A modular architecture for biological microscope image analysis

by

Douglas A. Creager

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2003

Author ....
Department of Electrical Engineering and Computer Science
February 4, 2003

Certified by ....
Bruce Tidor
Associate Professor
Department of Electrical Engineering and Computer Science
Thesis Co-Supervisor

Certified by ....
Peter Sorger
Associate Professor
Department of Biology
Thesis Co-Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A modular architecture for biological microscope image analysis

by
Douglas A. Creager

Submitted to the Department of Electrical Engineering and Computer Science
on February 4, 2003 in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Open Microscopy Environment (OME) provides a standardized, open-source environment in which microscope images can be acquired, analyzed, and visualized. The OME analysis system provides a modular architecture for analyzing these images. Analysis routines are broken down into their logical components, which are coded separately as modules. Modules are linked together into analysis chains by using semantic data types to form data dependencies between the modules. Tools are being developed to allow these chains to be pieced together graphically from a toolbox of analysis modules, and to allow the user to extend the toolbox in a seamless, language-independent manner. The execution of an analysis routine against a set of images is automated, allowing the user to focus on the design of the analysis routine, rather than the details of computation, and allowing the analysis engine to perform various optimizations, such as the reuse of analysis results.

Thesis co-supervisors:

Bruce Tidor
Associate Professor
Department of Electrical Engineering
and Computer Science

Peter Sorger
Associate Professor
Department of Biology

Acknowledgements

The core OME system is being developed at the Sorger Lab in MIT's Department of Biology; the Wellcome Trust Biocentre in Dundee, Scotland; the Danuser Lab at ETH Zürich; the Institute of Chemistry and Cell Biology at Harvard Medical School; and the Image Informatics and Computational Biology Unit at the National Institutes of Health.

Erik Brauner, Ilya Goldberg, Brian Hughes, Josiah Johnston, Peter Sorger, and Jason Swedlow have been invaluable in their help with the development and testing of the OME project in general, and the analysis system specifically.

The author would also like to thank Mike Bonnet, Enid Choi, Dennis Gregorovic, Rich Hanna, Todd Nightingale, Brian Pharris, Derik Pridmore, Jon Salz, and Kevin Schmidt for their support throughout the development of this project.

Table of Contents

List of Figures

CHAPTER 1
Introduction

Much of modern biological research revolves around the acquisition and analysis of images of cells and proteins obtained from high-powered optical microscopes. By imaging cells that carry fusions between proteins of interest and Green Fluorescent Protein (GFP) (or its spectral variants), the biologist can capture rich, detailed images that contain a wealth of biological information [1-3]. Two major paradigms have developed for the use of these images in the context of a biological experiment. In the first, cells are divided into groups and each treated with one of a group of hundreds of chemicals, cDNA's, or inhibitory RNA's. The microscope images are then used to search for and screen the phenotypic variations in the effects of each chemical. In the second, smaller groups of cells are modified genetically or treated with the same chemicals. Time-lapse movies of these cells are then captured by the microscope, looking for phenotypic changes which are not necessarily static; dynamic changes in the cell cycle, for instance, can be picked out of the resulting images [4]. In the general case, images obtained from the microscopes are five-dimensional in nature: They contain the usual three spatial dimensions, one of which is much less precise in spatial granularity, plus one spectral and one temporal dimension.

Fully digital microscopes, in which these images are obtained with CCD's and sent directly to a computer to be stored and analyzed, are now standard. The usual workflow for digital imaging can be broken down into four main areas:

1. Acquiring images from the microscope;

2. Transforming the images and analyzing their content for relevant biological information;
3. Visualizing the images and the analysis results;
4. Archiving the images for future reference.

Tools exist to aid the biologist in all four of these areas. However, as things currently stand, there are several problems that hinder quantitative research. First, there is no standard file format for storing images. Rather, each microscope manufacturer uses its own proprietary format. Not only are the details of encoding the image different; often, even the logical structure of the file formats vary quite dramatically. Most common image formats, such as the Tagged Image File Format (TIFF), around which most microscope image formats are based, only store two-dimensional images. As such, there is no standard way for encoding a five-dimensional image in a two-dimensional file. Some file formats solve this problem by creating discrete two-dimensional files for each XY plane in an image. This approach has the problem of separating the image data into multiple pieces, all of which must be kept together for an image to remain valid. Another approach is to hack the two-dimensional format to hold all five dimensions. While files such as these are easier to keep track of, they still stuffer the disadvantages of not being standardized; each manufacturer modifies the two-dimensional format in different ways to support five-dimensional images.

The proprietary nature of the file formats is, in itself, another problem. Since the design of the acquisition software is closely coupled with the design of the microscope itself, the code is understandably proprietary and closed-source. However, the file format of the resulting images is usually proprietary as well. As long as an experimenter sticks to microscopes provided by a single manufacturer, this does not pose a problem. However, most biology labs contain several microscopes, built by different companies. Ensuring compatibility between all of the images and programs quickly becomes a logistical nightmare.

Another problem with the current arrangement lies in the analysis software available to the biologist. Most analysis packages that come from the microscope manufacturers provide the user with a fixed set of routines that can be used to analyze images. If an analysis that would be helpful in investigating a given image is not available in the package provided with the microscope, then the biologist is out of luck. There are few ways to transfer the image to another program that does have the desired routine; further, there is no way to extend the package to support the new routine. This turns out to be a problem fundamental to the way images are analyzed in a biological context. Biology experiments are extremely open-ended; the analysis routines that would be useful in any given instance are entirely dependent on the details of the experiment being performed. This means that a fixed set of analysis routines can never be general enough to support useful biological research. Instead, an analysis system where the user is free to create new analyses, is not just useful, but necessary.

Lastly, the idea that image analysis depends greatly on the details of the experiment means that entire categories of metadata are not being captured appropriately. None of the existing systems are built around the fundamental idea that to be truly useful, an image must not just consist of pixels. Rather, it should be the union of the actual image data along with all of its experimental metadata. Scientific knowledge is not just the results of an experiment, but the synthesis of the results into an understanding of the experimental subject. To be understood in their proper context, experimental details must be recorded at the time of image acquisition, and then maintained with the image and its analysis results for its entire life.

All of these problems surface not only during the actual experimentation and analysis portions of the biologist's workflow. They also can severely inhibit the ability for real scientific results to be published. Proprietary image formats greatly reduce the number of people who can actually view, let alone validate and reproduce, the data used to make the scientific claims. This problem is compounded by the difficulty in keeping the metadata associated with their appropriate images. This can result in incorrect images being used in a publication, and also prevents a complete operational record of the analysis results from being retained.

1.1 Open Microscopy Environment

The Open Microscopy Environment (OME)[1] is being developed to answer these problems. The main components of the OME system are:

1. a standardized five-dimensional file format;

2. a data model and XML specification for storing all image metadata together;

3. a workflow-based user interface for collecting, analyzing, and visualizing images.

These three components, which will be released under an open-source license, provide a solution to the problems presented above. The standardized file format would eliminate the difficulties in sharing images by providing a general transfer format that would be readable by all imaging applications. The data model would provide a single, consistent location for all of the metadata associated with an image, whether it was entered by the user or generated as a result of an analysis routine. It would also provide all of the information necessary to reconstruct the operational record of every analysis result.

The initial alpha version of OME, released in December 2001 by Ilya Goldberg in the Sorger Lab at MIT, developed the data model and XML specification, and showed the power of tightly coupling image data and metadata. Jason Swedlow's lab at the Wellcome Biocentre Trust used this OME release to analyze

[1] http://www.openmicroscopy.org/

the movement of intra-nuclear organelles (Cajal bodies) and find results that were not obvious and would not have been easy to obtain using existing conventional tools [5].

However, this initial version also highlighted the difficulty in designing a robust analysis system that met all of the OME goals and would also help assuage the problems mentioned previously. The system provided the user with a series of analysis routines, and made it possible to run the routines against the datasets that had been collected. The results from those routines were placed in database tables, becoming just as coupled to the original image as metadata that had been captured at image import. Using standard SQL queries, biologists could ask meaningful questions about the data and get useful answers. However, it was still difficult to add new analyses to the system, and the record of the analyses performed was incomplete.

1.2 Towards a new analysis system

The purpose of this thesis project is to expand the existing analysis system to better support the aims of the OME project. The specific goals of the new analysis system include the following:

1. It would support a more modular notion of analysis design.
2. It would need to be easy to add new modules to the analysis system, and easy to chain modules together to execute as a whole.
3. It would need to gather a sufficient amount of information as to provide a complete operational record of the analyses that were performed against a dataset.
4. The operational record would have to be easily accessed via simple queries.
5. It would need to be open-ended enough to allow future refinements to support such features as parallel and distributed computation.

CHAPTER 2

Design overview

2.1 Modular analysis design

The most visible change to the OME analysis system is the focus on modular analysis design. The existing version of OME provides the means for analysis routines to be executed against a set of images, but these routines are still monolithic in nature. The ability to make small changes to the way an analysis worked required changing the code of the routine, and in order to have all of the variations available to be executed, copies of the routine had to be made.

In the new system, the fundamentally modular design is much more obvious to the user. The routines known to the analysis system are intended to be small and relatively atomic; they do not, by themselves, necessarily calculate anything that is scientifically useful. Instead, they must be pieced together into analysis chains in order to perform a meaningful computation. This construction of analysis chains from modular pieces is fundamental to the new analysis system.

To support the notion of building analysis chains, there had to be a set of rules determining which modules could link to each other in a chain. To this end, I formalized a notion of semantic OME data types. Every piece of metadata in the OME database, whether provided by the user or an analysis routine, belongs to exactly one semantic data type. Modules are then pieced together using data-dependency links; the outputs of one module are fed in as the inputs to the next. The concept of semantic data types supports this well. It provides the necessary safeguards that prevent the wrong kind of data from being presented as input to an analysis module, and also imposes a qualitative relationship between the modules in an analysis chain.

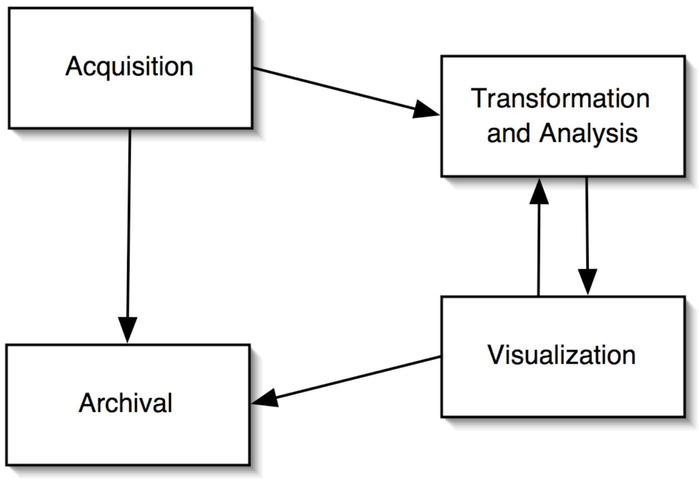

Figure 1 – Conceptual workflow

The ability to add modules to the system has been extended, as well. The idea of language independence has always been key to OME; the system has been designed so that every module that can access the OME database can interact seamlessly with each other. Instead of the conceptual workflow in which the acquisition, analysis, and visualization steps relate to each other as in Figure 1, OME uses the database as a communications medium (Figure 2), allowing each piece to be developed independently of each other, in any language, and by anyone. This language-independence extends to the routines used by the analysis system. The new analysis system, however, eases the use of these language-independent features by factoring out much of that logic. This is done through the use of analysis handlers, which bridge the gap between the analysis engine and the various analysis modules. In this way, new analysis modules can be written in the biologist's language of choice without having to develop their own database access layer.

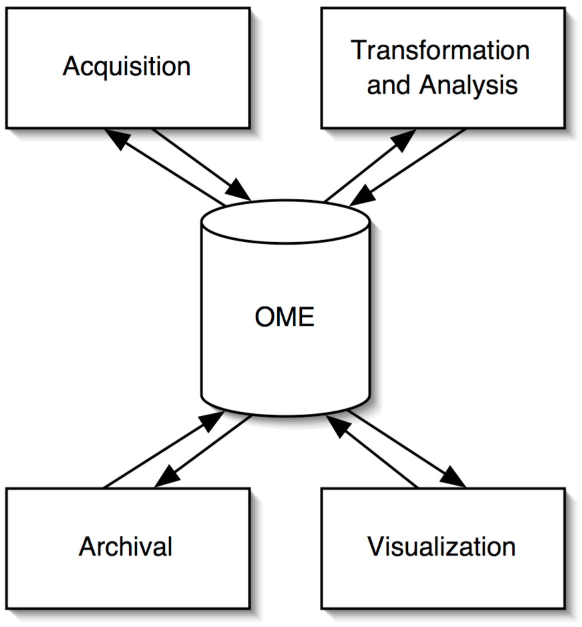

Figure 2 – Database as a communications medium

Analysis design in the context of a scientific experiment is usually an iterative process. The scientist starts with the data, and a preliminary understanding of which routines would provide useful insights. The results of these initial analyses can then lead to a refinement of the analysis routine itself, and even suggest completely new avenues of investigation. The use of analysis chains makes it much simpler to perform this kind of iterative exploration of the data and derived information.

The ability to reuse the results of an analysis are especially important in the context of iterative design, especially when each individual analysis can be computationally expensive. When a user changes a small portion of an analysis chain, the analysis engine should be smart enough to recalculate only those portions of the chain which were affected by the change. To do otherwise would incur unreasonable penalties in both storage and execution time. The OME analysis engine is able to take this result reuse one step further, and recognize the possibilities for reuse across different analysis chains as well.

Finally, while the logical structure imposed by a chain-based approach to analysis design is helpful by itself, the analysis engine must support the automated execution of an entire chain to be truly useful. This execution algorithm must take into account not only the underlying structure of the analysis chain, but also the requirements of each analysis module. Complicating this is the language-independence of the OME analysis modules; the execution algorithm ends up being responsible for ensuring that communication between the modules happens in a defined and consistent manner.

2.2 Operational record

In keeping with the importance of image metadata mentioned above, the OME analysis system strives to maintain essential information about each analysis result generated for an image. For each calculated result, the engine stores not only the data itself, but a full operational record of the derivation of the data. This helps to provide, at least in part, a more qualitative meaning for each result, rather than just its quantitative contents.

The largest part of this operational record takes the form of the data dependency graph used to generate each analysis result. Every result is produced as the output of some analysis module. This module might require inputs in order to complete its computation; in this case, the module's results are dependent on the outputs of its predecessor modules. These predecessor modules will also have dependencies based on their inputs; this process can be carried all the way back to modules which declare no inputs, forming a data dependency graph for the original result. Each analysis result has one of these data dependency trees; the tree encodes all of the pieces of information used to calculate the result.

When searching a large data dependency tree, it is often difficult to find and select a specific set of results. The SQL standard does not support querying tree-based structures; to find the appropriate information, the biologist must use non-standard query statements or a specialized tool. To support the ability to retrieve this information with simpler queries, the analysis system subdivides the trees into linear paths. In this way, the scientist can focus on a small portion of an analysis chain at a time, and can retrieve all of the results in that portion without having to include the data dependencies into the query. These paths are described in more detail in section 3.3 (page 29).

2.3 Support for future refinements

Most of the improvements to the analysis system were built on top of the analysis logic that was part of the initial version of OME. This proved to be invaluable to the research groups that were already using OME in their experiments. In creating the new analysis system, we made sure to design the architecture in such a way to allow future improvements to be made in this incremental fashion as well.

First, we hope to include support in the future for more generalized analysis chains. In the current system, an analysis chain must be a directed acyclic graph (DAG). While this restriction makes much of the code of the analysis engine simpler, it eliminates several classic design patterns as possibilities in an analysis chains. The most conspicuous of these is looping; while loops can easily be built into the logic of an individual analysis module, any loop in an analysis chain would violate the acyclic constraint of the system. Eliminating this constraint would greatly increase the expressive power of an analysis chain.

Second, the modular makeup of the analysis chains also lends itself well to a parallel-processing implementation of the execution algorithm. The analysis modules in a chain cleanly separate the logic of the overall analysis into pieces which could be delegated to different processors. The execution algorithm at the heart of the analysis system does not currently take advantage of any parallel computation abilities of the underlying computer. However, this behavior can be added later with relatively little effort.

The same argument applies to a distributed model of computation, as well. Instead of delegating each module off to a different processor, the modules could be delegated to different machines entirely. The behavior of the two models is basically the same, but each has its own benefits. In the distributed case, each module would have the entire resources of a computer workstation available; in the parallel case, many resources, most notably memory, would have to be shared. Of course, the distributed scheme is more complicated to implement, since the delegation routines would have to ensure that each machine has access to the image data and metadata. In the parallel case, this is provided by the operating system. Either way, the distributed case presents another interesting area of future research for the analysis system.

Closely related to the distributed computation model is an application services approach to analysis writing. In the distributed case, all of the modules exist locally, where "locally" is defined to include all of the machines eligible for executing a distributed module. The application services model would similarly execute modules on remote machines, but in this case, the modules themselves reside on another computer entirely, completely outside the realm of the local OME installation. In the case of a research group or company that writes analysis modules for public use, this would greatly ease the process of distributing code and updates. It also better allows intellectual property to be protected, by preventing the source code from having to be released. While the OME core is itself open-source, we have strived to include the ability to incorporate proprietary third-party code without violating licensing agreements or intellectual property. An application services model would easily support this.

Of the three models of computation, the application services approach is the closest to being implemented in the current version of the analysis system, since it would require no changes to the underlying execution algorithm or data structures. As described later in section 4.4, the engine uses analysis handlers to factor out some of the database communications logic. The initial purpose of this was to aid in using existing analysis routines without having to write a database access wrapper; the analysis handlers basically serve this function in a generalized way. However, they can also be used equivalently to support a calling an analysis routine remotely, using a Standardized Object Access Protocol (SOAP) method call or something similar.

CHAPTER 3

Database schema details

3.1 Data types and attributes

The fundamental piece of information on which analysis modules operate is called an *attribute*. Attributes can be used to store low-level information about an image, such as its dimensions or pixel intensity statistics, and can also be used to store more derivative, high-level information, such as the number of cells in an image, and the percentage of those cells that can be considered apoptotic.

Further, OME attributes are strongly typed; every attribute belongs to exactly one *data type*. These data types do not match the traditional, storage-based notion of computer data types. Rather, OME data types are semantic in nature. In a standard programming language, an analysis routine could be declared as outputting a list of integer 3-tuples. In OME, however, the corresponding analysis module would be declared to output a list of centroids. Since the notion of a centroid includes some notion of its storage requirements, no descriptive power is lost by using semantic data types.

Finally, attributes are not restricted to describing single images. Rather, they have a property called *granularity* that determines whether an attribute describes a single image, a subset of a single image, or a collection of images. For instance, the mean intensity of the pixels in an image is considered an image attribute. A module could also calculate the mean intensity across multiple images in a dataset; this would be considered a dataset attribute. It is important to note that these two attributes are considered to be of different semantic data types, even though they both describe mean pixel intensities.

In terms of the underlying database schema, the data type determines in which database table an attribute resides. Every data type has its own table in the database; every attribute of that data type is a row in the corresponding table. The primary keys used in the attribute tables are expected to remain unique across all data types; i.e., two attributes may not have the same ID, even if they have different types and reside in different database tables.

The selection of data types is not fixed; rather, it is fully expected to grow to incorporate new data types as outside modules are included into the analysis system. This means the analysis engine needs to know which data types are defined in the system at any given time. Therefore, a set of reflection tables (`DATATYPES`, Figure 24, page 66, and `DATATYPE_COLUMNS`, Figure 25, page 67) is included to capture this information. Each data type has a row in the `DATATYPES` table that specifies the database table for the data type, a short description, and the data type's granularity. Further, each column in the data type's table has a row in the `DATATYPE_COLUMNS` table. This row contains the name of the database column, a short description, and a reference property, similar in nature to the SQL `REFERENCES` clause.

In terms of the higher-level analysis design, the data types provide the mechanism for linking modules together into chains. An output of one module can only be linked to the input of another if the two have matching data types. The fact that OME data types are semantic in nature gives this requirement extra power; instead of merely requiring a superficial storage-based correspondence between connected modules in a chain, the analysis engine enforces a qualitative relationship between the data being passed between modules.

Merely stating that OME data types are semantic in nature leaves unanswered an important question. What (or who) actually defines what a semantic data type represents? The example given above is the centroid; at first glance, it seems relatively simple to define what a centroid is. However, to be scientifically useful, the definition of a centroid needs to be more than "center of mass", or "weighted average of the pixel coordinates." Equally important is the exact algorithm used to calculate the position of the centroid. As an example, the specification for the Java language is very mathematically precise when defining even simple operations such as multiplication, so as to guarantee exact reproducibility [6]. Similar precision is necessary in defining the meanings of each semantic data type used by OME, especially if analysis results are to be used in the context of scientific research.

However, providing this level of precision raises two problems. First, it would be cumbersome and inflexible to define each semantic data type this precisely, and even more so to allow for the appropriate level of precision in defining new data types not included in the base OME installation. Further, the Java operators mentioned above are defined precisely via an English prose description in a language specification. This information is not readily available in a representation that a computer can interpret and verify; indeed, such a representation does not even exist, nor is the verification of the specification a solvable problem. Thus, it is impossible to simultaneously provide this level of detail in describing OME semantic data types and allow for the verification of those descriptions and the extension of the set of available data types.

The second problem involves one of the reasons for incorporating a modular approach to analysis in the first place. One of the goals of OME's analysis system is to allow a scientist to explore how incremental changes to the analysis chains affect the results. These incremental changes not only include minor adjustments to various input parameters, but the ability to use different modules at various points in the chain to investigate different analysis algorithms. To extend the running example, a scientist might wish to see how different methods of finding a centroid, each of which gives different locations, yields different final results. By providing an extremely precise definition of "centroid," the analysis system would prohibit this kind of investigation.

This leaves us with a conundrum: we must be precise enough to provide useful semantic data types (a centroid is more than just a location), leave enough flexibility to allow meaningful variations in how a module calculates its results (a centroid can be calculated in more than one way), and still record enough detail about what actually happened to maintain a reasonable operational record of the analysis (in the end, we need to know which particular centroid algorithm was used). Our solution is to give each data type a general description (a centroid is the weighted average of the coordinates) that any algorithm must comply with. This provides a first-order definition which can be used to provide meaningful connections between analysis modules, and leaves open the possibility of using different algorithms to calculate the result, as long as that result falls into the broad category specified by the definition.

To maintain the operational record, the full description of a particular attribute must not only include the data type and values; for true completeness, it must provide the tree of analyses which were used to produce the result. This would specify, via the description of the analysis modules, which particular algorithm was used to produce the result, and further, would specify which algorithms were used to produce every attribute on the result's data dependency tree.

3.2 Analysis modules

The fundamental computation step in the analysis engine is the *analysis module*. Analysis modules are not meant to perform computations that are scientifically useful in and of themselves. Instead, modules are intended to perform a useful, atomic subset of a full analysis. The user can then piece together multiple modules to create and perform an actual analysis.

Each analysis module known to the system is defined by a row in the `PROGRAMS` table (Figure 29, page 68). This table specifies the name of the module as seen to the user and a short description, in addition to the location of the module's code. The modules can also be categorized, to allow the user to be presented with a more organized list of available modules. Lastly, a placeholder field is provided to modules to specify their own user interface for the collection of input parameters. This functionality is not currently implemented, but the analysis engine has hooks to allow this to be added later without affecting large portions of the code.

Each module also specifies its formal inputs and outputs, which are stored in the `FORMAL_INPUTS` (Figure 27, page 67) and `FORMAL_OUTPUTS` (Figure 28, page 68) tables. Each input and output has a name, a short description, and a data type. The data type is specified as a reference into the `DATATYPE_COLUMNS` table rather than the `DATATYPES` table; this is to allow different fields of an attribute to be populated by different analysis modules. Internally, all data links in an analysis chain must maintain this column-based granularity; however, a user interface can try to collapse the data links into groups based on the data type table to eliminate clutter.

When the analysis engine executes an analysis module, every formal input is guaranteed to be given a value. It is possible, however, for some of the inputs to be assigned the null value. It is up to the analysis module to check that the inputs that are presented have acceptable values, and to raise an error otherwise. The module, however, does not have to provide values for every output; the analysis engine will assume a null value for any output not provided.

3.3 Analysis chains

As mentioned above, analysis modules must be pieced together into *analysis chains* before they can be executed against a set of images. An analysis chain is defined as a directed acyclic graph (DAG), where the nodes of the graph are instances of particular analysis modules, and the edges of the graph are the data dependency links (or *data links*) between the modules. The data types of the inputs and outputs of the modules determine which data links are valid; an output of one module can be connected to the input of another if and only if they have equivalent data types.

Two nodes of the graph are said to be connected by a *module link* if there is any data link connecting them. (In other words, the module links specify the general connectedness of the nodes in the chain; the data links specify precisely which inputs and outputs are used in each connection.) The nodes in the chain that contain no incoming module links are *root nodes*, while the nodes that contain no outgoing module links are *leaf nodes*. The module links also define the *module paths* in an analysis chain. The module paths of a graph are all of the possible paths along module links from each root node to each leaf node.

The distinction between data links, module links, and module paths are illustrated below.

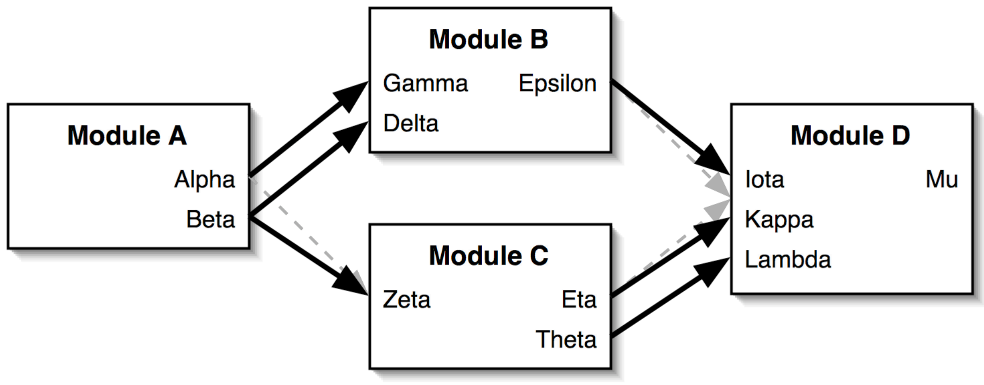

Figure 3 – Data links

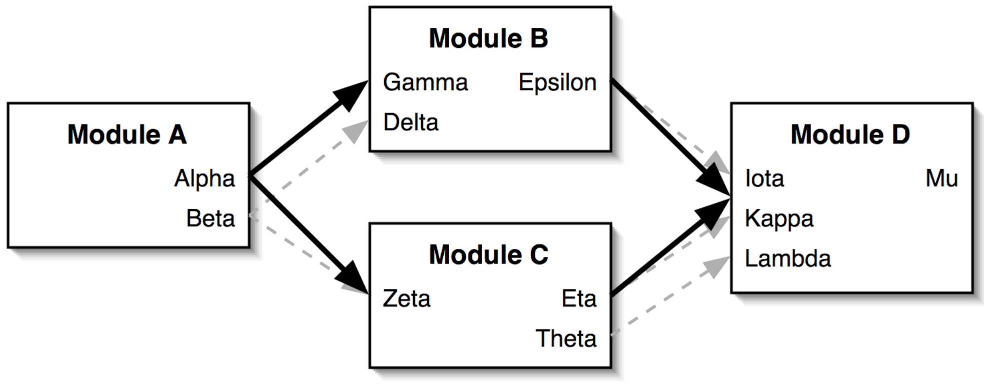

Figure 4 – Module links

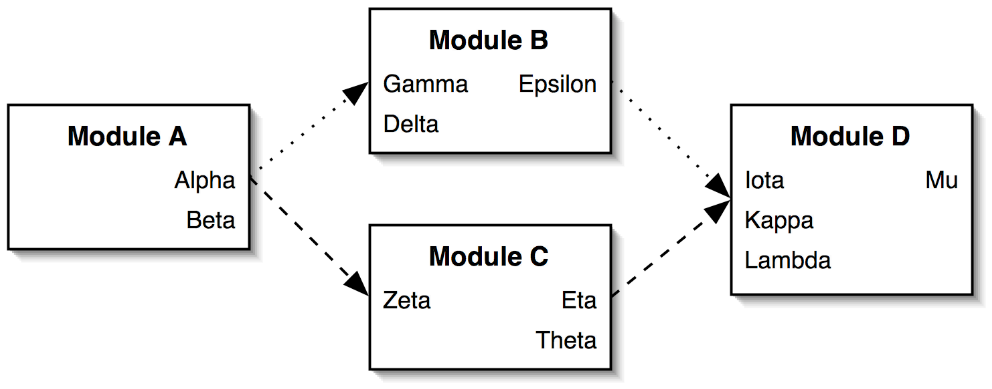

Figure 5 – Module paths

Since every input is guaranteed to have a value when presented to an analysis module, the data links in a graph divide the set of inputs into two disjoint subsets, known as the *bound inputs* and the *free inputs*. The bound inputs are those with incoming data links providing them with a value; the free inputs are all others. Upon executing an analysis chain, the user must provide a value for each free input in order for the guarantee to be met. The specification of an analysis chain can include default values for all of the free inputs to eliminate some of the burden from the user at execution time.

The structure of an analysis chain is encoded in the `ANALYSIS_VIEWS` (Figure 32, page 69), `ANALYSIS_VIEW_NODES` (Figure 31, page 69), and `ANALYSIS_VIEW_LINKS` (Figure 30, page 69) tables. The chain itself has an owner and a name, in addition to a `LOCKED` column. Once an analysis chain has been executed against a dataset, it is prevented from being modified, to allow a snapshot of the analysis execution to be reconstructed later. The nodes table contains a list of module nodes in the chain; each row contains a mapping between a node and the analysis module of which it is an instance. The links table contains a list of all of the data links in the chain; the list of module links can be easily derived from the data links. Each data link is defined in terms of not only which nodes it connects; but also specifically which output and input are connected. In order for a chain to be well-formed, the `FROM_NODE` and `FROM_OUTPUT` columns must link to the same analysis module, as must the `TO_NODE` and `TO_INPUT` columns.

3.4 Analysis executions

Each analysis chain can be executed against multiple datasets multiple times. This process is called an *analysis chain execution* (or *analysis execution* for short). In order to maintain a complete record of every analysis run, each of these executions is treated as a distinct object in the database. The `ANALYSIS_EXECUTIONS` table (Figure 36, page 71) contains one row for each of them.

Every time an analysis module is executed with a specific set of parameters, known as an *analysis module execution* (to distinguish it from an analysis chain execution), an entry is recorded in the `ANALYSES` table (Figure 35, page 70). This module execution is run either against a dataset or a specific image within the dataset; this dataset-dependence is property discussed in more detail in section 4.3. The attributes used as inputs and generated as outputs are known as the analysis module execution's *actual inputs and outputs*. The actual inputs and outputs are stored as a mapping between an analysis module execution, a formal input, and an attribute, and are kept in the `ACTUAL_INPUTS` (Figure 33, page 69) and `ACTUAL_OUTPUTS` (Figure 34, page 70) tables.

If, during a later execution, a module needs to be executed against an image, and an existing analysis has already calculated the appropriate value, it will be reused. In this way, each entry in the `ANALYSES` table can possibly belong to more than one analysis chain or analysis execution. Further, within an analysis chain, a node can belong to more than one module path. The `ANALYSIS_PATH_MAP` table (Figure 37, page 71) provides this three-way map between analysis executions, analysis module executions, and module paths.

CHAPTER 4

Analysis chain execution algorithm

At the heart of the analysis subsystem is the algorithm which executes an analysis chain against a dataset. This algorithm has several responsibilities; foremost is to ensure that the modules are executed in the correct order and that the results are collected and recorded in the OME database. In addition to recording the actual analysis results, it must also record all of the details of the underlying computations performed, to provide a operational record for future study. Finally, it must deal with error handling in a robust way. The algorithm is presented in its entirety in Figure 39 (page 93); the following sections describe its major components.

4.1 Finite state machine

The main body of the algorithm works using a finite state machine (FSM) that every node must pass through completely during the execution of an analysis chain. The FSM used in the algorithm is presented in Figure 6. By using an FSM in this manner, the execution algorithm can look at each node in a completely localized manner; the only constraints on whether a node can progress further through the FSM is the state of its immediate predecessors in the analysis chain.

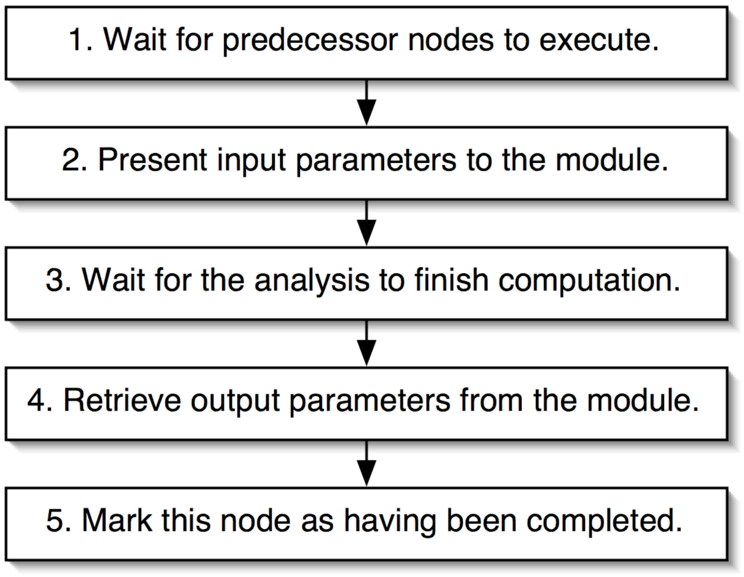
1. Wait for predecessor nodes to execute.
2. Present input parameters to the module.
3. Wait for the analysis to finish computation.
4. Retrieve output parameters from the module.
5. Mark this node as having been completed.

Figure 6 – Analysis algorithm finite state machine

Armed with this FSM, the execution algorithm is fairly straightforward. It uses a fixed-point algorithm that tries to move each node as far through the FSM as possible during each iteration. Once every node is in state 5, the execution of the chain is complete. The algorithm can also become "stuck," whereby not every node is in state 5, and yet, no node is able to progress further through the FSM. If this occurs, one of two possibilities exists: 1) A node generated an error during computation, or 2) the chain was malformed.

Of course, both of these error conditions can be checked before execution of the modules commences. This is desirable, especially in the case of a malformed chain, because it presents the user with a chance to fix small errors in the analysis chain before starting the potentially expensive computation steps involved in executing the chain. Execution does not actually proceed until there is a reasonable assurance on the part of the analysis subsystem that the computation can complete successfully.

The fixed-point algorithm can examine nodes in any order during each iteration. The nodes are still guaranteed to be executed in the proper order, even though the nodes are examined locally, since state 1 of the FSM inductively prevents a node from being executed before its predecessors have run. However, a future improvement could be made by ordering the nodes in a predetermined fashion, so we could decrease the number of times a node is tested and found unable to progress through the FSM. This would limit the amount of time the execution algorithm would spend in the fixed-point loop. The fact that the analysis chains are DAG's would make this ordering simple; it is merely a topological sort of the nodes in the chain. This ordering could be calculated quickly, and would provide a reduction in the running time of the fixed-point loop.

4.2 Calculating module paths

In addition to the logic described above for executing the analysis modules in the proper order, the execution algorithm needs to calculate the module paths of the analysis chain. Since the analysis chains are DAG's, this is a fairly simple calculation, which ends up dramatically reducing the complexity of the SQL queries needed to investigate certain kinds of relationships between analysis results.

To calculate the module paths, the algorithm starts by creating a single path for each root node in the chain. It then repeatedly takes each path it has found so far, looks at the node at the end of the path, and extends the path with that node's successors. If the tail node has no successors, then it is a complete module path for the analysis chain. If it has more than one successor, the path is duplicated so that there is one copy extended by each of the successors. When none of the paths can be extended any more, the algorithm has found all of the module paths. Once all of the module paths have been found, the algorithm creates rows in the `ANALYSIS_PATHS` table and stores them in an internal data structure for when the results are written to the database.

4.3 Reusing analysis results

One substantial optimization that the analysis engine provides is the ability to reuse analysis results when possible. This is especially useful when creating several analysis chains that differ only in the modules near the end of the chain; the running time of the root modules is amortized across all of the chains.

To determine whether the results of an analysis can be reused, we must define another property of the analysis modules, called *dataset-dependence*. If a module is dataset-dependent (or equivalently, *per-dataset*), then its outputs depend in some way on the dataset as a whole. Usually, these modules calculate some sort of statistic about the entire dataset, or use such a statistic as an input parameter. In this case, the results of the module can only be reused when the module is run on the exact same dataset; otherwise, the engine cannot guarantee that the calculations performed on an image would yield the same results.

On the other hand, in a module which is dataset-independent (or equivalently, *per-image*), the calculations performed on an image are completely independent of which other images are in the dataset. In this case, even if a module is run later on a different dataset, those images which were analyzed previously can still be skipped.

In a more formal sense, the dataset-dependence of a module can be defined inductively. Any module which declares an input or output with dataset granularity is initially defined to be dataset-dependent. The dataset-dependency of these modules can be determined at design time. Modules with only image and feature inputs and outputs, however, cannot have their dataset-dependence determined until runtime, since in this case, the property is determined by which other modules it is connected to. If any of a module's predecessors are per-dataset, then it, too is per-dataset. Otherwise, it is per-image.

Thus, dataset-dependence is a viral property; if a module is per-dataset, all of its successors in an analysis chain must be assumed to be per-dataset, as well. Luckily, the modules which are most likely to benefit from analysis reuse, those towards the root of an analysis chain, are exactly those which are most likely to retain their dataset-independence, and therefore be eligible for analysis reuse. Obviously, a per-image module is much better suited to analysis reuse.

In terms of the execution algorithm, we have to determine the dataset-dependence of each module in the chain before we can decide whether to reuse results. One solution is to recursively search the analysis chain, and all of the modules that each input in the chain depends on, searching for a per-dataset module. If one were found, then the module in question would also be per-dataset. If not, it would be per-image.

However, to reduce the amount of time spent searching through the data dependency tree, we can calculate this property inductively. To do so, we use the `DEPENDENCY` column in the `ANALYSES` table (which exists precisely for this purpose), and calculate the dataset-dependency of a module as one step in executing it. This means we must wait to calculate the dataset-dependency until the module is ready to be executed. In other words, we wait until all of its predecessor nodes have finished and are in state 5 of the FSM. This ensures that the dataset-dependencies of its immediate predecessors have also been calculated, and that we only need to check these immediate predecessors to determine the current module's dataset-dependency. This inductive solution greatly reduces the amount of tree searching required to determine a module's dataset dependency, at the cost of storing the dependency of each execution of a module.

Once we have determined the dataset-dependency of a module, we can search the OME database for an execution of the module that would be eligible for reuse. A module's results can be reused if the module was run on the same image (or dataset, in the case of per-dataset modules), with equivalent inputs, including the user-adjustable free inputs. We use shallow equality, rather than deep equality, to determine whether the inputs to two executions of a module are equivalent. In the case of the attribute tables in the OME database, using shallow equality means that in order for the inputs to the module to be considered equivalent, they must refer to the exact same row in the attribute table. Referring to distinct rows with identical contents is not sufficient.

It was deemed inappropriate to design a method of testing for deep equality, in which duplicate rows in the attribute table would also be considered equivalent. Shallow equality only allows reuse when the attributes in question were calculated by the same series of analysis modules, with identical inputs. Shallow equality better represents the true meaning of a semantic data type, which is not fully expressed without knowledge of the specific analyses that produced the data. Deep equality would blur this distinction, allowing attributes which had the same value to be considered equal, even if they were calculated in a wildly different manner.

With this notion of shallow equality, the test for reuse eligibility is fairly simple. The execution algorithm puts together an *input tag*, which is a string that succinctly encapsulates the necessary information about the module execution: the image or dataset on which the module was run, and the attributes used as inputs to the analysis. This tag is essentially a hash value of the analysis module and its inputs. Since the module's predecessors in the analysis chain have finished executing, the inputs are known. The algorithm calculates the current input tag based on this information, and then checks the database to see if an execution of the same module exists with the same input tag. If so, that execution's results are reused. This test must be performed at a different point in the execution algorithm depending on whether the module is per-dataset or per-image.

Note than in the case of analysis reuse, the module is not executed again, and therefore no new entry is created in the `ANALYSES` table. It is this fact that requires the `ANALYSIS_PATH_MAP` to be a three-way mapping. It not only encodes which module paths a module execution belongs to, but also encodes which module executions were used in each chain execution. The reuse of results makes this second mapping many-to-many. In the naïve approach, where each module is executed every time, the mapping would only be one-to-many.

4.4 Analysis handlers

The definition of analysis modules given in section 3.2 is only half-complete. Each analysis module has a contract to meet to correctly interact with the analysis system. The module is supposed to read inputs from the database, perform appropriate calculations, and write outputs back to the database. However, the analysis module itself is only responsible for meeting half of that contract. The modules will inevitably reuse the code to read inputs from and write outputs to the database. Further, this is exactly the code that we wish to prevent the casual scientist from having to write when creating new analysis modules.

To get around this, the analysis system uses the idea of an *analysis handler*. The handler factors out the logic of connecting to the database and meeting the analysis module contract. We provide the `OME::Analysis::Handler` module (Figure 40, page 95), which is an interface that defines the methods used by the analysis engine to specify and provide the attributes used as actual inputs to the module. For every way in which an analysis module can interact with the analysis engine, there is a separate handler implementing the interface. Since the number of possible analysis modules is much larger than the number of languages the modules are written in and calling conventions they will meet, this factorization is quite beneficial.

At this point, we have written three handlers. One is a Perl handler (Figure 42, page 101), which is a completely transparent handler that allows modules written against a Perl analysis interface (Figure 41, page 98) to connect to the analysis engine directly. The other two (Figure 43, page 105, and Figure 44, page 112) are command-line interface (CLI) handlers, which allow existing command-line tools to be incorporated as analysis modules. The command-line interfaces are currently coupled tightly to our test modules (described in chapter 5); work is in progress on an XML specification to generalize the translation of OME attributes in the database into text-based inputs and outputs, which can be placed in arbitrary positions on the command line or on standard in and standard out. We also plan to develop a Standardized Object Access Protocol (SOAP) handler, which would allow an application-services idiom to be applied to the routines used by the analysis system.

4.5 Code status

The next release of the OME system is still in development; as such, relatively minor changes can be made to the analysis system before the next release. As currently implemented, the analysis system described in this thesis forms a stable base on which to build these minor improvements and refinements. It supports the division of analysis routines into small modules, the execution of chains of these modules, and the reuse of previous analysis results. Our initial test cases of this base analysis system are presented in chapter 5. We are currently adding more analysis handlers to support more legacy analysis modules, and are developing several more advanced test cases.

CHAPTER 5
Test cases

To test the analysis engine, I incorporated three command-line utilities into OME as analysis modules. The first was a program known as `findSpots`, which was written by Ilya Goldberg as part of the previous OME version. This program was intended to be the main test-bed; it performs a useful calculation against imported images, and cleanly illustrates several important aspects of the engine: It segments images into features and calculates related feature attributes for each, and therefore demonstrates the various granularities of attributes available to modules. Further, it depends on previous analysis results in order to function, making it a good test case for the ordering and input propagation portions of the execution algorithm.

For the `findSpots` program to function, it needs intensity statistics for each XYZ stack in the five-dimensional image. This functionality is not included in the `findSpots` program itself; since these two operations are fundamentally different aspects of a larger analysis algorithm, they are implemented in two separate modules. This means that a module that calculates these statistics is needed for a chain involving `findSpots` to execute. To provide the statistics, I used a program called `OME_Image_XYZ_stats`, also written by Ilya Goldberg.

The final utility included in the testing suite was a modified version of the stack statistic routine. This new version, called `OME_Image_XY_stats`, was changed to calculate the intensity statistics on a per-plane basis.

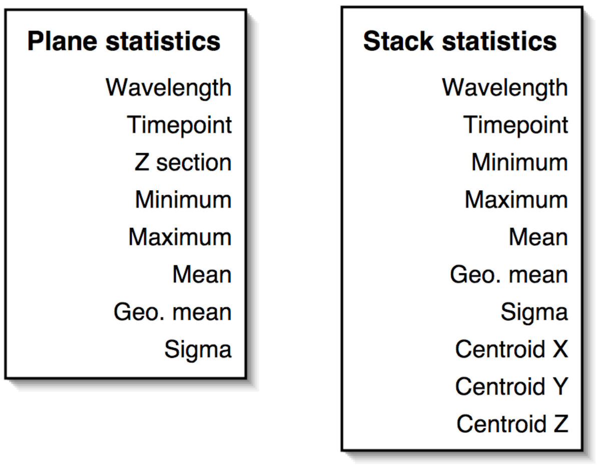

Figure 7 – Simple test analysis chain

These three modules were assembled into two analysis chains. The first, shown in Figure 7, is quite simple. There are no data dependency links, so executing it against a dataset should only take one iteration through the algorithm's fixed-point loop.

The second chain, shown in Figure 8, is more interesting, though still rather simple. The appropriate outputs of `OME_Image_XYZ_stats` are connected to the respective inputs of `findSpots`. Because of these data links, the FSM in the execution algorithm should not allow `findSpots` to execute until after `OME_Image_XYZ_stats` has completed.

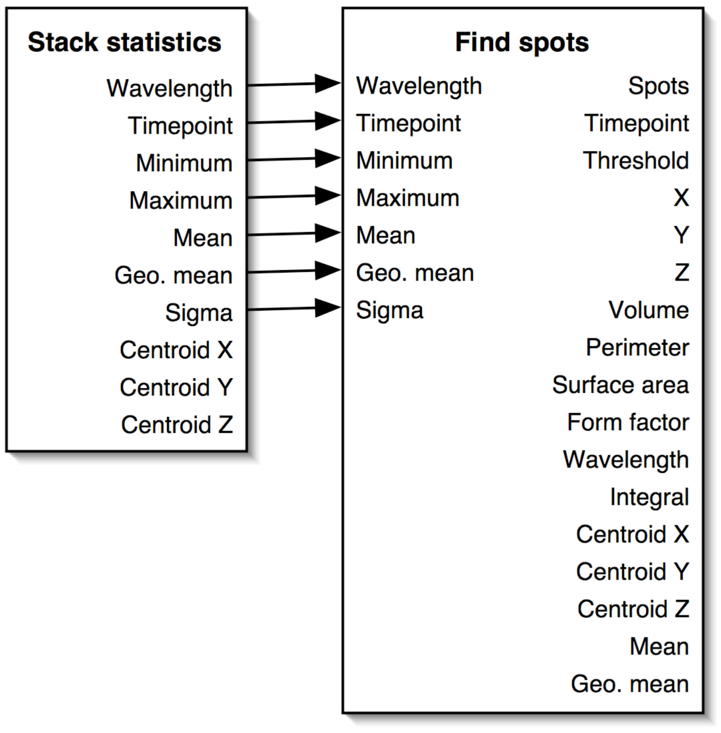

Figure 8 – Test analysis chain with links

5.1 Preparing the database

Before the analysis engine can execute these chains against a dataset, three things must be initialized in the database: First, the routines must be registered with the analysis engine by making the appropriate entries in the `PROGRAMS`, `FORMAL_INPUTS`, and `FORMAL_OUTPUTS` tables. Second, the chains must be created and placed into the `ANALYSIS_VIEWS`, `ANALYSIS_VIEW_NODES`, and `ANALYSIS_VIEW_LINKS` tables. Finally, at least one test image must be imported into OME (using an existing image import routine). These three steps are performed by a series of Perl scripts:

1. `OME::Tests::AnalysisEngine::CreateProgram` (Figure 45, page 123)

2. `OME::Tests::AnalysisEngine::CreateView` (Figure 46, page 128)

3. `OME::Tests::ImportTest` (Figure 47, page 131)

For testing, we used an image from Jason Swedlow's Cajal body experiment (see page 15) [5]. The output from these scripts is fairly straightforward:

```
OME Test Case - Create programs
-------------------------------
Please login to OME:
Username? dcreager
Password?

Great, you're in.

Finding datatypes...
  XYZ_IMAGE_INFO (2)
  XY_IMAGE_INFO (3)
  FEATURES (8)
  TIMEPOINT (11)
  THRESHOLD (14)
  LOCATION (9)
  EXTENT (13)
  SIGNAL (10)
Creating programs...
  Plane statistics (3)
    Wave (7)
    Time (8)
    Z (9)
    Min (10)
    Max (11)
    Mean (12)
    GeoMean (13)
    Sigma (14)
  Stack statistics (4)
    Wave (15)
    Time (16)
    Min (17)
    Max (18)
    Mean (19)
    GeoMean (20)
    Sigma (21)
    Centroid_x (22)
    Centroid_y (23)
    Centroid_z (24)
  Find spots (5)
    Wavelength (2)
    Timepoint (3)
    Minimum (4)
    Maximum (5)
    Mean (6)
    Geometric mean (7)
    Sigma (8)
    Timepoint (25)
    Threshold (26)
    X (27)
    Y (28)
    Z (29)
    Volume (30)
    Perimeter (31)
    Surface area (32)
    Form factor (33)
    Wavelength (34)
    Integral (35)
    Centroid X (36)
    Centroid Y (37)
    Centroid Z (38)
    Mean (39)
    Geometric Mean (40)
    Spots (41)
```

Figure 9 – `CreateProgram` output

```
OME Test Case - Create views
----------------------------
Please login to OME:
Username? dcreager
Password?

Great, you're in.

Finding programs...
Stack statistics (4)
Plane statistics (3)
Find spots (5)
Image import chain...
  Image import analyses (3)
    Node 1 Stack statistics (4)
    Node 2 Plane statistics (5)
Find spots chain...
  Find spots (4)
    Node 1 Stack statistics (6)
    Node 2 Find spots (7)
    Link [Node 1.Wavelength]->[Node 2.Wavelength]
    Link [Node 1.Timepoint]->[Node 2.Timepoint]
    Link [Node 1.Minimum]->[Node 2.Minimum]
    Link [Node 1.Maximum]->[Node 2.Maximum]
    Link [Node 1.Mean]->[Node 2.Mean]
    Link [Node 1.Geometric mean]->[Node 2.Geometric mean]
    Link [Node 1.Sigma]->[Node 2.Sigma]
```

Figure 10 – `CreateView` output

```
OME Test Case - Image Import
----------------------------
Please login to OME:
Username? dcreager
Password?

Great, you're in.

- Creating a new project...
Image is DV format
Times: 44, waves:1, zs: 20, rows: 256, cols: 256, sections: 880
output to /OME/repository/1-coilinSA5_4.ori
- Importing files into new project 'ImportTest2 project'... new image
id = 1
did import
done.
```

Figure 11 – `ImportTest` output

Two things are readily apparent from the analysis chains and the need for these scripts. First, any user interface used for creating and viewing analysis chains must perform the data link collapsing described in section 3.2. Many of the attribute tables will contain multiple columns; displaying each column as an input, output, or data link is more often than not redundant and cluttered. This is not done in Figure 8; it would be much cleaner (if less correct) if the links were collapsed.

Second, a tool for "importing" new analysis modules is needed. For simple test cases, a script such as `CreateProgram` is acceptable. Further, if the set of modules available to the user was to remain fixed after OME installation, then a script similar to `CreateProgram` could be part of the installation process. However, this is not the case, and requiring module designers to include specialized install scripts seems redundant when the way the information is entered into the database tables is extremely consistent. The XML specification described in section 4.4 would be a great aid to this tool. A generalized installation utility could easily be written around this specification, allowing the module designer to develop an XML document describing the module rather than a complete installation script.

5.2 Executing the no-link chain

Once the database is initialized properly, we can test the execution algorithm. This is done with the `OME::Tests::AnalysisEngine::ExecuteView` script (Figure 48, page 132), which executes a chain against a dataset, both of which are specified on the command line. Executing the simple chain (Figure 7) against the Cajal image, we see the following output:

```
OME Test Case - Execute view
----------------------------
Please login to OME:
Username? dcreager
Password?

Great, you're in.

Setup
  Creating ANALYSIS_EXECUTION table entry
  Plane statistics
    Loading module /OME/bin/OME_Image_XY_stats via handler
      OME::Analysis::CLIHandler
    Sorting input links by granularity
    Sorting outputs by granularity
      I Sigma
      I GeoMean
      I Mean
      I Max
      I Min
      I Z
      I Time
      I Wave
  Stack statistics
    Loading module /OME/bin/OME_Image_XYZ_stats via handler
      OME::Analysis::CLIHandler
    Sorting input links by granularity
    Sorting outputs by granularity
      I Centroid_z
      I Centroid_y
      I Centroid_x
      I Sigma
      I GeoMean
      I Mean
      I Max
      I Min
      I Time
      I Wave
  Building data paths
    Found root node 5
    Found root node 4
Round 1...
  Executing Plane statistics (I)
    startDataset
    Precalculate dataset
    Image coilinSA5_4
    Param I 1 d i f
    Creating ANALYSIS entry
    startImage
      /OME/bin/OME_Image_XY_stats
        Path=/OME/repository/1-coilinSA5_4.ori
        Dims=256,256,20,1,44,2
    Precalculate image
    Calculate feature
    Feature outputs
    Postcalculate image
    Image outputs
      Actual output Wave
      Actual output Time
      Actual output Z
      Actual output Min
      Actual output Max
      Actual output Mean
      Actual output GeoMean
      Actual output Sigma
    Postcalculate dataset
    Dataset outputs
    Marking state
  Executing Stack statistics (I)
    startDataset
    Precalculate dataset
    Image coilinSA5_4
    Param I 1 d i f
    Creating ANALYSIS entry
    startImage
      /OME/bin/OME_Image_XYZ_stats
        Path=/OME/repository/1-coilinSA5_4.ori
        Dims=256,256,20,1,44,2
    Precalculate image
    Calculate feature
    Feature outputs
    Postcalculate image
    Image outputs
      Actual output Wave
      Actual output Time
      Actual output Min
      Actual output Max
      Actual output Mean
      Actual output GeoMean
      Actual output Sigma
      Actual output Centroid_x
      Actual output Centroid_y
      Actual output Centroid_z
    Postcalculate dataset
    Dataset outputs
    Marking state
Round 2...
  Plane statistics already completed
  Stack statistics already completed

Timing:
  Total:  102 wallclock secs (23.91 usr +  2.72 sys = 26.63 CPU)
```

Figure 12 – Executing the no-link chain

As expected, both modules are executed during the first round of the fixed-point loop. Because of the column granularity of the data links and the size of the image (20 Z sections for each of 44 time points), the Stack Statistics program generates 440 entries in the ACTUAL_OUTPUTS table, while the Plane Statistics program generates 7,040 entries.

5.3 Attribute reuse #1: executing the no-link chain again

Our next test is to execute the exact same chain again. The attribute reuse portion of the algorithm should prevent the modules from being executed again; rather, the existing results will be reused. New entries will be made in the

ANALYSIS_EXECUTIONS and ANALYSIS_PATH_MAP tables, but the 7,480 entries in the ACTUAL_OUTPUTS table will not be duplicated.

```
OME Test Case - Execute view
----------------------------
Please login to OME:
Username? dcreager
Password?

Great, you're in.

Setup
  Creating ANALYSIS_EXECUTION table entry
  Plane statistics
    Loading module /OME/bin/OME_Image_XY_stats via handler
      OME::Analysis::CLIHandler
    Sorting input links by granularity
    Sorting outputs by granularity
      I Sigma
      I GeoMean
      I Mean
      I Max
      I Min
      I Z
      I Time
      I Wave
  Stack statistics
    Loading module /OME/bin/OME_Image_XYZ_stats via handler
      OME::Analysis::CLIHandler
    Sorting input links by granularity
    Sorting outputs by granularity
      I Centroid_z
      I Centroid_y
      I Centroid_x
      I Sigma
      I GeoMean
      I Mean
      I Max
      I Min
      I Time
      I Wave
  Building data paths
    Found root node 5
    Found root node 4
Round 1...
```

```
  Executing Plane statistics (I)
    startDataset
    Precalculate dataset
    Image coilinSA5_4
    Param I 1 d i f
      Found I 1 d i f
      Found reusable analysis
    Postcalculate dataset
    Dataset outputs
    Marking state
  Executing Stack statistics (I)
    startDataset
    Precalculate dataset
    Image coilinSA5_4
    Param I 1 d i f
      Found I 1 d i f
      Found reusable analysis
    Postcalculate dataset
    Dataset outputs
    Marking state
Round 2...
  Plane statistics already completed
  Stack statistics already completed

Timing:
  Total:  47 wallclock secs (27.00 usr +  1.98 sys = 28.98 CPU)
```

Figure 13 – Attribute reuse #1: executing the no-link chain again

The execution algorithm was able to find the existing results from both modules and reuse them. The execution time was only cut in half, mostly due to the amount of time necessary to load the previous results into the engine's internal state.

5.4 Attribute reuse #2: executing the linked chain

The third test case involves the `findSpots` chain. It is also a good example of analysis reuse across chains; the first node in this chain has already been calculated in the first test case. The execution algorithm should detect this and reuse the results.

```
OME Test Case - Execute view
----------------------------
Please login to OME:
Username? dcreager
Password?

Great, you're in.

Setup
  Creating ANALYSIS_EXECUTION table entry
  Find spots
    Loading module /OME/bin/findSpotsOME via handler
      OME::Analysis::FindSpotsHandler
    Sorting input links by granularity
      I Sigma
      I Geometric mean
      I Mean
      I Maximum
      I Minimum
      I Timepoint
      I Wavelength
    Sorting outputs by granularity
      I Spots
      F Geometric Mean
      F Mean
      F Centroid Z
      F Centroid Y
      F Centroid X
      F Integral
      F Wavelength
      F Form factor
      F Surface area
      F Perimeter
      F Volume
      F Z
      F Y
      F X
      F Threshold
      F Timepoint
  Stack statistics
    Loading module /OME/bin/OME_Image_XYZ_stats via handler
      OME::Analysis::CLIHandler
    Sorting input links by granularity
    Sorting outputs by granularity
      I Centroid_z
      I Centroid_y
      I Centroid_x
      I Sigma
      I GeoMean
      I Mean
      I Max
      I Min
      I Time
      I Wave
  Building data paths
    Found root node 6
    Extending 6 with 7
Round 1...
  Skipping Find spots
  Executing Stack statistics (I)
    startDataset
    Precalculate dataset
    Image coilinSA5_4
    Param I 1 d i f
      Found I 1 d i f
      Found reusable analysis
    Postcalculate dataset
    Dataset outputs
    Marking state
Round 2...
  Executing Find spots (I)
    startDataset
    Precalculate dataset
    Image coilinSA5_4
    Param I 1 d i 2(927 928 929 930 931 932 933 934 935 936 937 938
939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955
956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 ) 3(927
928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944
945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961
962 963 964 965 966 967 968 969 970 ) 4(927 928 929 930 931 932 933
934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950
951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967
968 969 970 ) 5(927 928 929 930 931 932 933 934 935 936 937 938 939
940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956
957 958 959 960 961 962 963 964 965 966 967 968 969 970 ) 6(927 928
929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945
946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962
963 964 965 966 967 968 969 970 ) 7(927 928 929 930 931 932 933 934
935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951
952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968
969 970 ) 8(927 928 929 930 931 932 933 934 935 936 937 938 939 940
941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957
958 959 960 961 962 963 964 965 966 967 968 969 970 ) f
```

```
    Creating ANALYSIS entry
    startImage
      /OME/bin/findSpotsOME /OME/repository/1-coilinSA5_4.ori 0
        gmean4.5s 10 -db -tt -th -c 0 -i 0 -m 0 -g 0 -ms 0 -gs 0 –mc
        -v -sa -per -ff
    Precalculate image
    Calculate feature
    Feature outputs
    Postcalculate image
    Image outputs
      Actual output Spots
    Postcalculate dataset
    Dataset outputs
    Marking state
  Stack statistics already completed
Round 3...
  Find spots already completed
  Stack statistics already completed

Timing:
  Total:  110 wallclock secs (32.80 usr  3.28 sys + 15.95 cusr  0.63
csys = 52.66 CPU)
```

Figure 14 – Attribute reuse #2: executing the linked chain

The `findSpots` program locates 448 spots in the Cajal image, and records 7,616 entries in the `ACTUAL_OUTPUTS` table (including the spots themselves).

5.5 Attribute reuse #3: executing the linked chain again

The final test is to execute the `findSpots` chain one more time, to verify that its results are reused, too.

```
OME Test Case - Execute view
----------------------------
Please login to OME:
Username? dcreager
Password?

Great, you're in.

Setup
  Creating ANALYSIS_EXECUTION table entry
  Find spots
```

```
    Loading module /OME/bin/findSpotsOME via handler
      OME::Analysis::FindSpotsHandler
    Sorting input links by granularity
      I Sigma
      I Geometric mean
      I Mean
      I Maximum
      I Minimum
      I Timepoint
      I Wavelength
    Sorting outputs by granularity
      I Spots
      F Geometric Mean
      F Mean
      F Centroid Z
      F Centroid Y
      F Centroid X
      F Integral
      F Wavelength
      F Form factor
      F Surface area
      F Perimeter
      F Volume
      F Z
      F Y
      F X
      F Threshold
      F Timepoint
  Stack statistics
    Loading module /OME/bin/OME_Image_XYZ_stats via handler
      OME::Analysis::CLIHandler
    Sorting input links by granularity
    Sorting outputs by granularity
      I Centroid_z
      I Centroid_y
      I Centroid_x
      I Sigma
      I GeoMean
      I Mean
      I Max
      I Min
      I Time
      I Wave
  Building data paths
    Found root node 6
    Extending 6 with 7
Round 1...
```

```
  Skipping Find spots
  Executing Stack statistics (I)
    startDataset
    Precalculate dataset
    Image coilinSA5_4
    Param I 1 d i f
      Found I 1 d i f
      Found reusable analysis
    Postcalculate dataset
    Dataset outputs
    Marking state
Round 2...
  Executing Find spots (I)
    startDataset
    Precalculate dataset
    Image coilinSA5_4
    Param I 1 d i 2(927 928 929 930 931 932 933 934 935 936 937 938
939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955
956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 ) 3(927
928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944
945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961
962 963 964 965 966 967 968 969 970 ) 4(927 928 929 930 931 932 933
934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950
951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967
968 969 970 ) 5(927 928 929 930 931 932 933 934 935 936 937 938 939
940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956
957 958 959 960 961 962 963 964 965 966 967 968 969 970 ) 6(927 928
929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945
946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962
963 964 965 966 967 968 969 970 ) 7(927 928 929 930 931 932 933 934
935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951
952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968
969 970 ) 8(927 928 929 930 931 932 933 934 935 936 937 938 939 940
941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957
958 959 960 961 962 963 964 965 966 967 968 969 970 ) f
      Found I 1 d i 2(927 928 929 930 931 932 933 934 935 936 937 938
939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955
956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 ) 3(927
928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944
945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961
962 963 964 965 966 967 968 969 970 ) 4(927 928 929 930 931 932 933
934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950
951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967
968 969 970 ) 5(927 928 929 930 931 932 933 934 935 936 937 938 939
940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956
957 958 959 960 961 962 963 964 965 966 967 968 969 970 ) 6(927 928
929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945
946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962
963 964 965 966 967 968 969 970 ) 7(927 928 929 930 931 932 933 934
935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951
952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968
969 970 ) 8(927 928 929 930 931 932 933 934 935 936 937 938 939 940
941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957
958 959 960 961 962 963 964 965 966 967 968 969 970 ) f
      Found reusable analysis
    Postcalculate dataset
    Dataset outputs
    Marking state
  Stack statistics already completed
Round 3...
  Find spots already completed
  Stack statistics already completed

Timing:
  Total:  68 wallclock secs (38.09 usr +  2.73 sys = 40.82 CPU)
```

Figure 15 – Attribute reuse #3: executing the linked chain again

The output from this test yields a good example of the input tag used to search for analyses to reuse. The line beginning with "`Param I 1 d i`" represents the input for the current analysis module, whereas the line beginning with "`Found I 1 d i`" represents the same input tag being found in the database, signaling a module whose results can be reused.

CHAPTER 6
Conclusion

The existing image acquisition tools for optical microscopy are extremely robust, useful, well-designed and well-tested applications. However, their lack of interoperability and extensibility often limits their usefulness. By defining several open standards with the cooperation of the industrial imaging community, the OME project hopes to provide the means for these programs to communicate with each other gracefully, allowing the scientist to use the unique advantages of each to its fullest potential.

The new analysis system represents a step forward in supporting these goals. The modular architecture allows the scientist to design analyses in a logical, incremental fashion. Using the database as the communications layer allows new modules to be developed easily in a variety of programming languages, and further, allows existing legacy routines to be easily incorporated with no change.

Most importantly, the OME analysis system deems paramount the importance of preserving the operational record of an analysis in addition to the results. Only with this record can the full meaning of each result be fully appreciated. The close association of the microscope images with the analysis results and their corresponding operational records ensures that nothing in this web of information slips away.

6.1 Future plans

The design of the analysis system leaves open the possibility for future extensions being added easily. Most of these anticipated extensions involve increasing the computational power of the execution algorithm by distributing the analysis workload. The two main possibilities along these lines are the distributed and parallel models of computation. Though the logistics of delegating the work differ between the two models, implementing either on top of the current analysis engine would proceed along the same lines. The delegation logic would be closely localized to the area of the execution algorithm which passes control to each analysis module; the rest of the algorithm and the underlying database schema would remain unchanged.

Further extensions can also be made to the underlying graph structure used by the analysis engine. Removing the acyclic constraint on analysis chains would greatly increase the expressive power of the analyses that the engine could handle. However, it would also require a substantial reworking of the execution algorithm itself, which makes many assumptions based on the DAG constraint.

Lastly, we hope to automate the process of including modules and their corresponding semantic data types into the OME database. We are currently developing an XML schema which will describe modules to be incorporated into the system, in addition to the semantic types on which they operate. We plan to develop a tool that will take these descriptions and extend the attribute tables in the OME database to include any new semantic types in the module definitions. Using this tool, it would be possible to "bootstrap" a large portion of the core OME database from the descriptions of the modules provided in the default OME toolbox.

References

[1] Abramowitz, M. *Fluorescence Microscopy: The Essentials*. Olympus America, Inc., New York, 1993.

[2] Herman, B. *Fluorescence Microscopy*. 2e, BIOS Scientific Publishers Ltd., Oxford, U.K., 1998.

[3] Rost, F. W. D. *Fluorescence Microscopy* (2 volumes). Cambridge University Press, New York, 1992.

[4] Sluder, G., and Wolf, D. E. (eds.). *Methods in Cell Biology, Vol. 56: Video Microscopy*. Academic Press, New York, 1998.

[5] Platani, M., Goldberg, I., Lamond, A. I., and Swedlow, J. R. (2002). "Cajal body dynamics and association with chromatin are ATP-dependent." *Nature Cell Biology* 4, 502-508.

[6] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2000). *The Java Language Specification.* http://java.sun.com/docs/books/jls/second_edition/-html/j.title.doc.html .

APPENDIX A
Source code - SQL

A.1 Preexisting OME tables

The tables in this section existed in the OME schema before the development of the new analysis engine. There are other tables in the schema than those below; only those relevant to the analysis engine and referenced in this document are included.

```
CREATE SEQUENCE DATASET_SEQ;
CREATE TABLE DATASETS (
  DATASET_ID   OID DEFAULT nextval('DATASET_SEQ') PRIMARY KEY,
  NAME         VARCHAR(256) NOT NULL,
  OWNER_ID     OID NOT NULL REFERENCES EXPERIMENTERS
               DEFERRABLE INITIALLY DEFERRED,
  GROUP_ID     OID REFERENCES GROUPS
               DEFERRABLE INITIALLY DEFERRED,
  DESCRIPTION  TEXT,
  LOCKED       BOOLEAN NOT NULL DEFAULT FALSE
);
```

Figure 16 – DATASETS table

```
CREATE TABLE IMAGE_DATASET_MAP (
  IMAGE_ID    OID NOT NULL REFERENCES IMAGES
              DEFERRABLE INITIALLY DEFERRED,
  DATASET_ID  OID NOT NULL REFERENCES DATASETS
              DEFERRABLE INITIALLY DEFERRED,
  PRIMARY KEY (IMAGE_ID, DATASET_ID)
);
```

Figure 17 – IMAGE_DATASET_MAP table

```
CREATE SEQUENCE IMAGE_SEQ;
CREATE TABLE IMAGES (
  IMAGE_ID          OID DEFAULT nextval('IMAGE_SEQ')
                    PRIMARY KEY,
  IMAGE_GUID        VARCHAR(256),
  NAME              VARCHAR(256) NOT NULL,
  FILE_SHA1         CHAR (40),
  DISPLAY_SETTINGS  OID REFERENCES
                    DEFERRABLE INITIALLY DEFERRED,
  DESCRIPTION       TEXT,
  INSTRUMENT_ID     OID REFERENCES INSTRUMENTS
                    DEFERRABLE INITIALLY DEFERRED,
  LENS_ID           VARCHAR(30),
  EXPERIMENTER_ID   OID NOT NULL REFERENCES EXPERIMENTERS
                    DEFERRABLE INITIALLY DEFERRED,
  GROUP_ID          OID REFERENCES GROUPS,
  CREATED           TIMESTAMP NOT NULL,
  INSERTED          TIMESTAMP NOT NULL,
  IMAGE_TYPE        OID,
  REPOSITORY_ID     OID NOT NULL REFERENCES REPOSITORIES
                    DEFERRABLE INITIALLY DEFERRED,
  PATH              VARCHAR(256) NOT NULL
);
```

Figure 18 – `IMAGES` table

```
CREATE SEQUENCE REPOSITORY_SEQ;
CREATE TABLE REPOSITORIES (
  REPOSITORY_ID  OID DEFAULT nextval('REPOSITORY_SEQ')
                 PRIMARY KEY,
  PATH           VARCHAR(256) NOT NULL UNIQUE
);
```

Figure 19 – `REPOSITORIES` table

A.2 Preexisting OME attribute tables

The tables in this section existed in the OME schema before the development of the new analysis engine, but were modified to conform to the new attribute table pattern.

```
CREATE TABLE IMAGE_DIMENSIONS (
  ATTRIBUTE_ID    OID DEFAULT nextval('ATTRIBUTE_SEQ') PRIMARY KEY,
  IMAGE_ID        OID NOT NULL REFERENCES IMAGES
                  DEFERRABLE INITIALLY DEFERRED,
  SIZE_X          INTEGER,
  SIZE_Y          INTEGER,
  SIZE_Z          INTEGER,
  NUM_WAVES       INTEGER,
  NUM_TIMES       INTEGER,
  BITS_PER_PIXEL  SMALLINT,
  PIXEL_SIZE_X    FLOAT4,
  PIXEL_SIZE_Y    FLOAT4,
  PIXEL_SIZE_Z    FLOAT4,
  WAVE_INCREMENT  FLOAT4,
  TIME_INCREMENT  FLOAT4
);
```

Figure 20 – `IMAGE_DIMENSIONS` table

```
CREATE TABLE IMAGE_WAVELENGTHS (
  ATTRIBUTE_ID   OID DEFAULT NEXTVAL('ATTRIBUTE_SEQ') PRIMARY KEY,
  IMAGE_ID       OID REFERENCES IMAGES
                 DEFERRABLE INITIALLY DEFERRED,
  WAVENUMBER     INTEGER,
  EX_WAVELENGTH  INTEGER,
  EM_WAVELENGTH  INTEGER,
  ND_FILTER      FLOAT,
  FLUOR          VARCHAR(32)
);
```

Figure 21 – `IMAGE_WAVELENGTHS` table

```
CREATE TABLE XY_IMAGE_INFO (
  ATTRIBUTE_ID  OID DEFAULT NEXTVAL('ATTRIBUTE_SEQ') PRIMARY KEY,
  IMAGE_ID      OID NOT NULL REFERENCES IMAGES
                DEFERRABLE INITIALLY DEFERRED,
  WAVENUMBER    INTEGER,
  TIMEPOINT     INTEGER,
  ZSECTION      INTEGER,
  DELTATIME     FLOAT,
  EXPTIME       FLOAT,
  STAGE_X       FLOAT,
  STAGE_Y       FLOAT,
  STAGE_Z       FLOAT,
  MIN           INTEGER,
  MAX           INTEGER,
  MEAN          FLOAT4,
  GEOMEAN       FLOAT4,
  SIGMA         FLOAT4
);
```

Figure 22 – XY_IMAGE_INFO table

```
CREATE TABLE XYZ_IMAGE_INFO (
  ATTRIBUTE_ID  OID DEFAULT nextval('ATTRIBUTE_SEQ') PRIMARY KEY,
  IMAGE_ID      OID NOT NULL REFERENCES IMAGES
                DEFERRABLE INITIALLY DEFERRED,
  WAVENUMBER    INTEGER,
  TIMEPOINT     INTEGER,
  DELTATIME     FLOAT,
  MIN           INTEGER,
  MAX           INTEGER,
  MEAN          FLOAT4,
  GEOMEAN       FLOAT4,
  SIGMA         FLOAT4,
  CENTROID_X    FLOAT4,
  CENTROID_Y    FLOAT4,
  CENTROID_Z    FLOAT4
);
```

Figure 23 – XYZ_IMAGE_INFO table

A.3 Data types and attributes

```
CREATE SEQUENCE DATATYPE_SEQ;
CREATE TABLE DATATYPES (
  DATATYPE_ID     OID DEFAULT nextval('DATATYPE_SEQ') PRIMARY KEY,
  ATTRIBUTE_TYPE  CHAR(1) NOT NULL
                  CHECK (ATTRIBUTE_TYPE IN ('D','I','F')),
  TABLE_NAME      VARCHAR(64) NOT NULL,
  DESCRIPTION     TEXT
);
```

Figure 24 – DATATYPES table

```
CREATE SEQUENCE DATATYPE_COLUMN_SEQ;
CREATE TABLE DATATYPE_COLUMNS (
  DATATYPE_COLUMN_ID  OID DEFAULT nextval('DATATYPE_COLUMN_SEQ')
                      PRIMARY KEY,
  DATATYPE_ID         OID NOT NULL REFERENCES DATATYPES
                      DEFERRABLE INITIALLY DEFERRED,
  COLUMN_NAME         VARCHAR(256) NOT NULL,
  REFERENCE_TYPE      VARCHAR(256)
);
```

Figure 25 – DATATYPE_COLUMNS table

```
CREATE TABLE FEATURES (
  ATTRIBUTE_ID  OID DEFAULT NEXTVAL('ATTRIBUTE_SEQ') PRIMARY KEY,
  IMAGE_ID      OID REFERENCES IMAGES
                DEFERRABLE INITIALLY DEFERRED
);
```

Figure 26 – FEATURES table

A.4 Analysis modules

```
CREATE SEQUENCE FORMAL_INPUT_SEQ;
CREATE TABLE FORMAL_INPUTS (
  FORMAL_INPUT_ID  OID DEFAULT nextval('FORMAL_INPUT_SEQ') PRIMARY
KEY,
  PROGRAM_ID       OID NOT NULL REFERENCES PROGRAMS DEFERRABLE
INITIALLY DEFERRED,
  NAME             VARCHAR(64) NOT NULL,
  DESCRIPTION      TEXT,
  COLUMN_TYPE      OID NOT NULL REFERENCES DATATYPE_COLUMNS DEFERRABLE
INITIALLY DEFERRED,
  LOOKUP_TABLE_ID  OID REFERENCES LOOKUP_TABLES,
  UNIQUE (PROGRAM_ID,NAME)
);
```

Figure 27 – FORMAL_INPUTS table

```
CREATE SEQUENCE FORMAL_OUTPUT_SEQ;
CREATE TABLE FORMAL_OUTPUTS (
  FORMAL_OUTPUT_ID  OID DEFAULT nextval('FORMAL_OUTPUT_SEQ') PRIMARY
KEY,
  PROGRAM_ID        OID NOT NULL REFERENCES PROGRAMS DEFERRABLE
INITIALLY DEFERRED,
  NAME              VARCHAR(64) NOT NULL,
  DESCRIPTION       TEXT,
  COLUMN_TYPE       OID NOT NULL REFERENCES DATATYPE_COLUMNS
DEFERRABLE INITIALLY DEFERRED,
  UNIQUE (PROGRAM_ID,NAME)
);
```

Figure 28 – FORMAL_OUTPUTS table

```
CREATE SEQUENCE PROGRAM_SEQ;
CREATE TABLE PROGRAMS (
  PROGRAM_ID     OID DEFAULT NEXTVAL('PROGRAM_SEQ') PRIMARY KEY,
  PROGRAM_NAME   VARCHAR(64) NOT NULL,
  DESCRIPTION    TEXT,
  LOCATION       VARCHAR(128) NOT NULL,
  MODULE_TYPE    VARCHAR(128) NOT NULL,
  CATEGORY       VARCHAR(32) NOT NULL,
  VISUAL_DESIGN  TEXT
);
```

Figure 29 – PROGRAMS table

A.5 Analysis chains

```
CREATE SEQUENCE ANALYSIS_VIEW_LINKS_SEQ;
CREATE TABLE ANALYSIS_VIEW_LINKS (
  ANALYSIS_VIEW_LINK_ID  OID DEFAULT
                         nextval('ANALYSIS_VIEW_LINKS_SEQ')
                         PRIMARY KEY,
  ANALYSIS_VIEW_ID       OID NOT NULL REFERENCES ANALYSIS_VIEWS
                         DEFERRABLE INITIALLY DEFERRED,
  FROM_NODE              OID NOT NULL REFERENCES ANALYSIS_VIEW_NODES
                         DEFERRABLE INITIALLY DEFERRED,
  FROM_OUTPUT            OID NOT NULL REFERENCES FORMAL_OUTPUTS
                         DEFERRABLE INITIALLY DEFERRED,
  TO_NODE                OID NOT NULL REFERENCES ANALYSIS_VIEW_NODES
                         DEFERRABLE INITIALLY DEFERRED,
  TO_INPUT               OID NOT NULL REFERENCES FORMAL_INPUTS
```

```
                         DEFERRABLE INITIALLY DEFERRED
);
```

Figure 30 – ANALYSIS_VIEW_LINKS table

```
CREATE SEQUENCE ANALYSIS_VIEW_NODES_SEQ;
CREATE TABLE ANALYSIS_VIEW_NODES (
  ANALYSIS_VIEW_NODE_ID  OID
                         DEFAULT nextval('ANALYSIS_VIEW_NODES_SEQ')
                         PRIMARY KEY,
  ANALYSIS_VIEW_ID       OID NOT NULL REFERENCES ANALYSIS_VIEWS
                         DEFERRABLE INITIALLY DEFERRED,
  PROGRAM_ID             OID NOT NULL REFERENCES PROGRAMS
                         DEFERRABLE INITIALLY DEFERRED
);
```

Figure 31 – ANALYSIS_VIEW_NODES table

```
CREATE SEQUENCE ANALYSIS_VIEW_SEQ;
CREATE TABLE ANALYSIS_VIEWS (
  ANALYSIS_VIEW_ID  OID DEFAULT nextval('ANALYSIS_VIEW_SEQ')
                    PRIMARY KEY,
  OWNER             OID NOT NULL REFERENCES EXPERIMENTERS
                    DEFERRABLE INITIALLY DEFERRED,
  NAME              VARCHAR(64) NOT NULL,
  DESCRIPTION       TEXT
);
```

Figure 32 – ANALYSIS_VIEWS table

A.6 Analysis executions

```
CREATE SEQUENCE ACTUAL_INPUT_SEQ;
CREATE TABLE ACTUAL_INPUTS (
  ACTUAL_INPUT_ID  OID DEFAULT nextval('ACTUAL_INPUT_SEQ')
                   PRIMARY KEY,
  ANALYSIS_ID      OID NOT NULL REFERENCES ANALYSES
                   DEFERRABLE INITIALLY DEFERRED,
  FORMAL_INPUT_ID  OID NOT NULL REFERENCES FORMAL_INPUTS
                   DEFERRABLE INITIALLY DEFERRED,
  ATTRIBUTE_ID     OID NOT NULL
);
```

Figure 33 – ACTUAL_INPUTS table

```
CREATE SEQUENCE ACTUAL_OUTPUT_SEQ;
CREATE TABLE ACTUAL_OUTPUTS (
  ACTUAL_OUTPUT_ID  OID DEFAULT nextval('ACTUAL_OUTPUT_SEQ')
                    PRIMARY KEY,
  ANALYSIS_ID       OID NOT NULL REFERENCES ANALYSES
                    DEFERRABLE INITIALLY DEFERRED,
  FORMAL_OUTPUT_ID  OID NOT NULL REFERENCES FORMAL_OUTPUTS
                    DEFERRABLE INITIALLY DEFERRED,
  ATTRIBUTE_ID      OID NOT NULL
);
```

Figure 34 – ACTUAL_OUTPUTS table

```
CREATE SEQUENCE ANALYSIS_SEQ;
CREATE TABLE ANALYSES (
  ANALYSIS_ID  OID DEFAULT nextval ('ANALYSIS_SEQ') PRIMARY KEY,
  DEPENDENCE   CHAR(1) NOT NULL
               CHECK (DEPENDENCE IN ('D','I')),
  DATASET_ID   OID REFERENCES DATASETS
               DEFERRABLE INITIALLY DEFERRED,
  IMAGE_ID     OID REFERENCES IMAGES
               DEFERRABLE INITIALLY DEFERRED,
  TIMESTAMP    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  STATUS       VARCHAR(16),
  CHECK
    (((DEPENDENCE = 'D') AND
      (DATASET_ID IS NOT NULL) AND (IMAGE_ID IS NULL)) OR
     ((DEPENDENCE = 'I') AND
      (DATASET_ID IS NULL) AND (IMAGE_ID IS NOT NULL)))
);
```

Figure 35 – ANALYSES table

```
CREATE SEQUENCE ANALYSIS_EXECUTION_SEQ;
CREATE TABLE ANALYSIS_EXECUTIONS (
  ANALYSIS_EXECUTION_ID  OID
                         DEFAULT nextval('ANALYSIS_EXECUTION_SEQ')
                         PRIMARY KEY,
  ANALYSIS_VIEW_ID       OID NOT NULL REFERENCES ANALYSIS_VIEWS
                         DEFERRABLE INITIALLY DEFERRED,
  DATASET_ID             OID NOT NULL REFERENCES DATASETS
                         DEFERRABLE INITIALLY DEFERRED,
  EXPERIMENTER_ID        OID NOT NULL REFERENCES EXPERIMENTERS
                         DEFERRABLE INITIALLY DEFERRED,
  TIMESTAMP                TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Figure 36 – ANALYSIS_EXECUTIONS table

```
CREATE TABLE ANALYSIS_PATH_MAP (
  PATH_ID                  OID NOT NULL,
  PATH_ORDER               INTEGER NOT NULL,
  ANALYSIS_ID              OID NOT NULL REFERENCES ANALYSES
                           DEFERRABLE INITIALLY DEFERRED,
  ANALYSIS_EXECUTION_ID    OID NOT NULL REFERENCES ANALYSIS_EXECUTIONS
                           DEFERRABLE INITIALLY DEFERRED,
  PRIMARY KEY (PATH_ID, ANALYSIS_ID, ANALYSIS_EXECUTION_ID)
);
```

Figure 37 – ANALYSIS_PATH_MAP table

```
CREATE SEQUENCE ANALYSIS_PATH_SEQ;
CREATE TABLE ANALYSIS_PATHS (
  PATH_ID      OID DEFAULT nextval('ANALYSIS_PATH_SEQ') PRIMARY KEY,
  PATH_LENGTH  INTEGER NOT NULL
);
```

Figure 38 – ANALYSIS_PATHS table

APPENDIX B
Source code - Perl

B.1 Analysis executions

```
package OME::Tasks::AnalysisEngine;

use strict;
our $VERSION = '1.0';

use OME::Factory;
use OME::DBObject;
use OME::Dataset;
use OME::Image;
use OME::Program;
use OME::Analysis;
use OME::AnalysisView;
use OME::AnalysisPath;
use OME::AnalysisExecution;

# For now assume the module type is the Perl class of the
# module handler.

sub findModuleHandler {
  return shift;
}

{

  # We'll need several shared variables for these internal
  # functions.  They are defined here.

  # The user's database session.
  my $session;

  # The database factory used to create new database objects and to
  # find existing ones.
  my $factory;

  # The analysis view being executed.
  my $analysis_view;

  # The hash of the user-specified input parameters.
  my $input_parameters;

  # The dataset the chain is being executed against.
  my $dataset;

  # A list of nodes in the analysis view.
  my @nodes;

  # A hash of nodes keyed by node ID.
  my %nodes;

  # The instantiated program modules for each analyis node.
  my %node_modules;

  # The current state of each node.
  use constant INPUT_STATE    => 1;
  use constant FINISHED_STATE => 2;
  my %node_states;

  # The input and output links for each node.
  # $input_links{$nodeID}->{$granularity} = $analysis_link
  # $output_links{$nodeID}->{$granularity} = $analysis_link
  my %input_links;
  my %output_links;

  # The ANALYSIS_EXECUTION entry for this chain execution.
  my $analysis_execution;

  # The dataset-dependence of each node.
  # $dependence{$nodeID} = [D,I]
  my %dependence;

  # The ANALYSES for each node.
  # $perdataset_analyses{$nodeID} = $analysis
  # $perimage_analyses{$nodeID}->{$imageID} = $analysis
  my %perdataset_analysis;
  my %perimage_analysis;

  # The outputs generated by each node
  # $dataset_outputs{$nodeID}->
  #   {$formal_outputID} = $attribute
  # $image_outputs{$nodeID}->{$formal_outputID}->
  #   {$imageID} = $attribute
  # $feature_outputs{$nodeID}->{$formal_outputID}->
  #   {$imageID}->{$featureID} = $attribute
  my (%dataset_outputs, %image_outputs, %feature_outputs);

  # Whether or not we need another round in the fixed-point loop.
  my $continue;

  # Which of those rounds we are in.
  my $round;

  # The node which was most recently executed.
  my $last_node;

  # The following variables are only valid within the per-node loop.
  # They refer to the module currently being examined/executed.
  my ($curr_node,           $curr_nodeID);
  my ($curr_module,         $curr_inputs, $curr_outputs);
  my (@curr_dataset_inputs, @curr_image_inputs, @curr_feature_inputs);
  my (@curr_dataset_outputs, @curr_image_outputs,
    @curr_feature_outputs);
  my ($curr_image, $curr_imageID);

  # The list of data paths found.
  my @data_paths;

  # The data paths to which each node belongs.
  my %data_paths;

  # This routine prepares the all of the internal variables for each
  # node in the chain.  It loads in the appropriate module handler,
  # and initializes it with the module's location, and sets up the
  # [dataset,image,feature]_[inputs,outputs] hashes with the input
  # and output links of the module.  Currently, all outputs are
  # added to the hashes, regardless of whether or not they are
  # linked to anything.  However, only those inputs which are
  # connected are pushed into their hashes.  *** This is where I
  # will add support for user parameters; the inputs without links
  # will look for their values in the $input_attributes parameter,
  # and push those values into the hash accordingly.
  sub __initializeNode {
    my $program     = $curr_node->program();
    my $module_type = $program->module_type();
    my $location    = $program->location();
    $nodes{$curr_nodeID} = $curr_node;

    print STDERR "  " . $program->program_name() . "\n";

    print STDERR
      "    Loading module $location via handler $module_type\n";
    my $handler = findModuleHandler($module_type);
    eval "require $handler";
    my $module = $handler->new($location, $factory, $program);
    $node_modules{$curr_nodeID} = $module;

    print STDERR "    Sorting input links by granularity\n";

    $input_links{$curr_nodeID}->{D} = [];
    $input_links{$curr_nodeID}->{I} = [];
    $input_links{$curr_nodeID}->{F} = [];

    # this pushes only linked inputs
    my @inputs = $curr_node->input_links();
    foreach my $input (@inputs) {
      my $attribute_type =
        $input->to_input()->column_type()->datatype()
        ->attribute_type();
      push @{$input_links{$curr_nodeID}->{$attribute_type}}, $input;
      print STDERR "      $attribute_type "
        . $input->to_input()->name() . "\n";
    }

    print STDERR "    Sorting outputs by granularity\n";

    $output_links{$curr_nodeID}->{D} = [];
    $output_links{$curr_nodeID}->{I} = [];
    $output_links{$curr_nodeID}->{F} = [];

    my @outputs = $program->outputs();
    foreach my $output (@outputs) {
      my $attribute_type =
        $output->column_type()->datatype()->attribute_type();
      push @{$output_links{$curr_nodeID}->{$attribute_type}}, $output;
      print STDERR "      $attribute_type " . $output->name() . "\n";
    }

    $node_states{$curr_nodeID} = INPUT_STATE;
  }

  # Returns true if a node has no predecessors.
  sub __rootNode {
    my ($node) = @_;
    foreach my $granularity ('D', 'I', 'F') {
      my $inputs = $input_links{$node->id()}->{$granularity};
      return 0 if (scalar(@$inputs) > 0);
    }
    return 1;
  }

  # Returns a list of successor nodes to a node.
  sub __successors {
    my ($nodeID) = @_;
    my (@succ, %succ);
    my $outputs = $nodes{$nodeID}->output_links();
    while (my $output = $outputs->next()) {
      my $succ_nodeID = $output->to_node()->id();
      if ($succ{$succ_nodeID} != 1) {
        push @succ, $succ_nodeID;
        $succ{$succ_nodeID} = 1;
      }
    }
    return \@succ;
  }

  sub __buildDataPaths {
    print STDERR "  Building data paths\n";

    # A data path is represented by a list of node ID's, starting
    # with a root node and ending with a leaf node.

    # First, we create paths for each root node in the chain
    foreach my $node (@nodes) {
      if (__rootNode($node)) {
        print STDERR "    Found root node " . $node->id() . "\n";
        my $path = [$node->id()];
        push @data_paths, $path;
      }
    }

    # Then, we iteratively extend each path until it reaches a
    # leaf node.  If at any point, it branches, we create
    # duplicates so that there is one path per branch-point.
    my $continue = 1;
    while ($continue) {
      $continue = 0;
      my @new_paths;
      while (my $data_path = shift (@data_paths)) {
        my $end_nodeID     = $data_path->[$#$data_path];
        my $successors     = __successors($end_nodeID);
        my $num_successors = scalar(@$successors);

        if ($num_successors == 0) {
          push @new_paths, $data_path;
        } elsif ($num_successors == 1) {
          print STDERR "    Extending "
            . join (':', @$data_path)
            . " with "
            . $successors->[0] . "\n";
          push @$data_path, $successors->[0];
          push @new_paths,  $data_path;
          $continue = 1;
        } else {
          foreach my $successor (@$successors) {

            # make a copy
            my $new_path = [@$data_path];
            print STDERR "    Extending "
              . join (':', @$new_path)
              . " with "
              . $successor . "\n";
            push @$new_path, $successor;
            push @new_paths, $new_path;
          }
          $continue = 1;
        }
      }

      @data_paths = @new_paths;
    }

    foreach my $data_path_list (@data_paths) {
      my $data_path =
        $factory->newObject("OME::AnalysisPath",
        {path_length => scalar(@$data_path_list)});
      my $data_pathID = $data_path->id();

      my $order = 0;
      foreach my $nodeID (@$data_path_list) {
        my $path_entry = {
          path       => $data_path,
          path_order => $order
        };
        push @{$data_paths{$nodeID}}, $path_entry;
        $order++;
      }
    }
  }

  # Returns the correct ANALYSIS entry for the current node.  Takes
  # into account the dataset-dependence of the node; if it is a
  # per-image node, it finds the ANALYSIS entry for the current
  # image.
  sub __getAnalysis {
    my ($nodeID) = @_;

    if ($dependence{$nodeID} eq 'D') {
      return $perdataset_analysis{$nodeID};
    } else {
      return $perimage_analysis{$nodeID}->{$curr_imageID};
    }
  }

  # Creates ACTUAL_INPUT entries in the database for a node's input
  # link and a list of attributes.
  sub __createActualInputs {
    my ($input, $attribute_list) = @_;
    my $formal_input = $input->to_input();

    foreach my $attribute (@$attribute_list) {
      my $actual_input_data = {
        analysis     => __getAnalysis($curr_nodeID),
        formal_input => $formal_input,
        attribute_id => $attribute->id()
      };
      my $actual_input =
        $factory->newObject("OME::Analysis::ActualInput",
        $actual_input_data);
    }
  }

  # Creates ACTUAL_OUTPUT entries in the database for a formal
  # output and a list of attributes.
  sub __createActualOutputs {
    my ($output, $attribute_list) = @_;
    my $formal_output = $output;

    foreach my $attribute (@$attribute_list) {
      my $actual_output_data = {
        analysis      => __getAnalysis($curr_nodeID),
        formal_output => $formal_output,
        attribute_id  => $attribute->id()
      };
      my $actual_output =
        $factory->newObject("OME::Analysis::ActualOutput",
        $actual_output_data);
    }
  }

  # If any of the predecessors has not finished, then this
  # node is not ready to run.
  sub __testModulePredecessors {
    my $ready = 1;
  TEST_PRED:
    foreach my $granularity ('D', 'I', 'F') {
      my $inputs = $input_links{$curr_nodeID}->{$granularity};
      foreach my $input (@$inputs) {
        my $pred_node = $input->from_node();
        if ($node_states{$pred_node->id()} < FINISHED_STATE) {
          $ready = 0;
          last TEST_PRED;
        }
      }
    }

    return $ready;
  }

  # Determines whether the current node is a per-dataset or
  # per-image module.  If a module takes in any dataset inputs, or
  # outputs any dataset outputs, or if any of its immediate
  # predecessors nodes are per-dataset, then it as per-dataset.
  # Otherwise, it is per-image.  This notion of dataset-dependency
  # comes in to play later when determine whether or not a module's
  # results can be reused.
  sub __determineDependence {
    if (scalar(@curr_dataset_inputs) > 0) {
      $dependence{$curr_nodeID} = 'D';
      return;
    }

    if (scalar(@curr_dataset_outputs) > 0) {
      $dependence{$curr_nodeID} = 'D';
      return;
    }

    foreach my $granularity ('D', 'I', 'F') {
      my $inputs = $input_links{$curr_nodeID}->{$granularity};
      foreach my $input (@$inputs) {
        my $pred_node = $input->from_node();
        if ($dependence{$pred_node->id()} eq 'D') {
          $dependence{$curr_nodeID} = 'D';
          return;
        }
      }
    }

    $dependence{$curr_nodeID} = 'I';
  }

  # The following routines are used to check to see if we need to
  # execute the current module, or if it can be reused.
  #
  # We allow results to be reused if the "input tag" of the current
  # module's state is equal to the input tag of a previous execution
  # of the same module.  The input tag is a string that captures the
  # essence of a module's input.  It records: whether the module was
  # run in a per-dataset manner, or a per-image manner; which
  # dataset or image (respectively) it was run against; and the
  # attribute ID's presented to the module as input.

  # This routine calculates the input tag of the current module.
  # This routine will not get called unless a module is ready to be
  # executed, which means that the results of the predecessor
  # modules are available.  It is the attribute ID's of these
  # results that are encoded into the input tag.
  sub __calculateCurrentInputTag {
    my $paramString;

    if ($dependence{$curr_nodeID} eq 'D') {
      $paramString = "D " . $dataset->id() . " ";
    } else {
      $paramString = "I " . $curr_imageID . " ";
    }

    $paramString .= "d ";
    foreach my $input (@curr_dataset_inputs) {
      my $formal_input  = $input->to_input();
      my $formal_output = $input->from_output();
      my $pred_node     = $input->from_node();
      $paramString .= $formal_input->id() . "(";
      my $attribute_list =
        $dataset_outputs{$pred_node->id()}->{$formal_output->id()};
      if (ref($attribute_list) ne 'ARRAY') {
        $attribute_list = [$attribute_list];
      }
      foreach my $attribute (sort { $a->id() <=> $b->id() }
        @$attribute_list)
      {
        $paramString .= $attribute->id() . " ";
      }
      $paramString .= ") ";
    }

    $paramString .= "i ";
    foreach my $input (@curr_image_inputs) {
      my $formal_input  = $input->to_input();
      my $formal_output = $input->from_output();
      my $pred_node     = $input->from_node();
      $paramString .= $formal_input->id() . "(";
      my $attribute_hash =
        $image_outputs{$pred_node->id()}->{$formal_output->id()};
      my @attribute_list;

      if ($dependence{$curr_nodeID} eq 'D') {
        foreach my $imageID (keys %$attribute_hash) {
          push @attribute_list, @{$attribute_hash->{$imageID}};
        }
      } else {
        push @attribute_list, @{$attribute_hash->{$curr_imageID}};
      }

      foreach
        my $attribute (sort { $a->id() <=> $b->id() } @attribute_list)
      {
        $paramString .= $attribute->id() . " ";
      }
      $paramString .= ") ";
    }

    $paramString .= "f ";
    foreach my $input (@curr_feature_inputs) {
      my $formal_input  = $input->to_input();
      my $formal_output = $input->from_output();
      my $pred_node     = $input->from_node();
      $paramString .= $formal_input->id() . "(";
      my $attribute_hash =
        $feature_outputs{$pred_node->id()}->{$formal_output->id()};
      my @attribute_list;

      if ($dependence{$curr_nodeID} eq 'D') {
        foreach my $imageID (keys %$attribute_hash) {
          my $feature_hash = $attribute_hash->{$imageID};
          foreach my $featureID (keys %$feature_hash) {
            push @attribute_list, @{$feature_hash->{$featureID}};
          }
        }
      } else {
        my $feature_hash = $attribute_hash->{$curr_imageID};
        foreach my $featureID (keys %$feature_hash) {
          push @attribute_list, @{$feature_hash->{$featureID}};
        }
      }

      foreach
        my $attribute (sort { $a->id() <=> $b->id() } @attribute_list)
      {
        $paramString .= $attribute->id() . " ";
      }
      $paramString .= ") ";
    }

    return $paramString;
  }

  # This routine calculates the input tag of a previous analysis.
  # All of the appropriate elements of the tag can be retrieved from
  # the ANALYSES and ACTUAL_INPUTS tables.
  sub __calculatePastInputTag {
    my ($past_analysis) = @_;
    my $past_paramString;

    if ($past_analysis->dependence() eq 'D') {
      $past_paramString =
        "D " . $past_analysis->dataset()->id() . " ";
    } else {
      $past_paramString = "I " . $past_analysis->image()->id() . " ";
    }

    my @all_actuals = $past_analysis->inputs();
    my %actuals;
    foreach my $actual (@all_actuals) {
      my $formal_input = $actual->formal_input();
      push @{$actuals{$formal_input->id()}}, $actual;
    }

    foreach my $formal_inputID (keys %actuals) {
      my @sorted =
        sort { $a->attribute_id() <=> $b->attribute_id() }
        @{$actuals{$formal_inputID}};
      $actuals{$formal_inputID} = \@sorted;
    }

    $past_paramString .= "d ";
    foreach my $input (@curr_dataset_inputs) {
      my $formal_input = $input->to_input();
      $past_paramString .= $formal_input->id() . "(";
      my $actuals = $actuals{$formal_input->id()};

      foreach my $actual (@$actuals) {
        $past_paramString .= $actual->attribute_id() . " ";
      }
      $past_paramString .= ") ";
    }

    $past_paramString .= "i ";
    foreach my $input (@curr_image_inputs) {
      my $formal_input = $input->to_input();
      $past_paramString .= $formal_input->id() . "(";
      my $actuals = $actuals{$formal_input->id()};

      foreach my $actual (@$actuals) {
        $past_paramString .= $actual->attribute_id() . " ";
      }
      $past_paramString .= ") ";
    }

    $past_paramString .= "f ";
    foreach my $input (@curr_feature_inputs) {
      my $formal_input = $input->to_input();
      $past_paramString .= $formal_input->id() . "(";
      my $actuals = $actuals{$formal_input->id()};

      foreach my $actual (@$actuals) {
        $past_paramString .= $actual->attribute_id() . " ";
      }
      $past_paramString .= ") ";
    }

    return $past_paramString;
  }

  # This routine performs the actual reuse of results.  It does not
  # create new entries in ANALYSES, ACTUAL_INPUTS, or
  # ACTUAL_OUTPUTS, since the module is not actually run again.
  # Rather, it updates the internal state of the fixed-point loop to
  # include the already-calculated results, and (will soon add) the
  # appropriate mappings to the ANALYSIS_PATH_MAP table.
```

```
sub __copyPastResults {
  my ($matched_analysis) = @_;

  my @all_outputs = $matched_analysis->outputs();
  my %actuals;
  foreach my $actual_output (@all_outputs) {
    my $formal_output = $actual_output->formal_output();
    my $datatype      = $formal_output->column_type()->datatype();
    my $pkg           = $datatype->requireAttributePackage();
    my $attribute     =
      $factory->loadObject($pkg, $actual_output->attribute_id());

    push @{$actuals{$actual_output->formal_output()->id()}},
      $attribute;
  }

  foreach my $formal_output (@curr_dataset_outputs) {
    my $attribute_list = $actuals{$formal_output->id()};

    $dataset_outputs{$curr_nodeID}->{$formal_output->id()}  =
      $attribute_list;
  }

  foreach my $formal_output (@curr_image_outputs) {
    my $attribute_list = $actuals{$formal_output->id()};

    foreach my $attribute (@$attribute_list) {
      my $image = $attribute->image_id();
      push @{$image_outputs{$curr_nodeID}->{$formal_output->id()}
          ->{$image->id()}}, $attribute;
    }
  }

  foreach my $formal_output (@curr_feature_outputs) {
    my $attribute_list = $actuals{$formal_output->id()};

    foreach my $attribute (@$attribute_list) {
      my $feature = $attribute->feature();
      my $image   = $feature->image();
      push @{$feature_outputs{$curr_nodeID}->{$formal_output->id()}
          ->{$image->id()}->{$feature->id()}}, $attribute;
    }
  }
}

# Updates the hash of ANALYSIS entries.  Takes into account the
# dataset-dependency of the current module.
```

```
sub __assignAnalysis {
  my ($analysis) = @_;
  if ($dependence{$curr_nodeID} eq 'D') {
    $perdataset_analysis{$curr_nodeID} = $analysis;
  } else {
    $perimage_analysis{$curr_nodeID}->{$curr_imageID} = $analysis;
  }
  foreach my $data_path_entry (@{$data_paths{$curr_nodeID}}) {
    my $path_map = $factory->newObject(
      "OME::AnalysisPath::Map",
      {
        path               => $data_path_entry->{path},
        path_order         => $data_path_entry->{path_order},
        analysis           => $analysis,
        analysis_execution => $analysis_execution
      }
    );
  }
}

# This routine performs the check that determines whether results
# can be reused, using the methods described above.
sub __checkPastResults {
  my $paramString = __calculateCurrentInputTag();
  my $space = ($dependence{$curr_nodeID} eq 'D') ? '' : '  ';
  print STDERR "$space  Param $paramString\n";

  my $match = 0;
  my $matched_analysis;
  my @past_analyses =
    OME::Analysis->search(
    program_id => $curr_node->program()->id());

FIND_MATCH:
  foreach my $past_analysis (@past_analyses) {
    my $past_paramString = __calculatePastInputTag($past_analysis);
    print STDERR "$space    Found $past_paramString\n";

    if ($past_paramString eq $paramString) {
      $match            = 1;
      $matched_analysis = $past_analysis;
      last FIND_MATCH;
    }
  }

  if ($match) {
    print STDERR "$space    Found reusable analysis\n";
    __copyPastResults($matched_analysis);
    __assignAnalysis($matched_analysis);
  }

  return $match;
}

# The main body of the analysis engine.  Its purpose is to execute
# a prebuilt analysis chain against a dataset, reusing results if
# possible.
sub executeAnalysisView {
  ($session, $analysis_view, $input_parameters, $dataset) = @_;
  $factory = $session->Factory();

  # all nodes
  @nodes = $analysis_view->nodes();

  print STDERR "Setup\n";

  print STDERR "  Creating ANALYSIS_EXECUTION table entry\n";

  $analysis_execution = $factory->newObject(
    "OME::AnalysisExecution",
    {
      analysis_view => $analysis_view,
      dataset       => $dataset,
      experimenter  => $session->User()
    }
  );

  # initialize all of the nodes
  foreach my $node (@nodes) {
    $curr_node   = $node;
    $curr_nodeID = $curr_node->id();
    __initializeNode();
  }

  # Build the data paths.
  __buildDataPaths();

  $continue = 1;
  $round    = 0;

  while ($continue) {
    $continue = 0;
    $round++;
    print STDERR "Round $round...\n";
    # Look for input_nodes that are ready to run (i.e., whose
    # predecessor nodes have been completed).
  ANALYSIS_LOOP:
    foreach my $node (@nodes) {
      $curr_node   = $node;
      $curr_nodeID = $curr_node->id();

      # Go ahead and skip if we've completed this module.
      if ($node_states{$curr_nodeID} > INPUT_STATE) {
        print STDERR "  "
          . $curr_node->program()->program_name()
          . " already completed\n";
        next ANALYSIS_LOOP;
      }

      if (!__testModulePredecessors()) {
        print STDERR "  Skipping "
          . $curr_node->program()->program_name() . "\n";
        next ANALYSIS_LOOP;
      }

      $curr_module  = $node_modules{$curr_nodeID};
      $curr_inputs  = $input_links{$curr_nodeID};
      $curr_outputs = $output_links{$curr_nodeID};

      @curr_dataset_inputs =
        sort { $a->to_input()->id() <=> $b->to_input()->id() }
        @{$curr_inputs->{D}};
      @curr_image_inputs =
        sort { $a->to_input()->id() <=> $b->to_input()->id() }
        @{$curr_inputs->{I}};
      @curr_feature_inputs =
        sort { $a->to_input()->id() <=> $b->to_input()->id() }
        @{$curr_inputs->{F}};

      @curr_dataset_outputs =
        sort { $a->id() <=> $b->id() } @{$curr_outputs->{D}};
      @curr_image_outputs =
        sort { $a->id() <=> $b->id() } @{$curr_outputs->{I}};
      @curr_feature_outputs =
        sort { $a->id() <=> $b->id() } @{$curr_outputs->{F}};

      $last_node = $curr_node;

      __determineDependence();
```

```
if ($dependence{$curr_nodeID} eq 'D') {
  if (__checkPastResults()) {
    print STDERR "    Marking state\n";
    $node_states{$curr_nodeID} = FINISHED_STATE;
    $continue = 1;
    next ANALYSIS_LOOP;
  }
}

print STDERR "  Executing "
  . $curr_node->program()->program_name() . " ("
  . $dependence{$curr_nodeID} . ")\n";

# Execute away.
if ($dependence{$curr_nodeID} eq 'D') {
  print STDERR "    Creating ANALYSIS entry\n";
  my $analysis = $factory->newObject(
    "OME::Analysis",
    {
      program    => $curr_node->program(),
      dependence => 'D',
      dataset    => $dataset,
      image      => undef,
      timestamp  => 'now',
      status     => 'RUNNING'
    }
  );
  __assignAnalysis($analysis);
}

print STDERR "    startDataset\n";
$curr_module->startDataset($dataset);

# Collect and present the dataset inputs
my %dataset_hash;
foreach my $input (@curr_dataset_inputs) {
  my $formal_input   = $input->to_input();
  my $formal_output  = $input->from_output();
  my $pred_node      = $input->from_node();
  my $attribute_list =
    $dataset_outputs{$pred_node->id()}->{$formal_output->id()
    };
  if (ref($attribute_list) ne 'ARRAY') {
    $attribute_list = [$attribute_list];
  }
  __createActualInputs($input, $attribute_list);
```

```
      $dataset_hash{$formal_input->name()} = $attribute_list;
    }
    $curr_module->datasetInputs(\%dataset_hash);

    print STDERR "    Precalculate dataset\n";
    $curr_module->precalculateDataset();

    my $image_maps = $dataset->image_links();
  IMAGE_LOOP:
    while (my $image_map = $image_maps->next()) {

      # Collect and present the image inputs
      $curr_image   = $image_map->image();
      $curr_imageID = $curr_image->id();

      print STDERR "    Image " . $curr_image->name() . "\n";

      if ($dependence{$curr_nodeID} eq 'I') {
        if (__checkPastResults()) {
          next IMAGE_LOOP;
        }

        print STDERR "    Creating ANALYSIS entry\n";
        my $analysis = $factory->newObject(
          "OME::Analysis",
          {
            program    => $curr_node->program(),
            dependence => 'I',
            dataset    => undef,
            image      => $curr_image,
            timestamp  => 'now',
            status     => 'RUNNING'
          }
        );
        __assignAnalysis($analysis);
      }

      print STDERR "    startImage\n";
      $curr_module->startImage($curr_image);

      my %image_hash;

      foreach my $input (@curr_image_inputs) {
        my $formal_input   = $input->to_input();
        my $formal_output  = $input->from_output();
        my $pred_node      = $input->from_node();
        my $attribute_list =
          $image_outputs{$pred_node->id()}->{$formal_output->id()}
          ->{$curr_imageID};
        if (ref($attribute_list) ne 'ARRAY') {
          $attribute_list = [$attribute_list];
        }
        __createActualInputs($input, $attribute_list);

        $image_hash{$formal_input->name()} = $attribute_list;
      }

      $curr_module->imageInputs(\%image_hash);

      print STDERR "    Precalculate image\n";
      $curr_module->precalculateImage();

      # Collect and present the feature inputs.

      #print STDERR "    startFeature ".$feature->id()."\n";
      #$curr_module->startFeature($feature);

      my %feature_hash;

      foreach my $input (@curr_feature_inputs) {
        my $formal_input   = $input->to_input();
        my $formal_output  = $input->from_output();
        my $pred_node      = $input->from_node();
        my $attribute_hash =
          $feature_outputs{$pred_node->id()}
          ->{$formal_output->id()}->{$curr_imageID};
        foreach my $feature_id (keys %$attribute_hash) {
          my $attribute_list = $attribute_hash->{$feature_id};

          if (ref($attribute_list) ne 'ARRAY') {
            $attribute_list = [$attribute_list];
          }
          __createActualInputs($input, $attribute_list);
        }

        $feature_hash{$formal_input->name()} = $attribute_hash;
      }

      $curr_module->featureInputs(\%feature_hash);

      print STDERR "    Calculate feature\n";
      $curr_module->calculateFeature();

      # Collect and process the feature outputs
      my $feature_attributes =
        $curr_module->collectFeatureOutputs();

      print STDERR "    Feature outputs\n";
      foreach my $output (@curr_feature_outputs) {
        my $formal_output  = $output;
        my $attribute_hash =
          $feature_attributes->{$formal_output->name()};
        foreach my $feature_id (keys %$attribute_hash) {
          my $attribute_list = $attribute_hash->{$feature_id};
          if (ref($attribute_list) ne 'ARRAY') {
            $attribute_list = [$attribute_list];
          }
          __createActualOutputs($output, $attribute_list);
        }
        $feature_outputs{$curr_nodeID}->{$formal_output->id()}
          ->{$curr_imageID} = $attribute_hash;
      }

      # Collect and process the image outputs
      print STDERR "    Postcalculate image\n";
      $curr_module->postcalculateImage();

      my $image_attributes = $curr_module->collectImageOutputs();

      print STDERR "    Image outputs\n";
      foreach my $output (@curr_image_outputs) {
        my $formal_output  = $output;
        my $attribute_list =
          $image_attributes->{$formal_output->name()};
        if (ref($attribute_list) ne 'ARRAY') {
          $attribute_list = [$attribute_list];
        }
        __createActualOutputs($output, $attribute_list);
        $image_outputs{$curr_nodeID}->{$formal_output->id()}
          ->{$curr_imageID} = $attribute_list;
      }

      $curr_module->finishImage($curr_image);
    }    # foreach $curr_image

    # Collect and process the dataset outputs
    print STDERR "    Postcalculate dataset\n";
    $curr_module->postcalculateDataset();

    my $dataset_attributes =
          $curr_module->collectDatasetOutputs();

        print STDERR "    Dataset outputs\n";
        foreach my $output (@curr_dataset_outputs) {
          my $formal_output  = $output;
          my $attribute_list =
            $dataset_attributes->{$formal_output->name()};
          if (ref($attribute_list) ne 'ARRAY') {
            $attribute_list = [$attribute_list];
          }
          __createActualOutputs($output, $attribute_list);
          $dataset_outputs{$curr_nodeID}->{$formal_output->id()}  =
            $attribute_list;
        }

        $curr_module->finishDataset($dataset);

        # Mark this node as finished, and flag that we need
        # another fixed point iteration.

        print STDERR "    Marking state\n";
        $node_states{$curr_nodeID} = FINISHED_STATE;
        $continue = 1;
      }    # ANALYSIS_LOOP - foreach $curr_node
    }    # while ($continue)

    $last_node->dbi_commit();
  }

}

1;
```

Figure 39 – `OME::Tasks::AnalysisEngine` module

B.2 Analysis handlers

The following Perl modules are associated with the OME analysis handlers.

B.2.1 Handler interface

This is the interface to which each analysis handler must conform.

```
package OME::Analysis::Handler;
```

```
use strict;
our $VERSION = '1.0';

use fields qw(_location _factory _program);

sub new {
  my ($proto, $location, $factory, $program) = @_;
  my $class = ref($proto) || $proto;

  my $self = {};
  $self->{_location} = $location;
  $self->{_factory}  = $factory;
  $self->{_program}  = $program;

  bless $self, $class;
  return $self;
}

sub startDataset {
  my ($self, $dataset) = @_;
}

sub datasetInputs {
  my ($self, $inputHash) = @_;
}

sub precalculateDataset() {
  my ($self) = @_;
}

sub startImage {
  my ($self, $image) = @_;
}

sub imageInputs {
  my ($self, $inputHash) = @_;
}

sub precalculateImage() {
  my ($self) = @_;
}

sub startFeature {
  my ($self, $feature) = @_;
}

sub featureInputs {
  my ($self, $inputHash) = @_;
}

sub calculateFeature {
  my ($self) = @_;
}

sub collectFeatureOutputs {
  my ($self) = @_;
  return {};
}

sub finishFeature {
  my ($self) = @_;
}

sub postcalculateImage() {
  my ($self) = @_;
}

sub collectImageOutputs {
  my ($self) = @_;
  return {};
}

sub finishImage {
  my ($self) = @_;
}

sub postcalculateDataset {
  my ($self) = @_;
}

sub collectDatasetOutputs {
  my ($self) = @_;
  return {};
}

sub finishDataset {
  my ($self) = @_;
}

1;
```

Figure 40 – `OME::Analysis::Handler` module

B.2.2 Perl handler

The following modules are used for analysis modules written directly in Perl. The code for the analysis routine itself should be a subclass of the `OME::Analysis::PerlAnalysis` class (Figure 41), and should use the `OME::Analysis::PerlHandler` class (Figure 42) as is for its analysis handler.

```
package OME::Analysis::PerlAnalysis;

use strict;
our $VERSION = '1.0';

use fields qw(_factory
  _currentDataset _currentImage _currentFeature
  _datasetInputs  _imageInputs  _featureInputs
  _datasetOutputs  _imageOutputs  _featureOutputs);

sub new {
  my ($proto, $factory) = @_;
  my $class = ref($proto) || $proto;

  my $self = {};

  $self->{_factory} = $factory;

  $self->{_currentDataset} = undef;
  $self->{_datasetInputs}  = undef;
  $self->{_datasetOutputs} = undef;

  $self->{_currentImage} = undef;
  $self->{_imageInputs}  = undef;
  $self->{_imageOutputs} = undef;

  $self->{_currentFeature} = undef;
  $self->{_featureInputs}  = undef;
  $self->{_featureOutputs} = undef;

  bless $self, $class;
  return $self;
}

sub startDataset {
  my ($self, $dataset) = @_;
  $self->{_currentDataset} = $dataset;
```

```
}

sub datasetInputs {
  my ($self, $inputHash) = @_;
  $self->{_datasetInputs} = $inputHash;
}

sub precalculateDataset {
  my ($self) = @_;
}

sub startImage {
  my ($self, $image) = @_;
  $self->{_currentImage} = $image;
}

sub imageInputs {
  my ($self, $inputHash) = @_;
  $self->{_imageInputs} = $inputHash;
}

sub precalculateImage {
  my ($self) = @_;
}

sub startFeature {
  my ($self, $feature) = @_;
  $self->{_currentFeature} = $feature;
}

sub featureInputs {
  my ($self, $inputHash) = @_;
  $self->{_featureInputs} = $inputHash;
}

sub calculateFeature {
  my ($self) = @_;
}

sub collectFeatureOutputs {
  my ($self) = @_;
  return $self->{_featureOutputs};
}

sub finishFeature {
  my ($self) = @_;
  $self->{_currentFeature} = undef;
  $self->{_featureInputs}  = undef;
  $self->{_featureOutputs} = undef;
}

sub postcalculateImage {
  my ($self) = @_;
}

sub collectImageOutputs {
  my ($self) = @_;
  return $self->{_imageOutputs};
}

sub finishImage {
  my ($self) = @_;
  $self->{_currentImage} = undef;
  $self->{_imageInputs}  = undef;
  $self->{_imageOutputs} = undef;
}

sub postcalculateDataset {
  my ($self) = @_;
}

sub collectDatasetOutputs {
  my ($self) = @_;
  return $self->{_datasetOutputs};
}

sub finishDataset {
  my ($self) = @_;
  $self->{_currentDataset} = undef;
  $self->{_datasetInputs}  = undef;
  $self->{_datasetOutputs} = undef;
}

1;
```

Figure 41 – `OME::Analysis::PerlAnalysis` module

```
package OME::Analysis::PerlHandler;

use strict;
our $VERSION = '1.0';

use base qw(OME::Analysis::Handler);
use fields qw(_instance);

sub new {
  my ($proto, $location, $factory, $program) = @_;
  my $class = ref($proto) || $proto;

  my $self = $class->SUPER::new($location, $factory, $program);
  eval "require $location";
  $self->{_instance} = $location->new($factory);

  bless $self, $class;
  return $self;
}

sub startDataset {
  my ($self, $dataset) = @_;
  return $self->{_instance}->startDataset($dataset);
}

sub datasetInputs {
  my ($self, $inputHash) = @_;
  return $self->{_instance}->datasetInputs($inputHash);
}

sub precalculateDataset {
  my ($self) = @_;
  return $self->{_instance}->precalculateDataset();
}

sub startImage {
  my ($self, $image) = @_;
  return $self->{_instance}->startImage($image);
}

sub imageInputs {
  my ($self, $inputHash) = @_;
  return $self->{_instance}->imageInputs($inputHash);
}

sub precalculateImage {
  my ($self) = @_;
  return $self->{_instance}->precalculateImage();
}

sub startFeature {
  my ($self, $feature) = @_;
  return $self->{_instance}->startFeature($feature);
}

sub featureInputs {
  my ($self, $inputHash) = @_;
  return $self->{_instance}->featureInputs($inputHash);
}

sub calculateFeature {
  my ($self) = @_;
  return $self->{_instance}->calculateFeature();
}

sub collectFeatureOutputs {
  my ($self) = @_;
  return $self->{_instance}->collectFeatureOutputs();
}

sub finishFeature {
  my ($self) = @_;
  return $self->{_instance}->finishFeature();
}

sub postcalculateImage {
  my ($self) = @_;
  return $self->{_instance}->postcalculateImage();
}

sub collectImageOutputs {
  my ($self) = @_;
  return $self->{_instance}->collectImageOutputs();
}

sub finishImage {
  my ($self) = @_;
  return $self->{_instance}->finishImage();
}

sub postcalculateDataset {
  my ($self) = @_;
  return $self->{_instance}->postcalculateDataset();
}

sub collectDatasetOutputs {
  my ($self) = @_;
  return $self->{_instance}->collectDatasetOutputs();
}
```

```
sub finishDataset {
  my ($self) = @_;
  return $self->{_instance}->finishDataset();
}

1;
```

Figure 42 – `OME::Analysis::PerlHandler` module

B.2.3 Command-line handlers

The following classes are used to integrate existing command-line programs into OME as analysis modules. Currently, a separate handler had to be written for each analysis. Another solution would have been to write separate analysis "wrappers" for each utility; these wrappers would have been defined as subclasses of `OME::Analysis::PerlAnalysis` (Figure 41) rather than `OME::Analysis::Handler` (Figure 40), but would otherwise have remained basically the same.

```
package OME::Analysis::CLIHandler;

use strict;
our $VERSION = '1.0';

use IO::File;

use base qw(OME::Analysis::Handler);

use fields qw(_outputHandle _currentImage);

sub new {
  my ($proto, $location, $factory, $program) = @_;
  my $class = ref($proto) || $proto;

  my $self = $class->SUPER::new($location, $factory, $program);

  bless $self, $class;
  return $self;
}

sub startDataset {
  my ($self, $dataset) = @_;
}

sub datasetInputs {
  my ($self, $inputHash) = @_;
}

sub precalculateDataset {
  my ($self) = @_;
}

sub startImage {
  my ($self, $image) = @_;

  my $dims = $image->Dimensions();

  my $dimString = "Dims="
    . $dims->size_x() . ","
    . $dims->size_y() . ","
    . $dims->size_z() . ","
    . $dims->num_waves() . ","
    . $dims->num_times() . ","
    . $dims->bits_per_pixel() / 8;

  my $pathString = "Path=" . $image->getFullPath();

  my $output   = new IO::File;
  my $location = $self->{_location};
  open $output, "$location $pathString $dimString |"
    or die "Cannot open analysis program";

  print STDERR "      $location $pathString $dimString\n";

  $self->{_outputHandle} = $output;
  $self->{_currentImage} = $image;
}

sub imageInputs {
  my ($self, $inputHash) = @_;
}

sub precalculateImage {
  my ($self) = @_;
}

sub startFeature {
  my ($self, $feature) = @_;
```

```
}

sub featureInputs {
  my ($self, $inputHash) = @_;
}

sub calculateFeature {
  my ($self) = @_;
}

sub collectFeatureOutputs {
  my ($self) = @_;
  return {};
}

sub finishFeature {
  my ($self) = @_;
}

sub postcalculateImage {
  my ($self) = @_;
}

sub collectImageOutputs {
  my ($self) = @_;
  my $output  = $self->{_outputHandle};
  my $program = $self->{_program};
  my $image   = $self->{_currentImage};
  my $factory = $self->{_factory};

  my $headerString = <$output>;
  chomp $headerString;
  my @headers = split ("\t", $headerString);

  my %outputs;
  my @outputs = $program->outputs();
  foreach my $formal_output (@outputs) {

    #print STDERR "      - ".$formal_output->name()."\n";
    $outputs{$formal_output->name()} = $formal_output;
  }

  my %imageOutputs;
  my @attributes;

  while (my $input = <$output>) {
    chomp $input;
    my @data = split ("\t", $input);
    my $count = 0;
    my %attributes;
    foreach my $datum (@data) {
      my $output_name = $headers[$count];

      my $formal_output = $outputs{$output_name};
      my $column_type   = $formal_output->column_type();
      my $column_name   = lc($column_type->column_name());
      my $datatype      = $column_type->datatype();
      my $attribute;
      if (exists $attributes{$datatype->id()}) {
        $attribute = $attributes{$datatype->id()};
      } else {
        my $pkg = $datatype->requireAttributePackage();
        $attribute =
          $factory->newObject($pkg, {image_id => $image->id()});

        # so we can find it later
        $attributes{$datatype->id()} = $attribute;

        # so we can commit it later
        push @attributes, $attribute;
      }

      $attribute->set($column_name, $datum);
      push @{$imageOutputs{$formal_output->name()}}, $attribute;
      $count++;
    }
  }

  foreach my $attribute (@attributes) {
    $attribute->commit();
  }

  return \%imageOutputs;
}

sub finishImage {
  my ($self) = @_;
}

sub postcalculateDataset {
  my ($self) = @_;
}

sub collectDatasetOutputs {
  my ($self) = @_;
  return {};
}

sub finishDataset {
  my ($self) = @_;
}

1;
```

Figure 43 – `OME::Analysis::CLIHandler` module

```
package OME::Analysis::FindSpotsHandler;

use strict;
our $VERSION = '1.0';

use IO::File;
use OME::Analysis::CLIHandler;

use base qw(OME::Analysis::CLIHandler);

use fields qw(_options _inputHandle _outputHandle _errorHandle
  _inputFile _outputFile _errorFile _features _cmdLine);

sub new {
  my ($proto, $location, $factory, $program) = @_;
  my $class = ref($proto) || $proto;

  my $self = $class->SUPER::new($location, $factory, $program);

  $self->{_options} =
    "0 gmean4.5s 10 -db -tt -th -c 0 -i 0 -m 0 -g 0 ".
    "-ms 0 -gs 0 -mc -v -sa -per -ff";

  bless $self, $class;
  return $self;
}

sub startImage {
  my ($self, $image) = @_;

  my $path     = $image->getFullPath();
  my $location = $self->{_location};
  my $options  = $self->{_options};
  my $cmdLine  = "$location $path $options";
  my ($input, $output, $error, $pid);
  my $session   = $self->{_factory}->Session();
  my $inputFile =
    $session->getTemporaryFilename("findSpots", "stdin");
  my $outputFile =
    $session->getTemporaryFilename("findSpots", "stdout");
  my $errorFile =
    $session->getTemporaryFilename("findSpots", "stderr");

  $input  = new IO::File;
  $output = new IO::File;
  $error  = new IO::File;
  open $input, "> $inputFile";

  print STDERR "      $location $path $options\n";

  $self->{_inputHandle}  = $input;
  $self->{_outputHandle} = $output;
  $self->{_errorHandle}  = $error;
  $self->{_inputFile}    = $inputFile;
  $self->{_outputFile}   = $outputFile;
  $self->{_errorFile}    = $errorFile;
  $self->{_currentImage} = $image;
  $self->{_cmdLine}      = $cmdLine;
}

sub imageInputs {
  my ($self, $inputHash) = @_;

  my $input = $self->{_inputHandle};

  my $image     = $self->{_currentImage};
  my $dims      = $image->Dimensions();
  my $dimString = "Dims="
    . $dims->size_x() . ","
    . $dims->size_y() . ","
    . $dims->size_z() . ","
    . $dims->num_waves() . ","
    . $dims->num_times();

  print $input "$dimString\nWaveStats=\n";

  my $attribute_list = $inputHash->{Wavelength};
  my %wave_stats;
  foreach my $attribute (@$attribute_list) {
    my @stats = (
      $attribute->wavenumber(), $attribute->wavenumber(),
      $attribute->timepoint(),  $attribute->min(),
      $attribute->max(),        $attribute->mean(),
      $attribute->geomean(),    $attribute->sigma()
    );
    my $wave_stat = join (',', @stats);
    $wave_stats{$attribute->timepoint()}->{$attribute->wavenumber()}  =
      $wave_stat;

    #print STDERR "        $wave_stat\n";
    #print $input "$wave_stat\n";
  }

  foreach my $time (sort { $a <=> $b } (keys %wave_stats)) {
    my $stats = $wave_stats{$time};
    foreach my $wave (sort { $a <=> $b } (keys %$stats)) {
      my $wave_stat = $stats->{$wave};
      print STDERR "        $wave_stat\n";
      print $input "$wave_stat\n";
    }
  }

  close $input;

  my $cmdLine = $self->{_cmdLine};
  system("$cmdLine < "
      . $self->{_inputFile} . " > "
      . $self->{_outputFile} . " 2> "
      . $self->{_errorFile});

  open $self->{_outputHandle}, "< " . $self->{_outputFile};
  open $self->{_errorHandle},  "< " . $self->{_errorFile};

  $self->{_features} = [];
}

sub precalculateImage {
  my ($self) = @_;
}

sub startFeature {
  my ($self, $feature) = @_;
}

sub featureInputs {
  my ($self, $inputHash) = @_;
}
```

```
sub calculateFeature {
  my ($self) = @_;
}

sub collectFeatureOutputs {
  my ($self) = @_;
  my $factory = $self->{_factory};

  my %feature_outputs;
  my $output = $self->{_outputHandle};

  my $headers = <$output>;
  chomp $headers;
  my @headers;
  foreach my $header (split ("\t", $headers)) {
    $header =~ s/^\s+//;
    $header =~ s/\s+$//;
    push @headers, $header;
  }

  my $image = $self->{_currentImage};

  my $wavelength_rex = qr/^([cimg])\[([ 0-9]+)\]([XYZ])?$/;

  my $spotCount = 0;
  print STDERR "      ";
  while (my $line = <$output>) {
    chomp $line;
    my @data;
    foreach my $datum (split ("\t", $line)) {
      $datum =~ s/^\s+//;
      $datum =~ s/\s+$//;
      push @data, $datum;
    }

    my $feature = $factory->newAttribute(
      'FEATURES',
      {
        image_id => $image->id(),
        name     => "Spot" . $spotCount++
      }
    );
    my $featureID     = $feature->id();
    my $timepointData = {feature_id => $featureID};
    my $thresholdData = {feature_id => $featureID};
    my $locationData  = {feature_id => $featureID};
    my $extentData    = {feature_id => $featureID};
    my %signalData;

    my $i = 0;
    foreach my $datum (@data) {
      my $header = $headers[$i++];

      #print STDERR ".";
      $datum = undef if ($datum eq 'inf');
      if ($header eq "t") {
        $timepointData->{timepoint} = $datum;
      } elsif ($header eq "Thresh.") {
        $thresholdData->{threshold} = $datum;
      } elsif ($header eq "mean X") {
        $locationData->{x} = $datum;
      } elsif ($header eq "mean Y") {
        $locationData->{y} = $datum;
      } elsif ($header eq "mean Z") {
        $locationData->{z} = $datum;
      } elsif ($header eq "volume") {
        $extentData->{volume} = $datum;
      } elsif ($header eq "Surf. Area") {
        $extentData->{surface_area} = $datum;
      } elsif ($header eq "perimiter") {
        $extentData->{perimiter} = $datum;
      } elsif ($header eq "Form Factor") {
        $extentData->{form_factor} = $datum;
      } elsif ($header =~ /$wavelength_rex/) {
        my $c1         = $1;
        my $wavelength = $2;
        my $c2         = $3;

        #print STDERR " '$c1' '$wavelength' '$c2'";

        my $signalData;
        if (!exists $signalData{$wavelength}) {
          $signalData = {
            feature_id => $featureID,
            wavelength => $wavelength
          };
          $signalData{$wavelength} = $signalData;
        } else {
          $signalData = $signalData{$wavelength};
        }

        if (($c1 eq "c") && ($c2 eq "X")) {
          $signalData->{centroid_x} = $datum;
        } elsif (($c1 eq "c") && ($c2 eq "Y")) {
          $signalData->{centroid_y} = $datum;
        } elsif (($c1 eq "c") && ($c2 eq "Z")) {
          $signalData->{centroid_z} = $datum;
        } elsif ($c1 eq "i") {
          $signalData->{integral} = $datum;
        } elsif ($c1 eq "m") {
          $signalData->{mean} = $datum;
        } elsif ($c1 eq "g") {
          $signalData->{geomean} = $datum;
        }
      } else {

        #print STDERR "?";
      }
    }    # foreach datum

    my $timepoint =
      $factory->newAttribute('TIMEPOINT', $timepointData);
    my $threshold =
      $factory->newAttribute('THRESHOLD', $thresholdData);
    my $location = $factory->newAttribute('LOCATION', $locationData);
    my $extent   = $factory->newAttribute('EXTENT',   $extentData);

    my @signals;
    foreach my $signalData (values %signalData) {
      push @signals, $factory->newAttribute('SIGNAL', $signalData);
    }

    # Save the image attribute for later
    push @{$self->{_features}}, $feature;

    # Return the feature attributes
    $feature_outputs{'Timepoint'}->{$featureID}      = $timepoint;
    $feature_outputs{'Threshold'}->{$featureID}      = $threshold;
    $feature_outputs{'X'}->{$featureID}              = $location;
    $feature_outputs{'Y'}->{$featureID}              = $location;
    $feature_outputs{'Z'}->{$featureID}              = $location;
    $feature_outputs{'Volume'}->{$featureID}         = $extent;
    $feature_outputs{'Perimeter'}->{$featureID}      = $extent;
    $feature_outputs{'Surface area'}->{$featureID}   = $extent;
    $feature_outputs{'Form factor'}->{$featureID}    = $extent;
    $feature_outputs{'Wavelength'}->{$featureID}     = [@signals];
    $feature_outputs{'Integral'}->{$featureID}       = [@signals];
    $feature_outputs{'Centroid X'}->{$featureID}     = [@signals];
    $feature_outputs{'Centroid Y'}->{$featureID}     = [@signals];
    $feature_outputs{'Centroid Z'}->{$featureID}     = [@signals];
```

```
    $feature_outputs{'Mean'}->{$featureID}           = [@signals];
    $feature_outputs{'Geometric Mean'}->{$featureID} = [@signals];

    print STDERR "*$spotCount";
  }
  print STDERR "\n";
  print STDERR "*** " . $feature_outputs{'Timepoint'} . "\n";

 #print STDERR "*** ".join(',',@{$feature_outputs{'Timepoint'}})."\n";

  close $self->{_outputHandle};
  close $self->{_errorHandle};

  return \%feature_outputs;
}

sub finishFeature {
  my ($self) = @_;
}

sub postcalculateImage {
  my ($self) = @_;
}

sub collectImageOutputs {
  my ($self) = @_;
  my $image    = $self->{_currentImage};
  my $factory  = $self->{_factory};
  my $features = $self->{_features};

  return {Spots => $features};
}

sub finishImage {
  my ($self) = @_;
}

sub postcalculateDataset {
  my ($self) = @_;
}

sub collectDatasetOutputs {
  my ($self) = @_;
  return {};
}

sub finishDataset {
  my ($self) = @_;
}

1;
```

Figure 44 – `OME::Analysis::FindSpotsHandler` module

B.3 Test cases

The following Perl scripts are used to initialize the database prior to testing the analysis engine.

```
#!/usr/bin/perl -w
# OME/Tests/AnalysisEngine/CreateProgram.pl

use OME::Session;
use OME::SessionManager;
use OME::Program;
use OME::DataType;
use Term::ReadKey;

print "\nOME Test Case - Create programs\n";
print "-------------------------------\n";

if (scalar(@ARGV) != 0) {
  print "Usage:  CreateProgram\n\n";
  exit -1;
}

print "Please login to OME:\n";

print "Username? ";
ReadMode(1);
my $username = ReadLine(0);
chomp($username);

print "Password? ";
ReadMode(2);
my $password = ReadLine(0);
chomp($password);
print "\n";
ReadMode(1);

my $manager = OME::SessionManager->new();
my $session = $manager->createSession($username, $password);
```

```
if (!defined $session) {
  print "That username/password does not seem to be valid.\nBye.\n\n";
  exit -1;
}

print "Great, you're in.\n\n";

my $factory = $session->Factory();
$factory->Debug(0);

print "Finding datatypes...\n";

my $xyzImageInfo = OME::DataType->findByTable('XYZ_IMAGE_INFO');
print "  "
  . $xyzImageInfo->table_name() . " ("
  . $xyzImageInfo->id() . ")\n";

my $xyImageInfo = OME::DataType->findByTable('XY_IMAGE_INFO');
print "  "
  . $xyImageInfo->table_name() . " ("
  . $xyImageInfo->id() . ")\n";

my $features = OME::DataType->findByTable('FEATURES');
print "  " . $features->table_name() . " (" . $features->id() . ")\n";

my $timepoint = OME::DataType->findByTable('TIMEPOINT');
print "  "
  . $timepoint->table_name() . " ("
  . $timepoint->id() . ")\n";

my $threshold = OME::DataType->findByTable('THRESHOLD');
print "  "
  . $threshold->table_name() . " ("
  . $threshold->id() . ")\n";

my $location = OME::DataType->findByTable('LOCATION');
print "  " . $location->table_name() . " (" . $location->id() . ")\n";

my $extent = OME::DataType->findByTable('EXTENT');
print "  " . $extent->table_name() . " (" . $extent->id() . ")\n";

my $signal = OME::DataType->findByTable('SIGNAL');
print "  " . $signal->table_name() . " (" . $signal->id() . ")\n";

print "Creating programs...\n";
```

```
my ($input, $output);

my $calcXyInfo = $factory->newObject(
  "OME::Program",
  {
    program_name => 'Plane statistics',
    description  => 'Calculate pixel statistics for each XY plane',
    category     => 'Statistics',
    module_type  => 'OME::Analysis::CLIHandler',
    location     => '/OME/bin/OME_Image_XY_stats'
  }
);
print "  "
  . $calcXyInfo->program_name() . " ("
  . $calcXyInfo->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyInfo,
    name        => 'Wave',
    column_type => $xyImageInfo->findColumnByName('WAVENUMBER')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyInfo,
    name        => 'Time',
    column_type => $xyImageInfo->findColumnByName('TIMEPOINT')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyInfo,
    name        => 'Z',
    column_type => $xyImageInfo->findColumnByName('ZSECTION')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyInfo,
    name        => 'Min',
    column_type => $xyImageInfo->findColumnByName('MIN')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyInfo,
    name        => 'Max',
    column_type => $xyImageInfo->findColumnByName('MAX')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyInfo,
    name        => 'Mean',
    column_type => $xyImageInfo->findColumnByName('MEAN')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyInfo,
    name        => 'GeoMean',
    column_type => $xyImageInfo->findColumnByName('GEOMEAN')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyInfo,
    name        => 'Sigma',
    column_type => $xyImageInfo->findColumnByName('SIGMA')
  }
);
```

```
print "    " . $output->name() . " (" . $output->id() . ")\n";

my $calcXyzInfo = $factory->newObject(
  "OME::Program",
  {
    program_name => 'Stack statistics',
    description  => 'Calculate pixel statistics for each XYZ stack',
    category     => 'Statistics',
    module_type  => 'OME::Analysis::CLIHandler',
    location     => '/OME/bin/OME_Image_XYZ_stats'
  }
);
print "  "
  . $calcXyzInfo->program_name() . " ("
  . $calcXyzInfo->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Wave',
    column_type => $xyzImageInfo->findColumnByName('WAVENUMBER')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Time',
    column_type => $xyzImageInfo->findColumnByName('TIMEPOINT')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Min',
    column_type => $xyzImageInfo->findColumnByName('MIN')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Max',
    column_type => $xyzImageInfo->findColumnByName('MAX')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Mean',
    column_type => $xyzImageInfo->findColumnByName('MEAN')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'GeoMean',
    column_type => $xyzImageInfo->findColumnByName('GEOMEAN')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Sigma',
    column_type => $xyzImageInfo->findColumnByName('SIGMA')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Centroid_x',
    column_type => $xyzImageInfo->findColumnByName('CENTROID_X')
  }
);
```

```
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Centroid_y',
    column_type => $xyzImageInfo->findColumnByName('CENTROID_Y')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $calcXyzInfo,
    name        => 'Centroid_z',
    column_type => $xyzImageInfo->findColumnByName('CENTROID_Z')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

my $findSpots = $factory->newObject(
  "OME::Program",
  {
    program_name => 'Find spots',
    description  => 'Find spots in the image',
    category     => 'Segmentation',
    module_type  => 'OME::Analysis::FindSpotsHandler',
    location     => '/OME/bin/findSpotsOME'
  }
);
print "  "
  . $findSpots->program_name() . " ("
  . $findSpots->id() . ")\n";

$input = $factory->newObject(
  "OME::Program::FormalInput",
  {
    program     => $findSpots,
    name        => 'Wavelength',
    column_type => $xyzImageInfo->findColumnByName('WAVENUMBER')
  }
);
print "    " . $input->name() . " (" . $input->id() . ")\n";

$input = $factory->newObject(
  "OME::Program::FormalInput",
  {
    program     => $findSpots,
    name        => 'Timepoint',
    column_type => $xyzImageInfo->findColumnByName('TIMEPOINT')
  }
);
print "    " . $input->name() . " (" . $input->id() . ")\n";

$input = $factory->newObject(
  "OME::Program::FormalInput",
  {
    program     => $findSpots,
    name        => 'Minimum',
    column_type => $xyzImageInfo->findColumnByName('MIN')
  }
);
print "    " . $input->name() . " (" . $input->id() . ")\n";

$input = $factory->newObject(
  "OME::Program::FormalInput",
  {
    program     => $findSpots,
    name        => 'Maximum',
    column_type => $xyzImageInfo->findColumnByName('MAX')
  }
);
print "    " . $input->name() . " (" . $input->id() . ")\n";

$input = $factory->newObject(
  "OME::Program::FormalInput",
  {
    program     => $findSpots,
    name        => 'Mean',
    column_type => $xyzImageInfo->findColumnByName('MEAN')
  }
);
print "    " . $input->name() . " (" . $input->id() . ")\n";

$input = $factory->newObject(
  "OME::Program::FormalInput",
  {
    program     => $findSpots,
    name        => 'Geometric mean',
    column_type => $xyzImageInfo->findColumnByName('GEOMEAN')
  }
);
```

```
print "    " . $input->name() . " (" . $input->id() . ")\n";

$input = $factory->newObject(
  "OME::Program::FormalInput",
  {
    program     => $findSpots,
    name        => 'Sigma',
    column_type => $xyzImageInfo->findColumnByName('SIGMA')
  }
);
print "    " . $input->name() . " (" . $input->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Timepoint',
    column_type => $timepoint->findColumnByName('TIMEPOINT')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Threshold',
    column_type => $threshold->findColumnByName('THRESHOLD')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'X',
    column_type => $location->findColumnByName('X')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Y',
    column_type => $location->findColumnByName('Y')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Z',
    column_type => $location->findColumnByName('Z')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Volume',
    column_type => $extent->findColumnByName('VOLUME')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Perimeter',
    column_type => $extent->findColumnByName('PERIMITER')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Surface area',
    column_type => $extent->findColumnByName('SURFACE_AREA')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Form factor',
    column_type => $extent->findColumnByName('FORM_FACTOR')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Wavelength',
    column_type => $signal->findColumnByName('WAVELENGTH')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Integral',
    column_type => $signal->findColumnByName('INTEGRAL')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Centroid X',
    column_type => $signal->findColumnByName('CENTROID_X')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Centroid Y',
    column_type => $signal->findColumnByName('CENTROID_Y')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";
```

```
$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Centroid Z',
    column_type => $signal->findColumnByName('CENTROID_Z')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Mean',
    column_type => $signal->findColumnByName('MEAN')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Geometric Mean',
    column_type => $signal->findColumnByName('GEOMEAN')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output = $factory->newObject(
  "OME::Program::FormalOutput",
  {
    program     => $findSpots,
    name        => 'Spots',
    column_type => $features->findColumnByName('NAME')
  }
);
print "    " . $output->name() . " (" . $output->id() . ")\n";

$output->dbi_commit();

1;
```

Figure 45 – `OME::Tests::AnalysisEngine::CreateProgram` script

```
#!/usr/bin/perl -w
# OME/Tests/AnalysisEngine/CreateView.pl

use OME::Session;
use OME::SessionManager;
use OME::AnalysisView;
use Term::ReadKey;

print "\nOME Test Case - Create views\n";
print "----------------------------\n";

if (scalar(@ARGV) != 0) {
  print "Usage:  CreateView\n\n";
  exit -1;
}

print "Please login to OME:\n";

print "Username? ";
ReadMode(1);
my $username = ReadLine(0);
chomp($username);

print "Password? ";
ReadMode(2);
my $password = ReadLine(0);
chomp($password);
print "\n";
ReadMode(1);

my $manager = OME::SessionManager->new();
my $session = $manager->createSession($username, $password);

if (!defined $session) {
  print "That username/password does not seem to be valid.\nBye.\n\n";
  exit -1;
}

print "Great, you're in.\n\n";

my $factory = $session->Factory();
$factory->Debug(0);

my ($node1, $node2, $link);

print "Finding programs...\n";
```

```
my $calcXyzInfo = OME::Program->findByName('Stack statistics');
print $calcXyzInfo->program_name() . " ("
  . $calcXyzInfo->id() . ")\n";

my $calcXyInfo = OME::Program->findByName('Plane statistics');
print $calcXyInfo->program_name() . " (" . $calcXyInfo->id() . ")\n";

my $findSpots = OME::Program->findByName('Find spots');
print $findSpots->program_name() . " (" . $findSpots->id() . ")\n";

print "Image import chain...\n";

my $view = $factory->newObject(
  "OME::AnalysisView",
  {
    owner => $session->User(),
    name  => "Image import analyses"
  }
);
die "Bad view" if !defined $view;
print "  " . $view->name() . " (" . $view->id() . ")\n";

$node1 = $factory->newObject(
  "OME::AnalysisView::Node",
  {
    analysis_view => $view,
    program       => $calcXyzInfo
  }
);
print "    Node 1 "
  . $node1->program()->program_name() . " ("
  . $node1->id() . ")\n";

$node1 = $factory->newObject(
  "OME::AnalysisView::Node",
  {
    analysis_view => $view,
    program       => $calcXyInfo
  }
);
print "    Node 2 "
  . $node1->program()->program_name() . " ("
  . $node1->id() . ")\n";

print "Find spots chain...\n";
```

```
my $view = $factory->newObject(
  "OME::AnalysisView",
  {
    owner => $session->User(),
    name  => "Find spots"
  }
);
die "Bad view" if !defined $view;
print "  " . $view->name() . " (" . $view->id() . ")\n";

$node1 = $factory->newObject(
  "OME::AnalysisView::Node",
  {
    analysis_view => $view,
    program       => $calcXyzInfo
  }
);
print "    Node 1 "
  . $node1->program()->program_name() . " ("
  . $node1->id() . ")\n";

$node2 = $factory->newObject(
  "OME::AnalysisView::Node",
  {
    analysis_view => $view,
    program       => $findSpots
  }
);
print "    Node 2 "
  . $node2->program()->program_name() . " ("
  . $node2->id() . ")\n";

$link = $factory->newObject(
  "OME::AnalysisView::Link",
  {
    analysis_view => $view,
    from_node     => $node1,
    from_output   => $node1->program()->findOutputByName('Wave'),
    to_node       => $node2,
    to_input      => $node2->program()->findInputByName('Wavelength')
  }
);
print "    Link [Node 1.Wavelength]->[Node 2.Wavelength]\n";

$link = $factory->newObject(
  "OME::AnalysisView::Link",
  {
    analysis_view => $view,
    from_node     => $node1,
    from_output   => $node1->program()->findOutputByName('Time'),
    to_node       => $node2,
    to_input      => $node2->program()->findInputByName('Timepoint')
  }
);
print "    Link [Node 1.Timepoint]->[Node 2.Timepoint]\n";

$link = $factory->newObject(
  "OME::AnalysisView::Link",
  {
    analysis_view => $view,
    from_node     => $node1,
    from_output   => $node1->program()->findOutputByName('Min'),
    to_node       => $node2,
    to_input      => $node2->program()->findInputByName('Minimum')
  }
);
print "    Link [Node 1.Minimum]->[Node 2.Minimum]\n";

$link = $factory->newObject(
  "OME::AnalysisView::Link",
  {
    analysis_view => $view,
    from_node     => $node1,
    from_output   => $node1->program()->findOutputByName('Max'),
    to_node       => $node2,
    to_input      => $node2->program()->findInputByName('Maximum')
  }
);
print "    Link [Node 1.Maximum]->[Node 2.Maximum]\n";

$link = $factory->newObject(
  "OME::AnalysisView::Link",
  {
    analysis_view => $view,
    from_node     => $node1,
    from_output   => $node1->program()->findOutputByName('Mean'),
    to_node       => $node2,
    to_input      => $node2->program()->findInputByName('Mean')
  }
);
print "    Link [Node 1.Mean]->[Node 2.Mean]\n";

$link = $factory->newObject(
  "OME::AnalysisView::Link",
  {
    analysis_view => $view,
    from_node     => $node1,
    from_output   => $node1->program()->findOutputByName('GeoMean'),
    to_node       => $node2,
    to_input => $node2->program()->findInputByName('Geometric mean')
  }
);
print "    Link [Node 1.Geometric mean]->[Node 2.Geometric mean]\n";

$link = $factory->newObject(
  "OME::AnalysisView::Link",
  {
    analysis_view => $view,
    from_node     => $node1,
    from_output   => $node1->program()->findOutputByName('Sigma'),
    to_node       => $node2,
    to_input      => $node2->program()->findInputByName('Sigma')
  }
);
print "    Link [Node 1.Sigma]->[Node 2.Sigma]\n";

$link->dbi_commit();

1;
```

Figure 46 – `OME::Tests::AnalysisEngine::CreateView` script

The `OME::Tests::ImportTest` script was created jointly by the author, Ilya Goldberg, and Brian Hughes.

```
#!/usr/bin/perl -w
# OME/Tests/ImportTest.pl

use OME::Image;
use OME::Dataset;
use OME::Project;
use OME::Session;
use OME::SessionManager;
use OME::Tasks::ImageTasks;
use OME::Project;
use Term::ReadKey;

print "\nOME Test Case - Image Import\n";
print "----------------------------\n";
```

```
if (scalar(@ARGV) == 0) {
  print "Usage:  ImportTest dataset_name [file list]\n\n";
  exit -1;
}

print "Please login to OME:\n";

print "Username? ";
ReadMode(1);
my $username = ReadLine(0);
chomp($username);

print "Password? ";
ReadMode(2);
my $password = ReadLine(0);
chomp($password);
print "\n";
ReadMode(1);

my $manager = OME::SessionManager->new();
my $session = $manager->createSession($username, $password);

if (!defined $session) {
  print "That username/password does not seem to be valid.\nBye.\n\n";
  exit -1;
}

print "Great, you're in.\n\n";

my $projectName = "ImportTest2 project";
my $projectDesc =
  "This project was created by the ImportTest test case.";
my $projectUser = $session->User();
my $projectGroup;
my $status = 1;
my $age;
my $data;
my $project;
my $project_id;
my @projects;

# See if this project already defined. If not, create it.

@projects = OME::Project->search(name => $projectName);
if (scalar @projects > 0) {
  $project    = $projects[0];     # it exists, retrieve it from the DB
  $project_id = $project->ID();
  $project    =
    $session->Factory()->loadObject("OME::Project", $project_id);
  $status = 0
    unless defined $project;
  $age = "old";
} else {                          # otherwise create it
  print STDERR "- Creating a new project...\n";
  $age          = "new";
  $projectGroup = $projectUser->group()->ID();
  $data         = {
    name        => $projectName,
    description => $projectDesc,
    owner_id    => $projectUser->ID(),
    group_id    => $projectGroup
  };
  $project = $session->Factory()->newObject("OME::Project", $data);
  if (!defined $project) {
    $status = 0;
    print " failed to create new project $projectName.\n";
  } else {
    $project->writeObject();
  }
}

# Die if we don't have a project object at this juncture.
die "Project undefined\n" unless defined $project;

# Now, get a dataset.
# The dataset name on the command line either matches an existing
unlocked dataset owned by the current user,
# or is the name of a new dataset.
# Either way, we must associate the dataset with the current project.

my $datasetName = shift;                  # from @ARGV
my $datasetIter = OME::Dataset->search3(
  name     => $datasetName,
  owner_id => $projectUser->ID(),
  locked   => 'false'
);
my $dataset = $project->addDataset($datasetIter->next())
  if defined $datasetIter;
$dataset = $project->newDataset($datasetName) unless defined $dataset;

# die if we still don't have a dataset object.
die "Dataset undefined\n" unless defined $dataset;
```

```
$session->project($project);
$session->dataset($dataset);
$session->writeObject();
print "- Importing files into $age project '$projectName'... ";
OME::Tasks::ImageTasks::importFiles($session, $dataset, \@ARGV);
print "done.\n";

exit 0;
```

Figure 47 – `OME::Tests::ImportTest` script

```
#!/usr/bin/perl -w
# OME/Tests/AnalysisEngine/ExecuteView.pl

use OME::Session;
use OME::SessionManager;
use OME::AnalysisView;
use OME::Dataset;
use OME::Tasks::AnalysisEngine;
use Term::ReadKey;

print "\nOME Test Case - Execute view\n";
print "----------------------------\n";

if (scalar(@ARGV) != 2) {
  print "Usage:  ExecuteView <view id> <dataset id>\n\n";
  exit -1;
}

my $viewID    = $ARGV[0];
my $datasetID = $ARGV[1];

print "Please login to OME:\n";

print "Username? ";
ReadMode(1);
my $username = ReadLine(0);
chomp($username);

print "Password? ";
ReadMode(2);
my $password = ReadLine(0);
chomp($password);
print "\n";
ReadMode(1);
```

```
my $manager = OME::SessionManager->new();
my $session = $manager->createSession($username, $password);

if (!defined $session) {
  print "That username/password does not seem to be valid.\nBye.\n\n";
  exit -1;
}

print "Great, you're in.\n\n";

my $factory = $session->Factory();
$factory->Debug(0);

my $view    = $factory->loadObject("OME::AnalysisView", $viewID);
my $dataset = $factory->loadObject("OME::Dataset",      $datasetID);

OME::Tasks::AnalysisEngine::executeAnalysisView($session, $view, {},
  $dataset);

1;
```

Figure 48 – `OME::Tests::AnalysisEngine::ExecuteView` script