Extensible Framework for Standards in Functional Genomics

Angel Pizarro[1], Andrew Jones[2], Paul Spellman[3],
Michael Miller[4], Patricia Whetzel[5] and the FuGE working group

June 21, 2006

[1]*Institute for Translational Medicine and Therapeutics, University of Pennsylvania, Philadelphia, USA*
[2]*School of Computer Science, University of Manchester, UK*
[3]*Lawrence Berkeley National Laboratory, University of California*
[4]*Rosetta Inpharmatics*
[5]*Center for Bioinformatics, University of Pennsylvania*

Note: This version of "Extensible Framework for Standards in Functional Genomics" corresponds with FuGE milestone 3 only.

1 Background

Functional genomics can be defined as:

> an area of study aimed at determining the function of genes and the proteins they encode in determining traits, physiology or development of an organism. Generally the term is used for an experimental approach utilizing computational and high-throughput technologies at the level of whole genomes. [1]

Ambitious in breadth and scope, these high-throughput experimental approaches produce volumes of interrelated data, many of which are in completely different native formats. Much worse, all of these data are, for the most part, completely disassociated from the experimental protocol information that led to the data.

Recent data transfer standards and reporting recommendations have alleviated some of these concerns for a subset of technologies [2, 3, 4, 5], but only for those technologies. In each case, the standards body chose to model shared aspects of functional genomics experiments independently, using different vocabularies and levels of detail. The result is that each standard represents semantically equivalent information in syntactically incompatible ways.

When proposals for data standards for microarray (MAGE) [5] and proteomics (PEDRo) [2] technologies emerged, attempts were made to merge these models into a single standard, with varying degrees of success [6, 7]. The conclusion from these efforts is that a comprehensive standard for microarrays and proteomics would be large and complex, hindering adoption by the community and vendors. Furthermore, the standard would still leave much of the functional genomics domain unaccounted for.

Instead of trying to provide a complete stand-alone solution, we have attempted to model the common aspects of functional genomics experiments, such as sample preparation, contact information, and protocols, and tackle the problem of representing different functional genomics technologies in two ways.

First, we provide general structures for referencing external data formats while capturing the meta data that gives the format a context within an entire investigation. The external files can either be open standards, or proprietary formats. Such a mechanism allows us to leverage the currently available formats, and in addition allows the collation and annotation of functional genomics experiments that cross technologies. It also allows one to augment a standard that does not provide as rich a model for defining sample preparation.

Second, these same reference points act as placeholders for defining extensions to the model that are specific to a functional genomics technology. Such extended models gain a rich set of functionality from the core model, as well as having a structural basis shared with other standards, enabling future integration of data.

The Functional Genomics Experiment (`FuGE`) object model seeks to provide a framework for standards development efforts in the life science domain. It is subdivided into two main areas, one providing the basic functionality needed for data standards in general (`FuGE.Common`), and the other providing specific modules tailored for biological data standards (`FuGE.Bio`), such as structures to capture details of experimental design and annotation of biological samples.

In the following section, there are descriptions of these two main areas of the `FuGE` model. This document does not cover all aspects in depth but instead aims to provide a solid introduction to the main concepts. We refer to the developer documentation for further reading [8]. Since `FuGE` is specified using the Unified Modeling Language (UML) [9], it is assumed that the reader is familiar with some basic object oriented concepts, such as objects, classes, packages and inheritance, although this is not essential.

2 Results

2.1 The `FuGE` Object Model

The `FuGE` object model seeks to provide a framework to assist the development of other biological data standards. It does so by providing a UML abstract model of the concepts common to most functional genomics experiments. Model-Driven-Architecture (MDA) approaches are then used to generate platform-specific im-

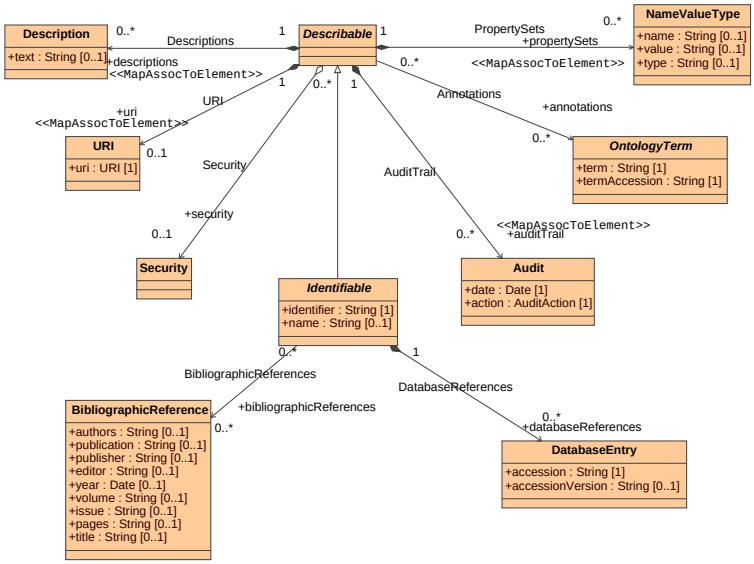

Figure 1: The main set of base classes in `FuGE` . All subsequent classes inherit from either one or both of the two abstract classes `Describable` and `Identifiable`.

plementations of the model, such as an XML schema, a relational database schema, and software libraries. The XML Schema is of primary interest for standardisation, since it specifies the data transfer format in XML and allows data to be validated against the standard.

The `FuGE.Common` package provides a solid foundation for data standards from any domain as it does not contain any information specific to biological investigation. `FuGE.Common` provides mechanisms for referencing external databases, controlled vocabularies, auditing and security, which are described in the rest of Section 2.1

Every object of the `FuGE` model is a descendent of either one or both of two base classes (Figure 1), which are themselves arranged in a hierarchy. The first and most basic class, `Describable`, provides functionality for aiding in-house management of the format, enabling the annotation of objects with plain text or with controlled vocabulary terms. Every object in `FuGE` is a subclass of `Describable`, which allows a URI to be specified for the object and audit information to be attached, such as managing changes to the document (when, who and what change has been made, Figure 2). `Describable` also provides a mechanism for specifying the security settings of the object, such as groups of users that have read or write access. Much of the support mechanisms for defining a rich standard are represented by the associations from this class.

The other base class, `Identifiable`, inherits from `Describable` (thus gaining all of its functionality) and adds a referencing mechanism. As the name

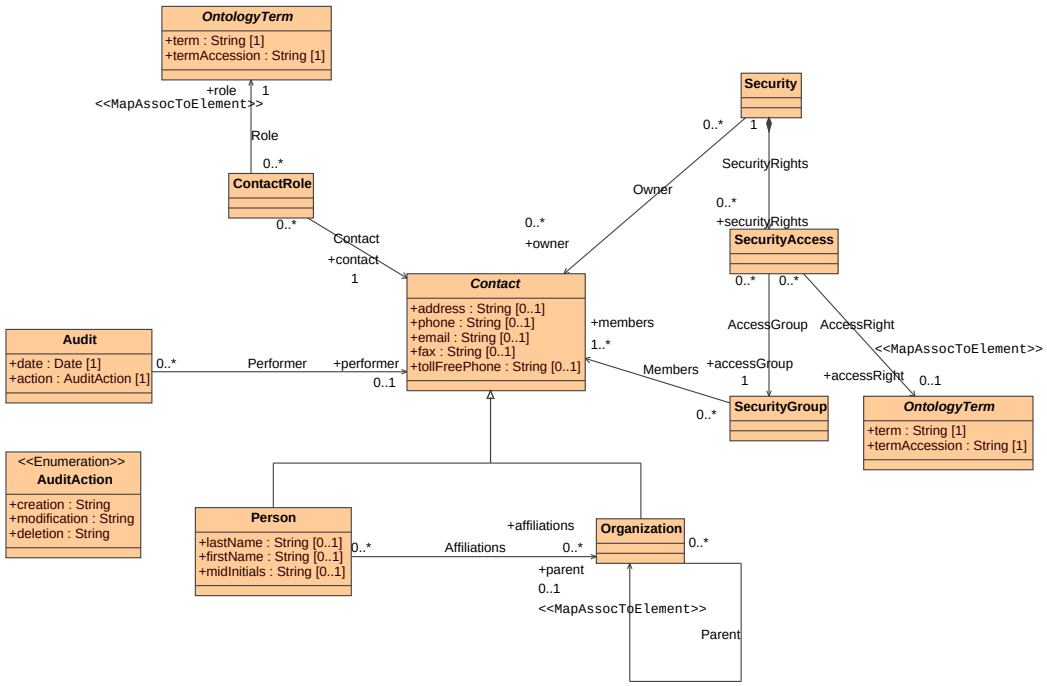

Figure 2: The `Audit` package allows an audit trail and security settings to be attached to objects.

`Identifiable` suggests, classes which inherit from it are able to be referenced by other classes. This is done by virtue of the `identifier` attribute. The `identifier` attribute is understood to be a globally unique string that resolves a particular instance of the object. The "globally unique" restriction implies that there is exactly one and only one object with a particular identifier, and that this identifier must always reference only that particular object. The `name` attribute stores a human readable name for the object that need not be unique.

When using `FuGE` within internal pipelines the definition of "global" can be relaxed to mean "system-wide", since the identification string need only be unique within that system. The globally unique issue arises only if the data are to be published to central repositories or shared among collaborators. When this situation arises, one can always qualify internal identifiers with a prefix such as the name of the institution. Another alternative is the life science identifier (LSID) proposal [10]. In the Web services world, identifiers usually take the form of URLs.

The use of globally unique identifiers provide several practical advantages, such as compact representation within file formats. For example, entities such as biological samples with rich sets of annotations can be referred to using their respective identifier. The underlying data store or data transfer standard can then save space if the object has been used in multiple settings by utilizing this string instead of repeating the information. The `Identifiable` objects also allow for federated data stores, by enabling large and complex experiments to be broken down into more easily manageable components while not losing the relationships that exist between each piece.

2.1.1 Referencing Outside Sources of Information

Descendants of `Identifiable` are also able to provide references to outside sources of information. `Identifiable` objects reference these external sources via the `DatabaseEntry` class. A `DatabaseEntry` contains a reference to the source of the entry (the `Database`), as well as the string, stored in the `accession` attribute, used within that source to identify uniquely the concept being referenced by the `FuGE` object. The use of the name "accession" highlights the distinction between `FuGE` identifiers and another system's identification scheme. These outside sources have no knowledge of `FuGE`, or the way in which `FuGE` defines identification strings. They also do not constrain their internal identifiers to be globally unique with respect to other sources. However, this mechanism could also be used for referencing a `FuGE` object stored in a database.

`FuGE` can be described as a reference model, in that it is designed not to recreate an external system's representation of a concept. Any information referenced from an external source must be interpreted within that system, not `FuGE`. This is especially important when referencing terms from an ontology. Ontologies are developed by specific user communities for their particular domain. The terms within an ontology have interdependent relationships that define the full scope of each term. When evaluating whether an association from `FuGE` object to an ontology term is semantically correct, one would need

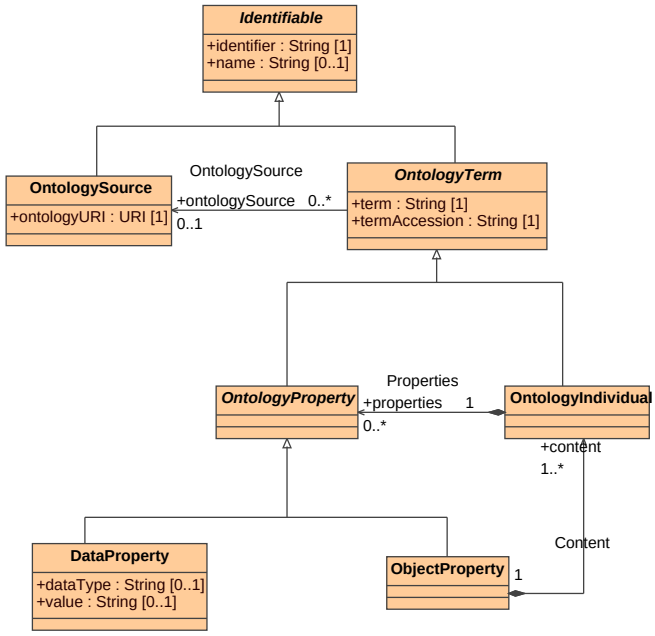

Figure 3: `OntologyTerm` represents instance terms from outside ontological sources.

to put the term in its proper context. Specifically, one would need to evaluate whether the annotation was correct given the definition of the term within its source system and the association to other terms in the ontology that may or may not have been referenced by the `FuGE` object.

It is vital for understanding `FuGE` that inheritance from these two base classes is ubiquitous to all objects. When reading the rest of this document, it is important to remember that all objects have the `Describable` functionality and most have the `Identifiable` functionality, but that these attributes and associations are not shown on the UML diagrams. We refer the reader to the `FuGE` developer's documentation [8] to determine whether a class is `Identifiable`.

2.1.2 Referencing external vocabularies and ontological sources

The use of standard vocabularies and ontologies for annotation of experimental data is vital to the goal of combining data from different functional genomics technologies. `FuGE` provides a rich model for referencing terms from arbitrary external ontology sources. The `OntologyIndividual` class provides a mechanism for representing ontological classes, instances of classes and terms from simple controlled vocabularies (Figure 3). The source of a term can be specified by the association to the `OntologySource` class. The `term` attribute stores the term itself (or the name of the ontology class) and `termAccession` stores the

identifier or accession assigned to the ontology term *within the source ontology*. If no such accession is available, the ontology term itself is the accession (by default), such that the values in the `term` and `termAccession` attributes are identical. The inherited `identifier` attribute stores a unique identifier for the instance of the term used within `FuGE`.

In an ontology, classes can be related to each other through associations. Such associations are modelled in `FuGE` by the abstract class `OntologyProperty`. If the association is between two terms, `ObjectProperty` models the association from the parent concept to the child that is also modelled by `OntologyIndividual`. The `ObjectProperty` class is required, rather than simply having a self-association on `OntologyIndividual`, because the association itself may be a named concept in the ontology with a definition that should be referenced. A second type of association exists in ontologies, modelled by `DataProperty`, which specifies that a value can be entered by the user of the ontology to complete the concept. In various parts of the `FuGE` model there are defined associations to `OntologyTerm` for capturing ontological information. The associations are to `OntologyTerm` rather than to `OntologyIndividual` to allow the user to import an ontology class or instance (`OntologyTerm`), an ontology property (`OntologyProperty`) or a data property (`DataProperty`). However, it is expected that in the majority of instances, the associations will be used to associate with ontology classes or instances, using `OntologyIndividual`. The examples below demonstrates the use of these classes within a practical setting.

A simple controlled vocabulary term (the species name *Mus musculus*) can be represented by this structure:

```
<OntologyIndividual identifier="FuGE.OI.1001" term="Mus musculus"
 termAccession="10090" OntologySource_ref="exp01:FuGO"/>
 ...
 <OntologySource ontologyURI="fugo.sf.net" identifier="exp01:FuGO"/>
```

The following example demonstrates the representation of a more complex concept. In this case, the concept ("4 hours") is modelled as a measurement with a value and a unit.

```
<OntologyIndividual term="Measurement" termAccession="FuGO:0123" identifier="exp01:OI1"
OntologySource_ref="exp01:OS:FuGO">
    <ObjectProperty term="has_unit" termAccession="FuGO:0124" identifier="exp01:OP1">
        <OntologyIndividual term="hours" termAccession="FuGO:0456" identifier="exp01:OI2"/>
    </ObjectProperty>
    <DataProperty term="has_value" termAccession="FuGO:0056"
     identifier="exp01:DP1" value="4"/>
</OntologyIndividual>

<OntologySource ontologyURI="fugo.sf.net" identifier="exp01:OS:FuGO"/>
```

2.1.3 Protocols and Workflows

A protocol in functional genomics usually consists of a procedure described by a set of ordered actions. The procedure may have a set of inputs, parameters, and

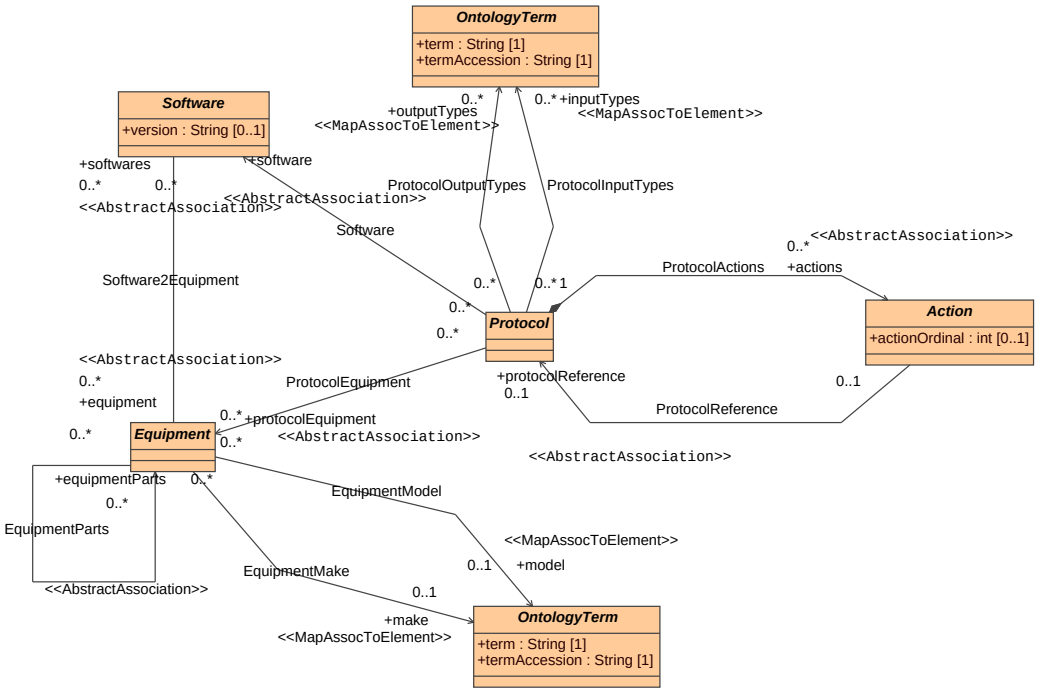

Figure 4: The package representing the abstract structure of all `FuGE` protocols in functional genomics experiments.

produces a set of outputs. Complex protocols can be broken down into a set of simpler protocols that may or may not have their own specific parameters, inputs and outputs. The `FuGE.Common.Protocol` package has structures to represent these requirements.

The `FuGE Protocol` package is separated into two parts: i) providing an abstract representation of how a protocol should be structured, which can be extended for modular formats and ii) providing non-abstract classes that can be used without extension. The abstract classes `Protocol`, `Action`, `Equipment`, `Software` and `Parameter` fall into category i), denoted by the class name shown in italic. These classes cannot be instantiated as they are, because it is intended they should be extended by subclassing to create specific modules. In addition to the abstract classes in the `Protocol` package, a set of non-abstract classes are provided in `FuGE` that can be used without extension, called `GenericProtocol`, `GenericAction`, `GenericEquipment`, `GenericSoftware` and `GenericParameter` (Figure 5). These classes can be instantiated with user entered text and ontology terms to capture the details of the protocol as described below.

The `GenericProtocol` class can be used without extension to capture details of laboratory procedures and protocols (Figure 5). The entire text of the pro-

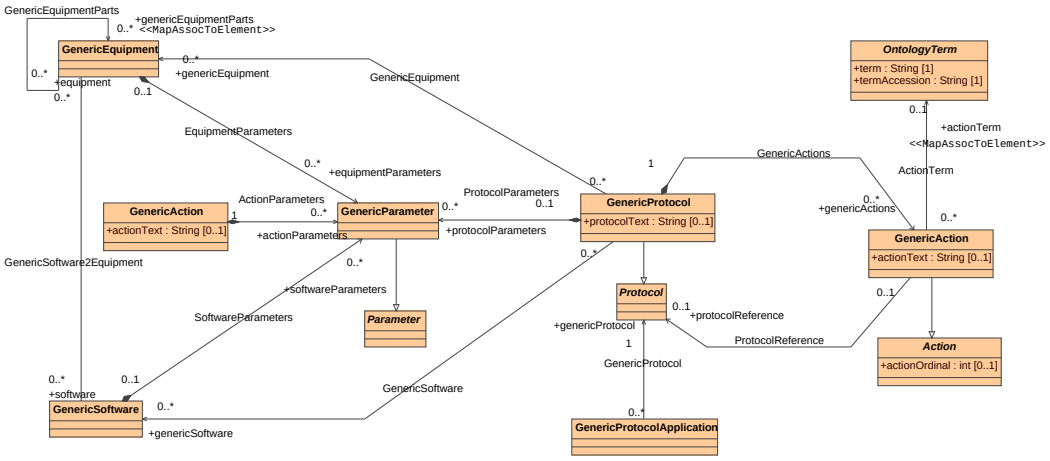

Figure 5: The package representing definitions of protocols that can be used without extension. `GenericProtocol`s have associations to `GenericEquipment`, `GenericSoftware` and `GenericParameter`s. `GenericProtocol`s also contain a set of ordered `GenericAction`s which may exclusively define text for the action, reference an ontology term defining that action, or reference a sub-protocol (for representing complex protocols).

tocol can be supplied in the `protocolText` attribute or the `GenericProtocol` can contain an ordered set of `GenericAction`s. These `GenericAction`s are understood to be atomic instructions such as "wait 10 minutes". The order of an `GenericAction` is determined by the `actionOrdinal` attribute (inherited from `Action`). `GenericAction`s with duplicate ordinals are understood to be conducted in parallel. A `GenericAction` can be specified either by free text entered by the user or it can reference an ontology containing a standard action term, such as the MGED Ontology (MO, [11]). `GenericProtocol` has associations inherited from `Protocol` that can be used to specify the intended input and output types using ontology terms, such as certain material or data types.

Complex procedures, or workflows, in laboratories are usually composed of a set of simpler protocols that are to be completed in some particular order. Since the `GenericProtocol` class is `Identifiable`, defining these types of complex workflows is straightforward, by creating new `GenericProtocol`s that reference the simpler ones. Sub-protocols are referenced using the `GenericAction` class, thus inheriting the same ordering scheme and allowing combinations of simple instructions among the references to sub-protocols (Figure 6 shows a simplified diagram of the `FuGE` representation of protocols).

The `GenericProtocol` class can have a set of defined parameters, modeled by associations to the `GenericParameter` class (Figure 5). Each `GenericParameter` has associations inherited from `Parameter` (Figure 7) allowing the unit and data type (Boolean, string, integer) to be specified via

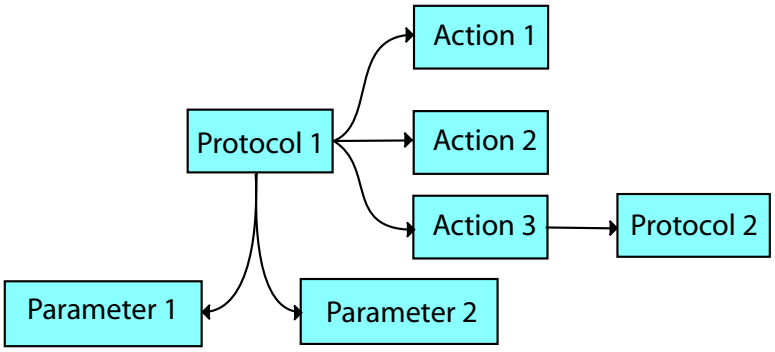

Figure 6: A summary of the structure of a protocol in `FuGE` that has two parameters and three actions, one of which is a reference to a further protocol.

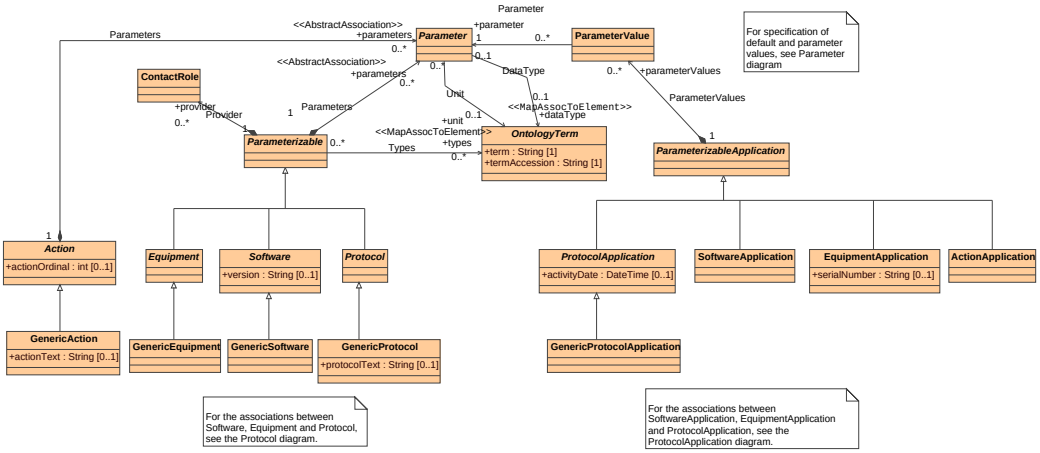

Figure 7: The `Protocol` package contains classes that can provide parameters with default and instance values.

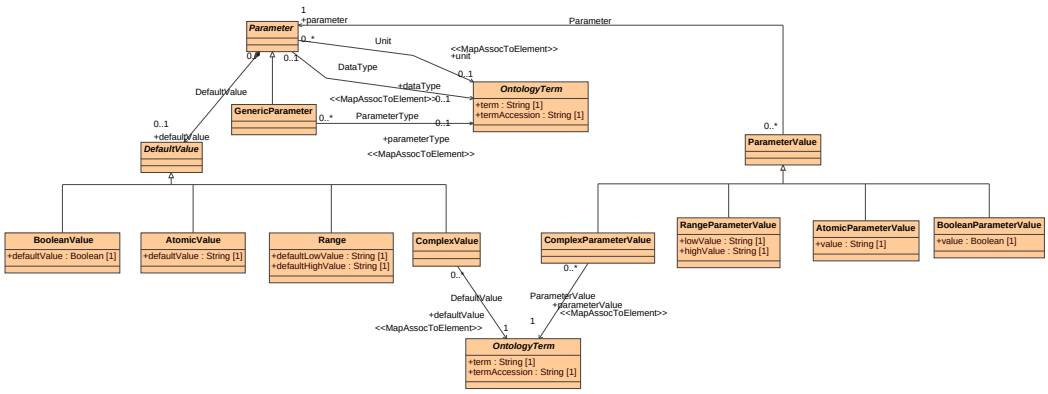

Figure 8: The values for `Parameter`s can be supplied with default and runtime values using the classes provided.

associations to an outside ontology. `GenericParameter` can also be annotated with an ontology term to describe its type. The parameter's default value is supplied using one of the provided slots for a `BooleanValue`, `RangeValue`, `AtomicValue` or `ComplexValue` (Figure 8). The actual values used when a protocol is implemented are recorded by `GenericProtocolApplication`, as described below. A `GenericProtocol` can be associated with `GenericEquipment` and `GenericSoftware`, which can also have a set of parameters defined with controlled vocabulary terms. There is an association between `GenericEquipment` and `GenericSoftware` to record that a piece of software is intrinsically linked to an instrument, which may be important if a `GenericProtocol` is associated with numerous instances of `GenericEquipment` and `GenericSoftware`.

Frequently, deviations from the default parameter values of a protocol must be recorded for a particular instance. Parameters without default values must also be recorded. It would be highly inefficient to redefine a complete protocol for every single deviation that occurs. `FuGE` handles this by separating *the definition of a protocol* from the object that annotates *the occurrence of a protocol*. The `GenericProtocolApplication` class references the parent `GenericProtocol` class and provides a place for annotating deviations and runtime parameter values, inherited from `ProtocolApplication` (Figure 9). It also provides mechanisms for recording the operator of the procedure and the date performed, both criteria that have been shown to be important when identifying and accounting for confounding factors in data analysis [12]. There are associations from `GenericProtocolApplication` (inherited from `ProtocolApplication`) to `EquipmentApplication` and `SoftwareApplication`, which record the parameter values used in conjunction with the software or equipment (Figure 9).

The division of the `Protocol` package into the abstract and non-abstract classes is to provide a simple framework for creating models of protocols, equipment, software, parameters and actions that are specific to a certain technol-

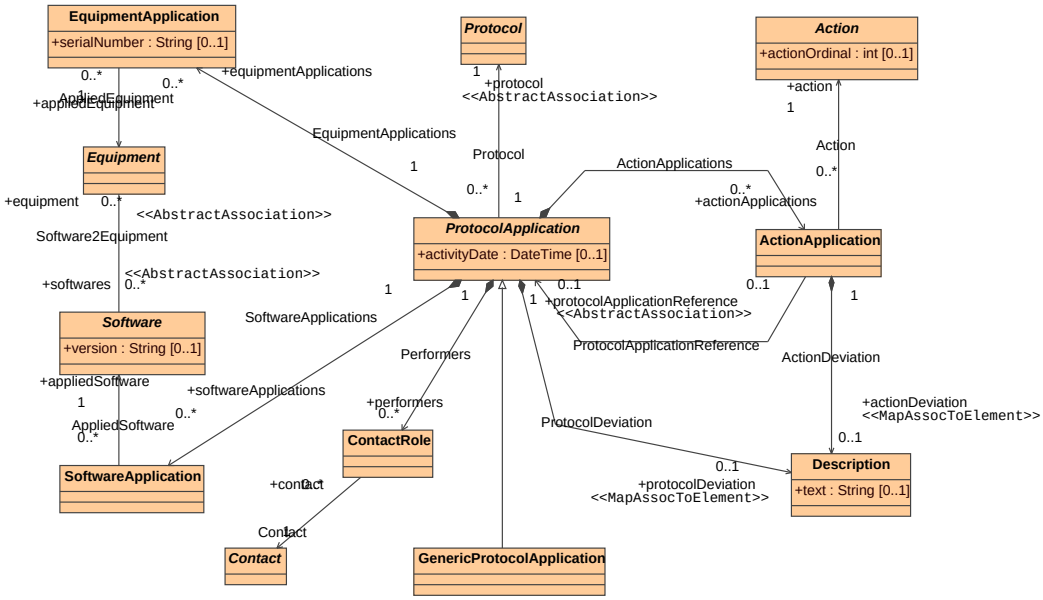

Figure 9: `ProtocolApplication` records an instance of a protocol.

ogy. For example, a developer may wish to create a model in UML of image acquisition that has the relevant steps, parameters and equipment without relying on external ontologies to supply these details. In this case, the `Protocol` class could be extended with `ImageAcquisition`, `Equipment` extended with `Scanner`, and `Action` extended with `Calibration` and `Scan` to represent specific steps within the protocol that must be reported. The associations between several of the classes in the `Protocol` package are also abstract (denoted by `<<abstract association>>` on Figure 4). These associations should also be extended using a UML inheritance association, which allows a `FuGE` compliant parser to process extension of the format. A guide for how to build extensions to `FuGE` is provided on the website (for milestone 2 version, http://fuge.sourceforge.net/dev/ExtendingM2.php, milestone 3 version to come).

2.2 Modules for Representation of Biological Data Standards

2.2.1 The treatment of materials

The biological samples and the types of procedures that act on them are modeled in the `FuGE.Bio.Material` package (Figure 10). The model seeks to be simple, robust and flexible for representing these concepts. Simple and flexible object models allow for the representation of a wide variety of concepts

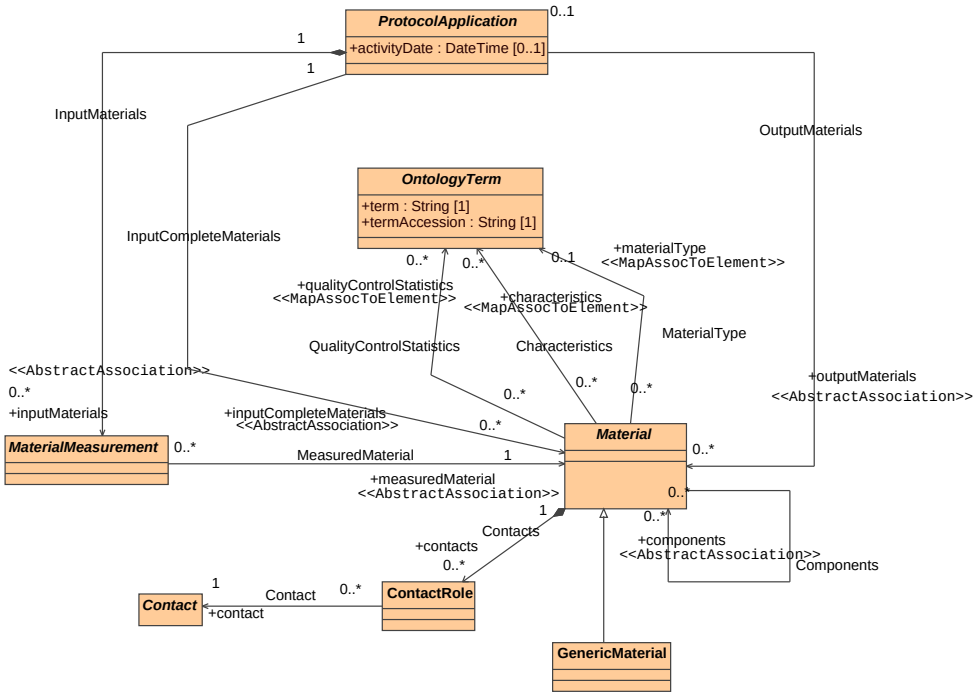

Figure 10: The `Material` package in `FuGE` .

using clear and consistent semantics. All biological materials from whole organisms to reagents are represented by the class `GenericMaterial`, which extends from an abstract `Material` class. The representation is robust since any `GenericMaterial` can have a set of annotations from external sources that can completely define its attributes and behavior. `Material` has three named associations for capturing ontological information: `MaterialType`, `Characteristics` and `QualityControlStatistics`. This leaves `FuGE` free from defining the content and constraints of objects in a domain, instead shifting the focus on to the domain's knowledge experts who can build ontologies that capture complete definitions of the required concepts. However, as for `Protocol`, two classes are provided: the abstract `Material` class and the non-abstract `GenericMaterial` class. If an extension developer wishes to extend the model in UML rather than by developing ontologies, the `Material` class can be extended by subclassing.

`GenericProtocolApplication` can be used to define actions that take a set of input `Materials` (either `GenericMaterials` or a subclasses of `Material` defined in an extension) to produce a set of new output `Materials` (Figure 10). The instance of `GenericProtocolApplication` can reference the corresponding `GenericProtocol` definition (or a relevant extension of `Protocol`). The input `Material` to a `GenericProtocolApplication` can be quantified using

the GenericMaterialMeasurement class. Any GenericMaterialMeasurement that does not have a defined measurement assumes that the treatment used all of the Material available. Since Materials are Identifiable, the history of biological samples and their descendents can be traced. This can be done across treatments, analyses and experiments. Starting Materials (sources of biological material) can be determined as they are not specified as the output from any treatment. The Identifiable functionality of Material will enable collating and combining of data from different technologies that have been applied to the same sample.

It is also worth noting that a GenericProtocolApplication can take Data as input and produce Data as output, primarily for describing the acquisition or transformation of data as described below. However, there are also cases where a treatment involving Materials can produce both new Material and Data. One notable example is a liquid chromatography (LC) separation that splits a complex protein mixture by size, molecular charge or hydrophobicity. An LC process typically produces not only the output fractions, but also a chromatogram of the protein or peptide abundance across all fractions as a function of elution time, which could be viewed as a data item. Furthermore, there are examples of treatments on Materials that can take Data as input. An example being the use of a robotic spot picker to extract plugs from two dimensional gels, which take a list of spot coordinates (a data item) as input. This could be encoded in FuGE as a GenericProtocolApplication that takes a instance of Material (the gel) and an instance of Data (the spot coordinates) as input, producing an output of multiple Material objects corresponding to each gel spot extracted.

The abstract ProtocolApplication class provides the framework for extending to build more specific models for certain techniques. For example, an extension could be developed for liquid chromatography, LCApplication, where LCApplication references GenericMaterial for the input, but produces as output a Fraction (defined as a subclass of Material) rather than a GenericMaterial and a Chromatogram (defined as a subclass of Data). This allows the specifics of a workflow, in terms of inputs and outputs to processes, to be encoded in a UML extension rather than by defining ontologies.

2.2.2 The production or transformation of Data

Acquisitions of data are also represented by the GenericProtocolApplication class, which take instances of Material as input, producing sets of raw data, represented by the Data class (Figure 11). Once data are generated, they often must be transformed into other formats, combined, or manipulated to enable a coherent conclusion to be derived. Thus a further usage of GenericProtocolApplication can be made, in which input Data are transformed to produce new sets of Data. Examples of a transformation are normalizations, file format changes, and data reduction. Analysis pipelines can be encoded by combining a series of GenericProtocols within a hierarchical structure in which the top-level GenericProtocol represents the entire pipeline

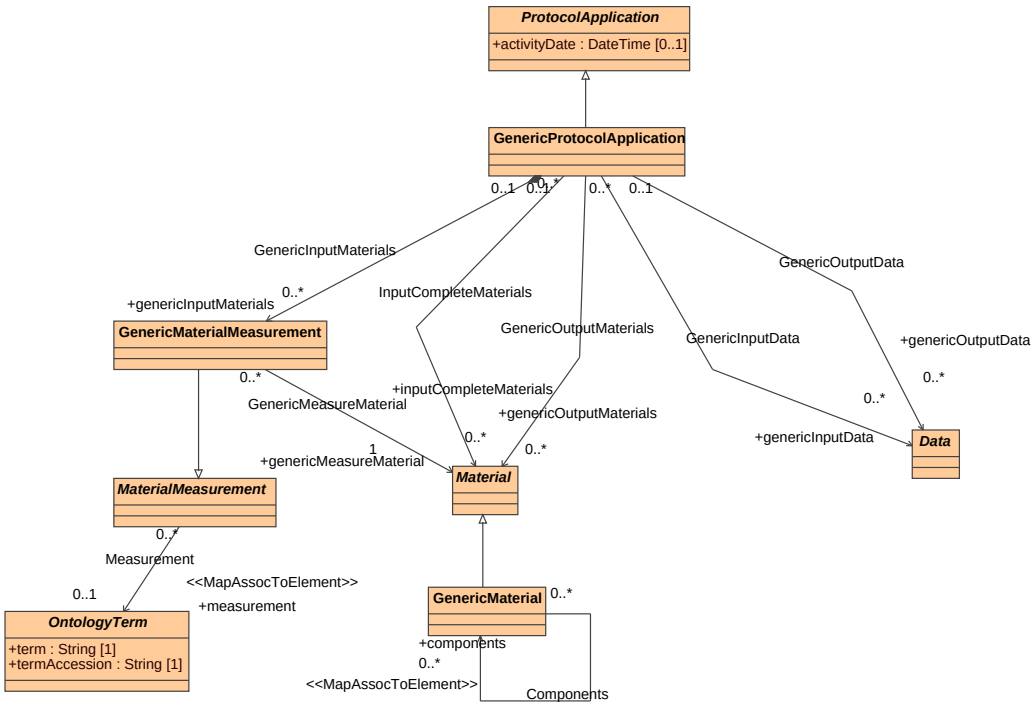

Figure 11: `GenericProtocolApplication` can take `Material` and/or `Data` as input, producing `Material` and `Data` as output.

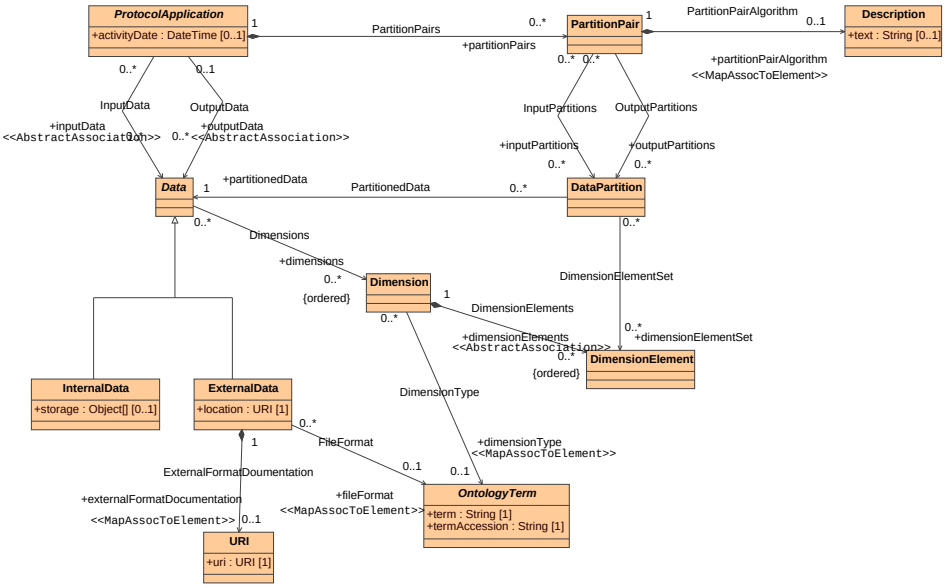

Figure 12: The `Data` package in `FuGE` .

and sub-protocols represent individual transformations.

2.2.3 Experimental Data

So far we have talked about the `Data` as an abstract concept. Data acquired from most high-throughput technologies can be represented as a matrix of values. Even images can be considered a two dimensional matrix of pixel hue and intensity utilizing a standard (x,y) coordinate system for the dimensions of the matrix. The `FuGE Data` object has been designed to enable representing n-dimensional data matrices (Figure 12).

The definition of `Data` axes, or `Dimension`s, are re-usable across `Data` instances. This saves space when the `Data` instances all have a very regular structure, such as multiple microarray results done on a single type of chip. The `Dimension`s of the data acquired from the microarray are defined once, and all `Data` items refer to these same `Dimensions`.

Reusable dimensions also separates the description of the data format from the actual data instances, allowing software manufacturers to publish the specifications on their data formats using a standard (and software friendly) mechanism. The disassociation of data instances from the descriptive attributes has worked well for atmospheric researchers [13, 14].

External data formats that are not directly represented in `FuGE` can be referenced in the file system using the `ExternalData` object's `URI location` association. There is additionally a named association to `URI` for storing a definition

or documentation about the external file format used. This mechanism allows for the procedures and input materials from which the data was produced to be captured. Furthermore, FuGE can specify transformations of data represented in external formats in a "black box" manner using the GenericProtocol class without requiring knowledge of the exact format of the data files. The downside of representing data in external formats is that additional processing software is required (alongside a FuGE parser) to read the data or query data files. However, for most cases this will work in practice because parsers for data files can usually be developed much quicker than a data standard can develop. The result is that FuGE allows the meta-data resulting from an investigation to be captured without preempting the formats of data files that will exist for each functional genomics technique.

The Data package also has concepts for relating particular subsets of data together (Figure 12). The DataPartition class can be used to describe a particular subset of a multidimensional data set by referencing certain DimensionElements and the storage matrix or external file in which the data are stored. The PartitionPair class references two DataPartitions, corresponding to a subset of the input Data to and output Data from the ProtocolApplication. The instance of PartitionPair can be used for multiple purposes, such as for describing "supporting evidence" where certain results in the output Data are dependent only on certain parts of the input Data set. The algorithm that relates the input DataPartition to the output DataPartition can be captured using the association to Description.

2.2.4 Investigational Designs

Many journals have started to require a minimum set of reporting guidelines for the microarray field, in the form of the MIAME requirements list [4]. It is highly likely that the same will hold true for proteomics data in the near future as the MIAPE standard matures [15].

Common to both of these guidelines is a section of reporting requirements that deal with the overall goal and design of the investigation. Example requirements for MIAME include

1. What is the high-level description of the motivation for the production of such an experiment?
2. What were the experimental factors (variables) of interest?
3. What portions of the data apply to which factors?

These requirements are met in the FuGE.Bio.Investigation package (Figure 13). The Investigation class captures the name and description of the entire investigation, with an association to OntologyTerm for the type of investigation design. Suitable terms from MO include: "methodological design", "biological property" or "perturbational design". Investigation has an association to Material representing the important sources of material, as determined by

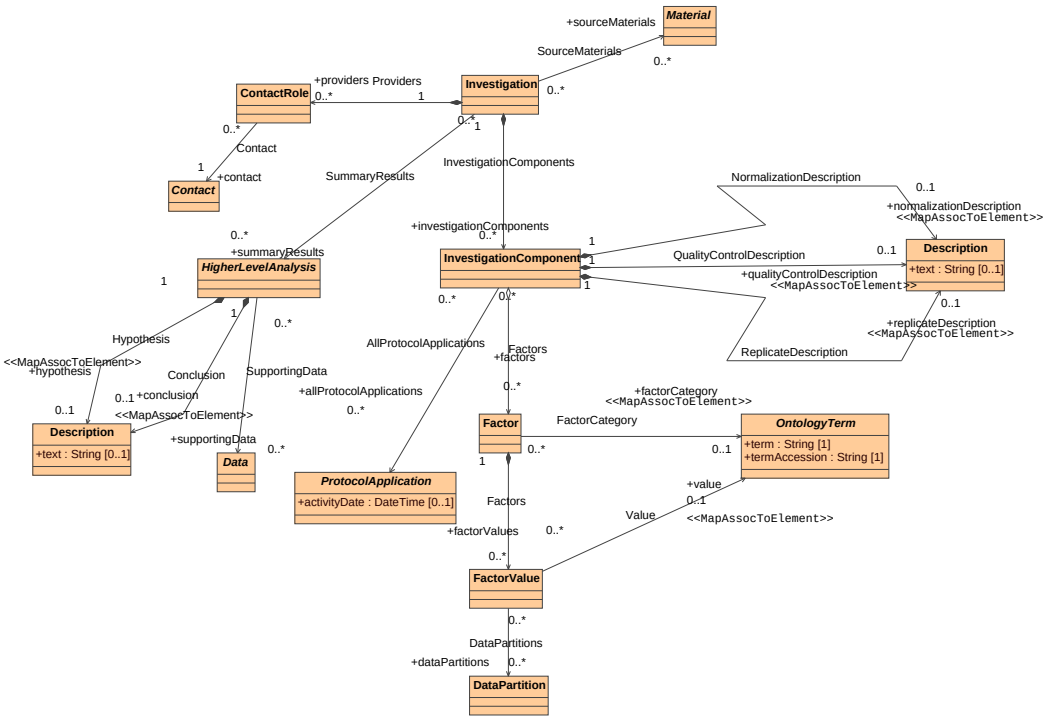

Figure 13: The `Investigation` package in `FuGE` .

the investigator, for the purpose of providing a summary in the `Investigation` package which can be queried. The `Material` could be of any type, including a single organism, a population, a tissue, a cell culture and so on. An instance of `InvestigationComponent` represents a single functional genomics technique used within this investigation and it allows the user to specify the number of replicates performed, the normalization strategy and quality control. Such properties are given prominence in the MIAME guidelines and are likely to feature in MIAPE as they provide a brief overview of the quality of a study. There is also an association from `InvestigationComponent` to `OntologyTerm` to capture the design with respect to the particular technology, for example "dye_swap". The principal comparators in an investigation are modeled by `Factor`, such a "time course", "genetic difference", "environmental factor" and so on. `Factor`s can, but need not, be shared across different `InvestigationComponent`s; for instance certain technologies might be used to measure certain factors but not others. The actual values for factors are stored in `FactorValue` and the association to `OntologyTerm` (for measurements or standard terms, such as a gene name from a knock out experiment). `FactorValue`s have an association to `DimensionElement` referencing the set of data values that correspond to that factor.

There is an association from `Investigation` to `HigherLevelAnalysis` for representing analyses that are specific to a technology, and the `Data` on which such analyses are based. It is intended that `HigherLevelAnalysis` will be extended by developers of standards for single domains, one example is an analysis of microarray data for genotyping.

2.2.5 Biological molecules

The majority of data formats in functional genomics have some concept of the computational representation of a molecule (as opposed to that which physically exists) that has been identified or measured in the experiment. The `FuGE.Bio.ConceptualMolecule` package contains a simple model of biological sequences that could be used for genomes, RNA reporters on microarrays or protein sequences identified in proteomics (Figure 14). The package also acts as a placeholder for extension with other models such as metabolites or chemical compounds.

3 Discussion and Conclusions

The previous sections outlined the major components that are required to understand the `FuGE` proposal. We saw that basic functionality, such as attaching descriptions and audit information, are gained by inheritance from the `Describable` class. We also saw that classes which are `Identifiable` can be referenced in a standard way by other classes in the model, and they can reference outside sources of information. The model of protocols allows rich descriptions of parameters, ordered sets of actions, and references to other protocols for rep-

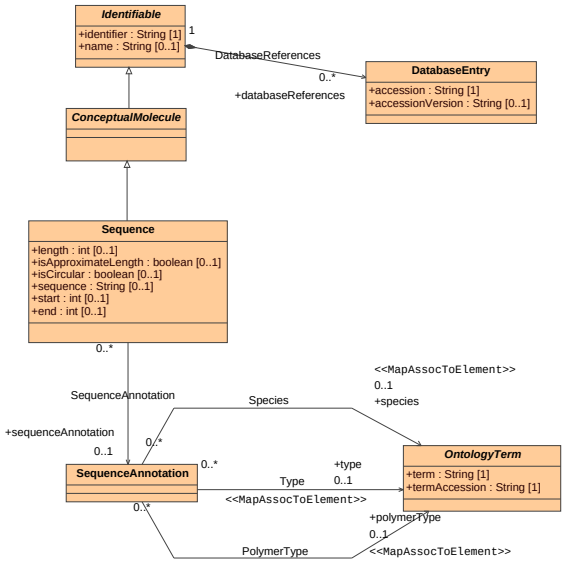

Figure 14: The `ConceptualMolecule` package.

resenting complex procedures. Deviations and observed values from completed protocols can be captured using the `ProtocolApplication` class.

The `FuGE.Bio` package takes advantage of these mechanisms to provide a solid foundation for data standards focused on representing functional genomics experiments. It provides facilities to describe bench procedures and experimental assays. It allows for cataloging of data acquired from biological samples and further analysis of those data. Lastly, it provides constructs for outlining the high-level goals of an investigation.

It is important to stress that `FuGE` is not an effort to unify all functional genomics data formats into one standard. Such an attempt would require a vastly complex (or overly generalised) proposal and agreement from a number of standards bodies, which is unlikely to be achievable. Instead, `FuGE` aims to provide a model of the components that are common to all functional genomics techniques, which can be used by standards developers to produce usable formats. We believe that there are substantial advantages to using `FuGE`, not least that if it receives wide spread support, integration of data produced by different techniques will be facilitated. At present, MGED and PSI are committed to using `FuGE` in the development of their next standard formats for microarrays and separation-based proteomics respectively. `FuGE` is also being implemented by several industrial and research organisations (Fred Hutchinson Cancer Research Centre, Rosetta and Genologics). `FuGE` is currently being tested with use cases from the microarray and proteomics domain, and early proposals involving metabolomics are also being discussed. We would like to encourage feedback from producers of functional genomics data and systems developers. The over-

all goal of FuGE is to facilitate the production of data standards that serve the needs of each community and to foster improved compatibility between different formats.

References

[1] RiceCAP Glossary. URL http://www.uark.edu/ua/ricecap/ricecapgloss.htm.

[2] Taylor, C. *et al.* A systematic approach to modeling, capturing, and disseminating proteomics experimental data. *Nature Biotech* **21**, 247–254 (2003).

[3] Orchard, S., Zu, W., Julian, R., Hermjakob, H. & Apweiler, R. Further advances in the development of a data interchange standard for proteomics data. *Proteomics* **3**, 2065–2066 (2003).

[4] Brazma, A. *et al.* Minimum information about a microarray experiment (MIAME) - toward standards for microarray data. *Nat Genet* **29**, 365–371 (2001).

[5] Spellman, P. *et al.* Design and implementation of microarray gene expression markup language (MAGE-ML). *Genome Biology* **3**, research0046.1–research0046.9 (2002). URL http://genomebiology.com/2002/3/9/research/0046.

[6] Jones, A., Hunt, E., Wastling, J. M., Pizarro, A. & Stoeckert Jr., C. J. An Object Model and Database for Functional Genomics. *Bioinformatics* **20**, 1583–1590 (2004).

[7] Xirasagar, S. *et al.* CEBS Object Model for Systems Biology Data, CEBS MAGE SysBio-OM. *Bioinformatics* **20**, 2004–2015 (2004).

[8] FuGE Developer's Documentation. URL http://fuge.sourceforge.net/dev.

[9] OMG - Object management group. URL http://www.omg.org.

[10] Clark, T., Martin, S. & Liefeld, T. Globally distributed object identification for biological knowledgebases. *Briefings in Bioinformatics* **5**, 59–70 (2004).

[11] Stoeckert, C. & Parkinson, H. The MGED ontology: a framework for describing functional genomics experiments. *Comparative and Functional Genomics* **4**, 127–132 (2003).

[12] Downey, T. Using Statistics to Improve the Quality of Genomic and Proteomic Data URL http://www.partek.com/html/pr/Scientific-Computing-Partek.pdf.

[13] HDF5. URL http://hdf.ncsa.uiuc.edu/HDF5/.

[14] Rew, R., Davis, G. & Emmerson, S. NetCDF User's Guide: An Interface for Data Access (1993).

[15] The MIAPE minimum reporting requirement. URL `http://psidev.sourceforge.net/gps/`.

Acknowledgements Many thanks to all the developers of the FuGE object model, especially the MGED and PSI standards working groups. Thanks also goes to the SourceForge.net team. Without their support, many open source projects would not exist.