]
            234889  Gucy1a2; guanylate cyclase 1, soluble, alpha 2 [KO:K12318] [EC:4.6.1.2]
            239134  Gucy1b2; guanylate cyclase 1, soluble, beta 2 [KO:K12319] [EC:4.6.1.2]
            54195  Gucy1b3; guanylate cyclase 1, soluble, beta 3 [KO:K12319] [EC:4.6.1.2]
            19091  Prkg1; protein kinase, cGMP-dependent, type I [KO:K07376] [EC:2.7.11.12]
            19092  Prkg2; protein kinase, cGMP-dependent, type II [KO:K07376] [EC:2.7.11.12]
            19051  Ppp1r17; protein phosphatase 1, regulatory subunit 17 [KO:K08067]
            51792  Ppp2r1a; protein phosphatase 2, regulatory subunit A, alpha [KO:K03456]
            73699  Ppp2r1b; protein phosphatase 2, regulatory subunit A, beta [KO:K03456]
            19052  Ppp2ca; protein phosphatase 2 (formerly 2A), catalytic subunit, alpha isoform [KO:KO
            19053  Ppp2cb; protein phosphatase 2 (formerly 2A), catalytic subunit, beta isoform [KO:K04
            15461  Hras; Harvey rat sarcoma virus oncogene [KO:K02833]
            16653  Kras; v-Ki-ras2 Kirsten rat sarcoma viral oncogene homolog [KO:K07827]
            18176  Nras; neuroblastoma ras oncogene [KO:K07828]
            11836  Araf; v-raf murine sarcoma 3611 viral oncogene homolog [KO:K08845] [EC:2.7.11.1]
            109880  Braf; Braf transforming gene [KO:K04365] [EC:2.7.11.1]
            110157  Raf1; v-raf-leukemia viral oncogene 1 [KO:K04366] [EC:2.7.11.1]
            26395  Map2k1; mitogen-activated protein kinase kinase 1 [KO:K04368] [EC:2.7.12.2]
            26396  Map2k2; mitogen-activated protein kinase kinase 2 [KO:K04369] [EC:2.7.12.2]
            26413  Mapk1; mitogen-activated protein kinase 1 [KO:K04371] [EC:2.7.11.24]
            26417  Mapk3; mitogen-activated protein kinase 3 [KO:K04371] [EC:2.7.11.24]
            14804  Grid2; glutamate receptor, ionotropic, delta 2 [KO:K05207]
```

```
            14816  Grm1; glutamate receptor, metabotropic 1 [KO:K04603]
            14677  Gnai1; guanine nucleotide binding protein (G protein), alpha inhibiting 1 [KO:K04630
            14679  Gnai3; guanine nucleotide binding protein (G protein), alpha inhibiting 3 [KO:K04630
            14678  Gnai2; guanine nucleotide binding protein (G protein), alpha inhibiting 2 [KO:K04630
            14681  Gnao1; guanine nucleotide binding protein, alpha O [KO:K04534]
            14687  Gnaz; guanine nucleotide binding protein, alpha z subunit [KO:K04535]
            14683  Gnas; GNAS (guanine nucleotide binding protein, alpha stimulating) complex locus [KO
            14673  Gna12; guanine nucleotide binding protein, alpha 12 [KO:K04346]
            14674  Gna13; guanine nucleotide binding protein, alpha 13 [KO:K04639]
            329502  Pla2g4e; phospholipase A2, group IVE [KO:K16342] [EC:3.1.1.4]
            18783  Pla2g4a; phospholipase A2, group IVA (cytosolic, calcium-dependent) [KO:K16342] [EC:
            211429  Pla2g4b; phospholipase A2, group IVB (cytosolic) [KO:K16342] [EC:3.1.1.4]
            232889  Pla2g4c; phospholipase A2, group IVC (cytosolic, calcium-independent) [KO:K16342] [
            78390  Pla2g4d; phospholipase A2, group IVD [KO:K16342] [EC:3.1.1.4]
            271844  Pla2g4f; phospholipase A2, group IVF [KO:K16342] [EC:3.1.1.4]
            18750  Prkca; protein kinase C, alpha [KO:K02677] [EC:2.7.11.13]
            18751  Prkcb; protein kinase C, beta [KO:K02677] [EC:2.7.11.13]
            18752  Prkcg; protein kinase C, gamma [KO:K02677] [EC:2.7.11.13]
            14682  Gnaq; guanine nucleotide binding protein, alpha q polypeptide [KO:K04634]
            14672  Gna11; guanine nucleotide binding protein, alpha 11 [KO:K04635]
            18795  Plcb1; phospholipase C, beta 1 [KO:K05858] [EC:3.1.4.11]
            18797  Plcb3; phospholipase C, beta 3 [KO:K05858] [EC:3.1.4.11]
            18798  Plcb4; phospholipase C, beta 4 [KO:K05858] [EC:3.1.4.11]
            18796  Plcb2; phospholipase C, beta 2 [KO:K05858] [EC:3.1.4.11]
            14799  Gria1; glutamate receptor, ionotropic, AMPA1 (alpha 1) [KO:K05197]
            14800  Gria2; glutamate receptor, ionotropic, AMPA2 (alpha 2) [KO:K05198]
            53623  Gria3; glutamate receptor, ionotropic, AMPA3 (alpha 3) [KO:K05199]
            17096  Lyn; Yamaguchi sarcoma viral (v-yes-1) oncogene homolog [KO:K05854] [EC:2.7.10.2]
            666513  Gm11787; predicted gene 11787 [KO:K05854] [EC:2.7.10.2]
            12286  Cacna1a; calcium channel, voltage-dependent, P/Q type, alpha 1A subunit [KO:K04344]
            16438  Itpr1; inositol 1,4,5-trisphosphate receptor 1 [KO:K04958]
            16439  Itpr2; inositol 1,4,5-triphosphate receptor 2 [KO:K04959]
            16440  Itpr3; inositol 1,4,5-triphosphate receptor 3 [KO:K04960]
            20190  Ryr1; ryanodine receptor 1, skeletal muscle [KO:K04961]
            12918  Crh; corticotropin releasing hormone [KO:K05256]
            12921  Crhr1; corticotropin releasing hormone receptor 1 [KO:K04578]
            16000  Igf1; insulin-like growth factor 1 [KO:K05459]
            16001  Igf1r; insulin-like growth factor I receptor [KO:K05087] [EC:2.7.10.1]
COMPOUND    C00025  L-Glutamate
            C00076  Calcium cation
            C00165  Diacylglycerol
            C00219  Arachidonate
            C00533  Nitric oxide
            C00641  1,2-Diacyl-sn-glycerol
            C00942  3',5'-Cyclic GMP
            C01245  D-myo-Inositol 1,4,5-trisphosphate
            C01330  Sodium cation
REFERENCE   PMID:12415297
  AUTHORS   Ito M.
  TITLE     The molecular organization of cerebellar long-term depression.
  JOURNAL   Nat Rev Neurosci 3:896-902 (2002)
REFERENCE   PMID:11427694
  AUTHORS   Ito M.
  TITLE     Cerebellar long-term depression: characterization, signal transduction, and functional role
  JOURNAL   Physiol Rev 81:1143-95 (2001)
REFERENCE
  AUTHORS   Purves D, Augustine GJ (ed).
  TITLE     Neuroscience
```

```
  JOURNAL   Sunderland, Mass.:Sinauer Associates (2004)
REFERENCE   PMID:9735948
  AUTHORS   Daniel H, Levenes C, Crepel F.
  TITLE     Cellular mechanisms of cerebellar LTD.
  JOURNAL   Trends Neurosci 21:401-7 (1998)
REFERENCE   PMID:9602501
  AUTHORS   Levenes C, Daniel H, Crepel F.
  TITLE     Long-term depression of synaptic transmission in the cerebellum: cellular and molecular mec
  JOURNAL   Prog Neurobiol 55:79-91 (1998)
REFERENCE   PMID:14509570
  AUTHORS   Metzger F, Kapfhammer JP.
  TITLE     Protein kinase C: its role in activity-dependent Purkinje cell dendritic development and pl
  JOURNAL   Cerebellum 2:206-14 (2003)
KO_PATHWAY  ko04730
///
```

Appendix C

NCBI Gi to Kegg Pathway Descriptions with Data Modeling workflow

This workflow had been created to build and populate a data model from the workflow pictured in Figure 3.3. The data model consists of seven instances that are added to the model by creating a table.

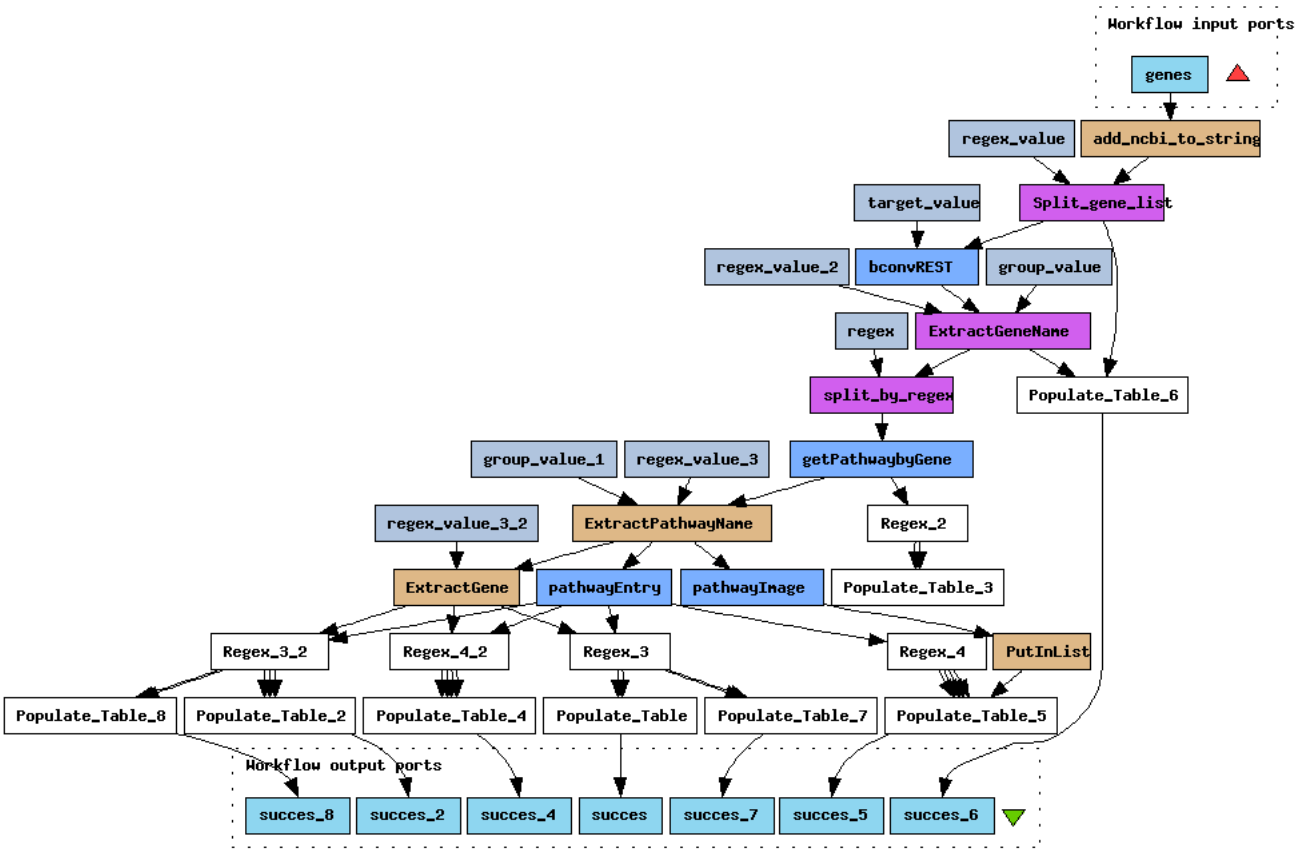

Figure C.1: NCBI Gi to Kegg Pathway Descriptions with Data Modeling workflow.

Appendix D

Colorize-workflow

The *Colorize*-workflow pictured in Figure D.1 is used in the workflows pictured in Figure 3.4 and Figure 3.5 as a nested workflow. It gives the genes specified in *ColorGenes* a distinctive color in the pathway image from *pathway*. It starts by preparing an URL directed to KEGG WebLinks where pathways can be colored. Such a URL would look like: `http://www.kegg.jp/kegg-bin/show_pathway?map=mmu04261&multi_query=12265+%23c9fcb0/20191+%23d7280b/26413+%23d9ec92/18795+%23ccca0c/224129+%232b47c4`. The services that follow gets the HTML page from the website in Taverna extract and the pathway image from that page.

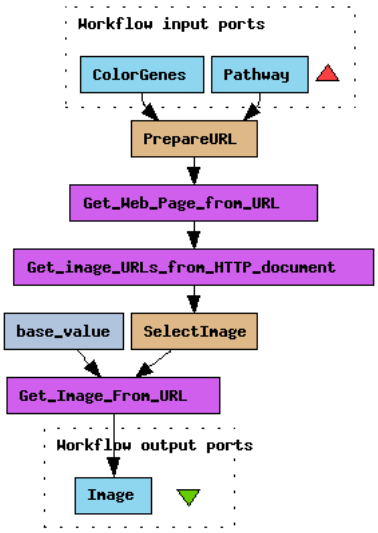

Figure D.1: Colorize workflow.

Appendix E

Source Code Plugin

All source code for the Data Modeling plugin can be found on the following web page: `http://liacs.leidenuniv.nl/~s1208470/code/`. The project can be imported into Eclipse if personal changes have to be made. Note that the plugin is developed for Taverna 2.5. Taverna plugins are currently version specific, which makes this plugin incompatible with other versions.

Appendix F

Workflows

All workflows used and referenced inside this thesis can be found on the following web page: `http://liacs.leidenuniv.nl/~s1208470/workflows/`