UNIVERSITY COLLEGE LONDON

A Framework for the Management of Changing Biological Experimentation

by

Ben Tagger

A thesis submitted in partial fulfillment for the
degree of Doctor of Engineering

Department of Computer Science

August 2010

Declaration of Authorship

I, Ben Tagger, declare that this thesis titled, 'A Framework for the Management of Changing Biological Experimentation' and the work presented therein is my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

Everything flows, nothing stands still. (The only constant is change.)

Heraclitus of Ephesus (c.535 BC - 475 BC)

UNIVERSITY COLLEGE LONDON

Abstract

Department of Computer Science

Doctor of Engineering

There is no point expending time and effort developing a model if it is based on data that is out of date. Many models require large amounts of data from a variety of heterogeneous sources. This data is subject to frequent and unannounced changes. It may only be possible to know that data has fallen out of date by reconstructing the model with the new data but this leads to further problems. How and when does the data change and when does the model need to be rebuilt? At best, the model will need to be continually rebuilt in a desperate attempt to remain current. At worst, the model will be producing erroneous results.

The recent advent of automated and semi-automated data-processing and analysis tools in the biological sciences has brought about a rapid expansion of publicly available data. Many problems arise in the attempt to deal with this magnitude of data; some have received more attention than others. One significant problem is that data within these publicly available databases is subject to change in an unannounced and unpredictable manner. Large amounts of complex data from multiple, heterogeneous sources are obtained and integrated using a variety of tools. These data and tools are also subject to frequent change, much like the biological data. Reconciling these changes, coupled with the interdisciplinary nature of *in silico* biological experimentation, presents a significant problem.

We present the **ExperimentBuilder**, an application that records both the current and previous states of an experimental environment. Both the data and metadata about an experiment are recorded. The current and previous versions of each of these experimental components are maintained within the **ExperimentBuilder**. When any one of these components change, the **ExperimentBuilder** estimates not only the impact within that specific experiment, but also traces the impact throughout the entire experimental environment. This is achieved with the use of *keyword profiles*, a heuristic tool for estimating the content of the experimental component. We can compare one experimental component to another regardless of their type and content and build a network of inter-component relationships for the entire environment.

Ultimately, we can present the impact of an update as a complete cost to the entire environment in order to make an informed decision about whether to recalculate our results.

Acknowledgements

I am indebted to my supervisor, Anthony Finkelstein, who has continued to provide support and encouragement in an honest, always-open and constructive manner, whose guidance and optimism has contributed largely to the completion of this research. I would also like to thank my industrial supervisor, Delmiro Fernandez-Reyes, for his provision and help with the case study.

I would also like to acknowledge the continued support from family and friends, in particular mum and dad, and also my beautiful fiancee who has stuck it out during the entire experience, offering encouragement, support and love throughout.

Thank you everyone.

Ben Tagger

Contents

List of Figures

[1]Sourced from http://www.ebi.ac.uk/IPI/FastaFormat.html

Abbreviations

BIND	**B**iomolecular **I**nteraction **N**etwork **D**atabase
CFS	**C**edar **F**ile **S**ystem
COTS	**C**ommercial **O**ff **T**he **S**helf
CVS	**C**oncurrent **V**ersioning **S**ystem
DAS	**D**istributed **A**nnotation **S**ystem
DBMS	**D**ata**B**ase **M**anagement **S**ystem
DBVS	**D**ata**B**ase **V**ersioning **S**ystem
DDBJ	DNA **D**ata **B**ank of **J**apan
DIP	**D**atabase of **I**nteracting **P**roteins
DNA	**D**eoxyribo**N**ucleic **A**cid
DTD	**D**ata **T**ype **D**efinition
EB	**E**xperiment**B**uilder
EBI	**E**uropean **B**ioinformatics **I**nstitute
EMBL	**E**uropean **M**olecular **B**iology **L**aboratory
GCP	**G**enetics **C**omputer **G**roup
GO	**G**ene **O**ntology
HDM	**H**ashed **D**ata **M**odel
HGP	**H**uman **G**enome **P**roject
HPRD	**H**uman **P**rotein **R**eference **D**atabase
INSDC	**I**nternational **N**ucleotide **S**equence **D**atabase **C**ollaboration
KEGG	**K**yoto **E**ncyclopaedia of **G**enes and **G**enomes
MAGIC	**M**ulti-source **A**ssociation of **G**enes by **I**ntegration of **C**lusters
MIAME	**M**inimum **I**nformation **A**bout a **M**icroarray **E**xperiment
MINT	**M**olecular **Int**eractions Database
MSD	**M**acromolecular **S**tructure **D**atabase
NBII	**N**ational **B**iological **I**nformation **I**nfrastructure

NBRF	**N**ational **B**iomedical **R**esearch **F**oundation
NCBI	**N**ational **C**enter for **B**iotechnology **I**nformation
NIAID	**N**ational **I**nstitute of **A**llergy and **I**nfectious **D**isease
NIMR	**N**ational **I**nstitute for **M**edical **R**esearch
NIG	**N**ational **I**nstitute of **G**enetics
NIH	**N**ational **I**nstitute of **H**ealth
OBO	**O**pen **B**iomedical **O**ntologies Library
PDB	**P**rotein **D**ata **B**ank
PFDB	**P**lasmodium **F**alciparum Genome **D**ata**B**ase
PIR	**P**rotein **I**nformation **R**esource
PSD	**P**rotein **S**equence **D**atabase
RCS	**R**evision **C**ontrol **S**ystem
RDF	**R**esource **D**escription **F**ramework
SCCS	**S**ource **C**ode **C**ontrol **S**ystem
SCOP	**S**tructural **C**lassification **O**f **P**roteins
TPA	**T**hird **P**arty **A**nnotation project
UCL	**U**niversity **C**ollege **L**ondon
UML	**U**nified **M**odelling **L**anguage
XML	e**X**tensible **M**arkup **L**anguage

Chapter 1

Introduction

There are many reasons for requiring data, specifically the latest version of the data. In many cases, we have become infatuated with the need for newest data[1]. There is no interest in using old data, especially if the data is *known* to be out of date. But there are many cases where *old* data is just as up-to-date and relevant today as when it was published. But how can we know when data becomes out of date? Sometimes people are kind enough to tell us, either explicitly or by supplying us with a new edition. Normally, they provide a updated version and leave us to find it. Occasionally, they are even less generous and change their original data without informing us or making any announcement at all. This can cause significant problems especially if we have already used the original data and, unbeknown to us, our work has become out of date.

So, how can we deal with these unannounced changes? We can regularly update our work with a fresh edition irrespective of any changes that may or may not exist in the data. Many systems today employ a 'constant refresh' or *dynamic* methodology, retrieving the latest data as per user request. But this can be very wasteful, especially if the data remains unchanged for long periods of time. And if we have used the data in complex and time-intensive ways, the matter worsens. We may have already conducted a string of experimentation, each based on the one before and ultimately leading back to the original data. We can not simply update the data on a whim, but we must also make sure that our data is valid[2]; a string of experiments, no matter how complex or cumbersome to repeat, is useless if invalidated. There is a fine balance to be achieved when updating an experimental environment with dangers on both sides. Erring on the

[1]One has only to witness the sums of money paid by financial institutions to locate themselves ever closer, both logically and physically, to city stock exchange hubs to get market information that bit quicker.

[2]We refer to data as being invalid with respect to the particular hypothesis or request on the data. It is, perhaps, more accurate to refer to the results and/or model as out-of-date or inconsistent with an available, but as far unused, experimental resource.

side of caution, you may spend all your time needlessly updating but ignoring these changes may cause your model and your results to become out of date.

Clearly, there are occasions where it is wise to update and those when it is not. It depends on the nature of the change and how much this change will affect your results. Unfortunately, it is often difficult to identify this effect without re-running your experiments and actually measuring it. An experienced scientist may have a feel for how a datasource change may affect their results but this is likely to be inaccurate, difficult for most, and nearly impossible for all but the simplest of experimental environments.

Our research aims to reduce this area of divination, giving the experimenter a real-world cost of updating their experiments based on a comprehensive knowledge of the change that has occurred based on a complete analysis of the entire experimental environment.

During this chapter, we describe our motivation for this research. We describe arguably the single most important contribution to genomic and proteomic biology of the last 30 years, namely the **Human Genome Project** (**HGP**). We briefly describe the history of the **HGP**, illustrating some of the problems arising from the project that finally lead to our research question. We conclude the introduction chapter with a concise description of our contribution to the research question posed below. The second chapter describes the key areas of identified related material, detailing their importance and significance to our own research. Chapter 3 provides a detailed analysis of a test case, identified as requiring change management. We present the **Hashed Data Model** in chapter 4, our first approach at tracking changes in biological datasets. Chapters 5 and 6 describe our change management framework, encapsulated in the **ExperimentBuilder** and the various tracking algorithms that we have created and implemented in order to facilitate change and impact tracking throughout an entire experimental environment. Chapter 7 describes the evaluation of our solutions to the research question, both the **HDM** and the **ExperimentBuilder**, and our conclusions are presented in chapter 8.

1.1 Motivation

Since the presentation of the double helical structure of DNA by Watson and Crick in 1953, the pace of related scientific and biological research has increased at an exponential and often alarming pace. Following this cracking of the genetic code, the fields of genomics and proteomics were born and quickly established as independent but related disciplines. During the 1970-1980s, Fred Sanger cemented the idea of genomics by sequencing the first complete genome of a single virus. Since that milestone, his group has established many techniques for sequencing, genome mapping, data storage and bioinformatic analyses.

From there on, the Human Genome Project **HGP** set about identifying and mapping the 20-25,000 genes of the human genome. The project ran from 1990 to 2000 whereupon it released a working draft of the entire human genome. The **HGP** was a vital step in the development of medicines, especially for conditions related to human genetics. Considering the hopes that were placed on the **HGP**, including over-zealous anticipations from many prominent scientific figures, little by comparison came to immediate fruition. The problem was that although the mapping had been completed, there was a fundamental lack of understanding of the genome and its constituent parts. The tools to derive understanding of the genome had largely been ignored, favouring the development of tools and techniques to quicker map the remaining parts of the genome [3]. Some believed, perhaps strangely, that the mapping alone of the human genome would herald a new era of biological discovery.

It is too easy to identify the failings of such a large and ambitious project, especially with the benefit of hindsight and it is unfair to overstate the shortcomings of the **HGP** when clearly so much has been achieved through its existence. Many mapping techniques were conceived and improved over the course of the project including many controversial strategies[4]. As the **HGP** continued, it drove forward surrounding areas of research that were required advancement in order to support the progressing requirements. Many requirements related directly to the enormity of the data that needed to be stored and managed. This presented several problems for the data management community. The volumes of the data were not specifically problematic; many databases are required to hold large quantities of data. But when you consider the amounts of data being appended and modified on an often daily basis, the presentation requirements of the data and, given the public nature of the repository, the storage and management of the data becomes anything but trivial.

Much, if not all, the emphasis of the **HGP** was aimed at the mapping of the human genome. Genomic data was added to the project as quickly as possible, largely driven by the desire to identify and patent certain *valuable* genes but also by an equally powerful and conflicting desire; to freely distribute the human genome to the public. Ultimately, the pioneers of free access to the genome succeeded and in March 2000, the then US president, Bill Clinton, announced that patents could not be taken against the human genome and the information should be freely available to all researchers. The pace that

[3]Ironically, considering the primary objective of the **HGP** was to *understand* the genetic makeup of the human species.

[4]We refer to Craig Venter's *whole genome shotgun sequencing* technique, which had already been used to sequence bacterial genomes of up to six million base pairs in length.

the human genome was assimilated later proved problematic. There were and still are many mistakes, discrepancies and even duplicates within the genomic sequence. This is true for various reasons ranging from simple mistakes in the mapping of the sequences to more complex situations. Sometimes, the same gene would be sequenced by two or more researchers and then published to the repository. In some cases, due to differences in method or metadata, these submissions would be identified as different, creating phantom genes.

Much research has been devoted to addressing these problems, particularly from the areas of computer science and software engineering. There are numerous opportunities for enhancing and improving the processes involved in genomic and proteomic experimentation and arguably the most significant factor to address is the management of the biological data used during experimentation. Most research, initially at least, aimed at dealing with the vast volumes of data being employed but as experiments became more complex and required access to multiple datasources, the community needed to address the problem of significant heterogeneity between datasources. In the rush to amass the data and provide mechanisms to present it to the public, there had been very little effort dedicated to ensuring any kind of standardisation[5]. Most datasources, therefore, possessed differing data formats even when describing the same data[6].

As these datasources were being appended, modified and annotated, researchers were conducting experiments based on these constantly evolving resources. Due to the changes, experiments would need to be re-run in order to reflect the most current state of the data. When you consider environments where experimental results are reused for further experimentation and there exist many nested levels of experimentation, a change at the original source data can cause havoc when attempting to reflect the current state of the data. In many cases, updates in the source data are not propagated and this causes inaccuracies in the experimental environment and the experimental results. Frequent changes to the experimental environment are even harder to manage as they often occur frequently and with no prior warning. An unannounced change can cause results to differ from one moment to the next with very little evidence of the cause of the difference.

In comparison to other areas of research in biological data, the management of biological data change has largely been ignored. A detailed analysis of a biological experimental environment is required with particular focus on the elements that can change. It is not only the data that is subject to change. Today, biologists employ a multitude of tools and methods in order to achieve the desired result and these too are subject to change. The experimental environment can be very complex and when a change occurs,

[5]With some notable exceptions; MIAME - Minimum Information About a Microarray Experiment.

[6]The same gene can be represented differently in different datasources, although the data is inherently identical.

it is imperative to know where the change has originated, how it affects the cooperating experimental components and how these changes can impact any existing results.

1.2 Problems with Biological Data

Biological data has three primary characteristics that render change management problematic; high volume, disparate heterogeneous sources, and the complexity of the associated semantic data. Although the first two have received considerable attention [61][51], comparatively little effort has been made in managing the semantic complexity in relation to the ongoing change.

Of all the scientific disciplines, biology has one of the most complex information structures with respect to the concepts, data types and algorithms [35] and the richness of the metadata[69]. Its richness and diversity provide many challenges for biological sciences, computational sciences and information technology. Definitions for the data must represent the degree of complexity within the substructure and relationships of the data and ensure that no information is lost during the biological data modelling. The data model must reflect several key points; multiple levels of schema complexity, the data relationships and the hierarchy and structure of the data points themselves. The variability of the data is high, therefore requiring flexible handling of data types and values.

There exists a substantial amount of biological data stored in a variety of formats in a multitude of heterogenous systems. Accessing the relevant data, combining data sources and coping with their distribution and heterogeneity is a very difficult task. Data systems frequently overlap in the data types between organisms or genome projects. This requires additional efforts during recombination and data integration of these geographically dispersed, heterogeneous, complex biological databases is a key research area [98]. One of the principal issues of data integration is the data format. Many current biological databases provide data in flat files which are poor data exchange formats. Worse still, each biological database has a different format, making the integration of such datasets difficult and time-consuming.

Biological database schemas change at a highly rapid pace and in most relational and object database systems, it is not possible to significantly extend the schema. Rather than incrementally and dynamically extending the schema, scientific databases such as **GenBank** release a new version of their database reflecting the new schema. This can sometimes be transparent to the user but it can often cause problems nonetheless. Changes in database schema should not, by definition of its semantic nature, affect the meaning of the data, although it can significantly impede the ability of the scientist to quickly establish impact to their results, requiring them to accommodate the new schema in their *in silico* experimental environment.

One issue arising from the use of biological data stems from the nature of the users of such data. Many, if not most, of the users of biological data will be biologists. Such biologists will likely have at least a basic familiarity with databases and their methods for interaction but the main goal will be centered on a biological hypothesis. The interface should therefore present the database to the user in a manner appropriate to the problem being addressed whilst reflecting the underlying data structures. A simple fact, although certainly not representing all, is that most biologists will either not know or not care about internal data structures or schema design and this can cause problems, especially when a change to these areas is required. These problems are generally addressed at the presentation level of the database with many providers opting for a user-friendly, transparent web interface. The transparency of such interfaces can also often cause problems at the point of a schema change, especially for *power users*, where the change is not properly documented or has been hidden.

Biologists access the databases using *queries*. The definition and representation of complex queries is extremely important for biologists and the average user will not be able to manage this without assisting tools. These tools are usually bespoke and provided as part of the web interface for the user and, therefore, often differ between databases but these aspects alone do not convey the full extent of the problems associated with biological data. The problems with biological data become unique when considering semantics. One of the distinctions of biological data with respect to other types of scientific data, is the complexity and variety of the experiments that yield the data. These characteristics hold influence over the data generated, but are often not recorded completely in the metadata. Representations of the same data and, indeed, the same or similar experiments by different biologists will most likely be recorded differently. Due to the similarity of the experiments and in spite of the designer differences, common points can be found and queried to understand the connections between apparently unrelated concepts.

1.3 Experimental Design

Consider the actions and behaviour of a computational scientist working with biological data. In order to begin experimentation, the scientist will have access to some form of database. This may be a locally-held dataset, a private collaborative effort containing experimental results or a public data repository. Once the scientist has obtained the data, it may need to be wrapped to become compatible with the current form of experimentation. Discounting the possibly numerous layers of manipulation that may be required to prepare the data, the scientist will choose an experimental method[7] with a set of suitable parameters.

As experimentation continues, the results grow and the scientist will require a log of completed experimentation phases. Further experimentation should be conducted based on the results from previous experiments and the log should be capable of representing this. Inevitably, the log will swell. Eventually, the information required to complete a single phase of experimentation will increase to the point where productivity is reduced. Changes can occur to the experimental design upon consideration of the results. Parameters may be tweaked, third-party tools may be updated or altered. These experimental details should be maintained as the results obtained are directly dependant on them.

The cycle of experimentation continues. But what happens when the original data source is updated to a new version? The large biological data repositories, from which the data is obtained, currently have no way of dynamically reflecting changes, updates, additions or deletions. They therefore, release flat files from time to time, representing one complete version. The scientist will want to know whether the data they have used during experimentation has been affected by the update and, more importantly, whether the affected data will affect their previous results. Based on their estimates of the impact to their results, the scientist will choose whether or not to recompute their results.

1.4 The Need for Component Tracking

When considering the provenance of biological experimentation, the history of the data is obviously very important. It is imperative to have detailed information about where the data came from and how it has changed over time to understand how results may be affected by change. However important the understanding of the data that feeds experimentation, we must also understand the nature of the other biological components that enable and aid the experimental process. A change or update in one of the experimental components is just as capable of affecting the results.

The tracking of data from one version to the next manifests its own challenges and our research aims to provide some answers but there are many areas of research aimed solely at the tracking of data, even for data as complex as ours. To address the tracking of biological experiments, we must first understand the nature of the biological problems being solved as well as the processes that are involved in the cycle of experimentation.

[7]We define an experimental method as a transformation of experimental data from one state to another.

We should not understate the importance of *all* the processes that form biological experimentation, including all data transformation tools, methods and post-processing tools and these components can change unpredictably and sometimes transparently. Each time they are used for an experiment, they may be configured differently and these configurations may affect the output (i.e., the results). They should therefore be recorded. It is the complexity of biological experimentation that requires a unique solution and this requires the tracking of all the components involved.

This presents some interesting problems. We can easily picture how data can affect results. If you change the input data, you will obviously affect the output data. The nature of these impacts depend on several factors. The quantity, frequency and severity of the changes will constitute the bulk of the impact but there are other, more subtle factors. Output impacts cannot be estimated simply by analysing the input data, or any other inputs for that matter. A dataset may change by a factor of 0.5, meaning that half of the data has changed in value but what does this factor mean to the results? It could be naively concluded that if the inputs change by a factor of 0.5, then the outputs can be estimated to be impacted by a similar factor. This may well be a good estimate, as it is often the case that inputs directly impact the outputs with a linear factor but this will certainly not hold true for all scenarios.

We must also consider the *topological importance* of a dataset. Topological importance refers to the near ubiquitous scenario where some areas of a dataset hold more importance to a particular line of experimentation than others. The experiment may actually only use a small fraction of the input data and we refer to that fraction as having a greater *topological weight*. Perhaps one column is particularly important and the rest of the dataset is required only for historical purposes. The experimental processes and methods may, and most likely will, not use the dataset evenly. It is therefore important to know not only how the dataset has changed as a whole but also where the changes have taken place within the dataset and how the areas of topological importance affect the experimentation.

In conclusion, we must identify the topological importance of each dataset in the same way that we must establish a topological importance amongst all components (including datasets) as components will not all contribute equally to an experiment. In order to correctly establish the impact of a change, we must possess the following information; the nature of the change[8], the location of the change within a component in order to establish the topological weight, and the topological importance of the component within the experimental environment.

[8]The nature of the change refers to some information regarding how an item has changed from one state to another.

1.5 Managing Changing Biological Data

Most data in publicly available biological databases changes in an unpredictable and unannounced manner. Large amounts of complex data from multiple, heterogeneous sources are obtained and integrated using a variety of tools and these data and tools are subject to frequent change. It is the very nature and characteristics of these changes that require specific management and representation. The need for genomic information, spurred on by the quest to complete the mapping of the human genome, required a new kind of database. These databases were intended not only for public access but also to allow widespread data submission, the aim being to accrue as much data as quickly as possible. The side-effects of the new style data-consuming databases have been numerous and have attracted much research. These effects range from standardisation problems to erroneous or duplicate entries.

Over time, these databases are added to or amended and their content naturally changes, representing a significant problem. If the submission process is open, there is no way of knowing how the database is changing from one moment to the next as data can be added at any time. Not only does the database change unpredictably over time but with open submission, there can also be problems with the existing data. The quality requirements of the submitted data to some publicly available databases have been deliberately lowered in order to allow the rapid assimilation of data. Data quality had, in the past for many databases, been left to the conscience of the individual scientist. Erroneous data inevitably crept into the databases, exacerbated perhaps by the commercial incentives and the voracity of the public attention drawn by the level of contribution to the genome projects. But it is not fair to imply that the levels of erroneous data were due solely to overly competitive scientific work and rushed results. Errors occur everywhere regardless of intent or efforts to eradicate them. The problems lie as much with the data warehouses that accept the erroneous data as with the scientists that produce them. With such high volumes of data accepted on a daily basis, there are obvious difficulties enforcing rigid error-detection. Moreover, many errors are undetectable by considering the data alone, requiring more specialised methods.

The database resulting from the above situation is subject to frequent and unannounced change, both from the continual and unhindered addition of data and the retrospective repairing of existing data. Due to the frequency of the changes, most databases are not capable of reflecting these changes immediately and due to the large volume of changes, they cannot be individually highlighted. Most database providers overcome this by releasing versions of their database, usually as flat files over a period of time. This provides users with a way of binding their results to a specific release of the database thus identifying the set of required components with which to recreate their results.

In other areas of research, this problem has been addressed with the use of versioning. Versioning helps the user track changes to documents or other files and allows the user to investigate previous iterations of work as well as the ability to roll back to them. This traditional view of versioning has served its purpose well for documents or other types of textual file tracking but lacks a degree of control for the application of biological experimentation.

A problem occurs with the work and results that have already been obtained. When a data provider releases a new version, how will this new version and the changes therein affect the previous work? There must be a mechanism for detecting changes in the new version. One change can hold more significance than another, however, and this depends entirely on the nature of the results and the experiments that yielded them. We must then estimate if and how these changes affect the results, the significant changes identified and the insignificant ignored. We can then estimate the impact that the significant changes have on our previous work. We do not want to have to recompute our results every time there is a new version of each datasource. Rather, we would like to be able to estimate the impact that a new version generates and, analysing these changes within a given threshold, decide whether to recompute results or not.

But why hasn't this problem been addressed before now? As we have already mentioned, the focus for the data providers has been the assimilation, management and improvement of the data. The focus of specialist data warehousers is the integration and fixing or cleaning of this data and the focus of the scientist is to assess the data and incorporate it into their work. All the stakeholders concern themselves primarily with the access and use of the latest version of the data, whatever the consequence of using that latest version may be. Indeed, the issue of database versioning and the updating of results is one that plagues bioinformaticiens. Until now, they have been left to deal with this problem themselves and to use their judgement to resolve the update requirement. With this in mind, consider the fact that the majority of bioinformatic workers come from either a biological or medical discipline with little or no previous software engineering experience. Couple this with their focus on achieving results and it becomes obvious why the provenance of the data and tools to deal with the fluctuating experimental components are of lower priority.

1.6 The Contribution

It is preferable to have complete knowledge of each and every experimental component from its initial value to the current state as well as every iteration between. This may or may not be possible. We can comprehend the management of a dataset including every previous version with all the metadata involved over the lifetime of that dataset. Each tuple and column can be encapsulated into a data model. Each data item in the row or column can change at any time and it is clear how these changes are represented in the dataset. In this way, monitoring change in datasets is *relatively* straight-forward. When considering the implications of using a software application as part of the experimental environment, monitoring change becomes much more complex. The applications available are as various as the experiments themselves and while a commercial off-the-shelf (**COTS**) product may have clear and transparent version iterations, it may be less obvious for the more obscure community-driven tools. It may be impossible to derive the differences between versions, let alone understand how these differences may affect your work.

Bespoke components are easier to deal with. If the application has been developed *in-house*, changes can be identified and announced in a manner befitting the level of communication within the individual working environment. There may even be a way of directly modelling what has been changed. For most other applications, especially **COTS** products, the application logic is sealed and the researcher must rely on the developer to tell them what has changed. In such a scenario, we must use information about the component to describe what has changed. We can identify various aspects of the product and record this metadata as a description of the persistent state of the product. We could then pursue our own model for managing change based on the component metadata rather than the component itself. Recording the component metadata is not as accurate as encapsulating the component itself and the success of this approach depends heavily on the metadata selected from the component as well as the sensitivity placed on each feature.

Having adequately recorded each component, we must then describe how each affects another. This is no easy feat considering the potential heterogeneity between components. There can also be situations where components of differing types can impact one another. Specifying these impacts is certainly not trivial. Comparing datasets of different source and function is difficult enough; consider the problems when comparing datasets against third-party tools or computational methods. We need a way of making these comparisons in order to successfully model the experimental environment. Seemingly disparate components can exhibit profound and sometimes serious impacts on one another. Conversely, obviously similar components can bear no actual relation to one another. We can hypothesise which components are related but this will rarely be accurate for more complex component environments. Occasionally, the impacts can be so small and subtle that only the researchers directly involved will be aware of them. Specifying each and every impact by hand is obviously a very inefficient way of describing environmental impact and, for large environments with many components, will usually be too time-consuming. It is clear, however, that some impacts will need direct user input.

It is important to carefully balance the level of user interaction and system automation in order to make the process of change management as useful and, conversely, as painless as possible. We aim to provide a change management system that allows users to model experimental environments and the impact relationships therein. We must do this with sufficient accuracy to enable impact tracking that can estimate propagated component change. The application will employ a conjoined user/machine strategy to define component relationships. In other words, the change management application will be able to estimate inter-component relationships and then allow user input, if necessary, to tweak any inaccuracies in that estimation.

We have discussed the need for component tracking along with the inherent problems and we have introduced the idea of using component metadata rather than encapsulating the actual component. If the component metadata and interactions are going to be recorded correctly, there must be a coherent, persistent environment within which to model the experiments and the construction of this environment constitutes a major factor for the success of this research. Not only should the *in silico* environment provide an adequate abstraction of the real experimental environment, but it must allow the continuation of this abstraction, propelled by the scientist so they may continue their own experimentation. The experimental environment must therefore allow user-input as well as being user-friendly.

There are pre-existing systems that record experimental design and flow and some of these are described in section 2.5.2. Some of these systems have the benefit of workflow re-enactment, referring to the fact that they have the ability to record experimental workflow and then recompute the experiments automatically without user-interaction at some later point. Although there are advantages to this approach, there are also some key disadvantages including the requirement that the workflow components are confined to those supported by the specific workflow enactment system. This will limit many bioinformaticiens who are required to write their own software as these bespoke programs will not fit into a conventional workflow problem[9].

By removing the capability of workflow re-enactment, we have removed the ability to automatically recompute our experiments. We do, however, allow for the entry of bespoke components regardless of format or standards and they behave in exactly the same way as any other component and, more importantly, can be measured against any other component.

[9]At least if they do fit, they will not be eligible for re-enactment.

1.7 Summary of the Contribution

We summarise our research contribution below;

- **Problem Definition** We introduce the idea of biological data and experiment versioning with a clear definition of related problems that lead ultimately to our research question.
- **Background Assimilation** A comprehensive and thorough examination of all the relevant background, together with an extensive review of existing related technologies.
- **Analysis of an Existing Case Study** An extensive analysis of an existing experimental environment containing a wide variety of biological data and experimental components detailing the interactions and impacts therein. This case study is an example of the requirement for change management and as validation for our change management framework.
- **The Hashed Data Model** A light-weight, abstracted representation of experimental data that, while simple to implement, enables change tracking at varying granularities as well as semantic differentiation of data.
- **The ExperimentBuilder** A complete model for an *in silico* experimental environment, encapsulating all required aspects of experimentation into a single, manageable framework, allowing investigation and integration of multiple experimental components.
- **Implementation of the Change Management Framework** A selection of change tracking and impact propagation methodologies integrated into the **ExperimentBuilder** to track changes in our experimental environment and determine the real-world cost of updating any affected components.

Chapter 2

Background

In chapter 1, we described our research question in response to an identified problem with current experimental processes. We briefly covered some of the basic background to the research and some of the problems already encountered. In the following chapter, we expand the background with a comprehensive review of the literature surrounding our work. We begin by describing some aspects of biological data, the primary data type for our research, followed by a description of some current versioning techniques in section 2.2. In section 2.3, we describe some existing examples of scientific data versioning as well as analysis of their respective limitations for our purpose. Section 2.4 presents some related work on ontologies and the semantic web and section 2.5 describes some current tools for describing data provenance.

2.1 Biological Data

The following section describes the organisational structure of some of the databases, repositories and warehouses involved in the storage and management of biological and medical data.

2.1.1 Organisational Characteristics

Sequence data is most commonly modelled as a succession of letters that represent the primary structure of the molecule or strand. Deoxyribonucleic acid **(DNA)** is a nucleic acid describing the genetic material of all living organisms and some viruses. **DNA** sequences consist of four nucleotide bases found in **DNA**, adenine **(A)**, cytosine **(C)**, guanine **(G)** and thymine **(T)**. Sequences are typically derived from an organism (the raw biological material) via a process referred to as *DNA Sequencing.* Other types of biological sequences include *protein sequences* (or *amino acid sequence*) which describe the order in which amino acid residues appear in the chain in the protein or peptide. The key participants of genomic and proteomic data warehousing are detailed below.

2.1.1.1 Genomic Data

There are three principal stakeholders in the collection, storage, curation and presentation of genomic information (the **National Center for Biotechnology Information** (**NCBI**), the **European Molecular Biology Laboratory** (**EMBL**), and the **DNA Data Bank of Japan**(**DDBJ**)). The three members of the **International Nucleotide Sequence Database Collaboration** (**INSDC**)[1], thanks largely to their exchange policy, have assimilated over 100 gigabases of sequence data, representing both the individual genes and complete genomes of over 165,000 organisms. The synchronisation of the three members is maintained with a set of guidelines, published by the International **INSDC** Advisory Committee, consisting of a common definition for the database feature tables. These guidelines regulate the content and the syntax of the database entries and specify a Document Type Definition (DTD)[2].

The **National Center for Biotechnology Information** (**NCBI**)[3], was established in 1988 under the **National Institute of Health** (**NIH**) as a national resource for molecular biology information. Among other activities, the **NCBI** creates and maintains public databases, conducts research in computational biology, develops software tools for analysing genomic data and aids in the dissemination of biomedical information. The primary aim of the **NCBI** is to aid the understanding of fundamental molecular and genetic processes that control human health and disease. This is achieved with the creation of automated systems for storing and analysing knowledge about molecular biology, biochemistry and genetics; facilitating the use of such databases and software by the research and medical community; coordinating efforts to gather biotechnology information both nationally and internationally; and performing research into advanced methods of computer-based information processing for analysing the structure and function of biologically important molecules[4]. **GenBank** is the **NIH**'s genetic sequence database consisting of an annotated collection of all the publicly available DNA sequences totalling over 37 billion bases [5][92].

The **European Molecular Biology Laboratory** (**EMBL**)[5] conducts basic research in molecular biology, aiming to provide essential services to its member states. **EMBL**

[1]INSDC, http://insdc.org
[2]For further information on INDSC syntax formats, please refer to section 2.1.2.
[3]NCBI, http://www.ncbi.nlm.nih.gov/
[4]From the NCBI mission statement - http://www.ncbi.nlm.nih.gov/About/glance/ourmission.htm
[5]EBI, http://www.ebi.ac.uk/embl/

also endeavours to provide high-level training for scientists at all levels. Together with the development of new instrumentation for biological research, **EMBL** seeks to further the fundamental understanding of the basic biological processes in model organisms. The **European Bioinformatics Institute** (**EBI**) is a non-profit organisation, which forms part of **EMBL**. The purpose of the **EBI** is to provide freely-available data and bioinformatics services to all areas of the scientific community in a way that promotes scientific progress. It aims to contribute to the advancement of biology through basic investor-driven research in bioinformatics. As part of the **EMBL**, the **EBI** also provides training to scientists and aims to disseminate relevant technologies to industry. Modern technologies have provided a vast amount of information on a variety of living organisms. There is a danger of being overcome by the size and complexity of this data resulting in the persistent requirement to collect, store and curate the information in ways that allow efficient retrieval and exploitation. The **EBI** aims to fulfill this important task.

The **National Institute of Genetics** (**NIG**)[6] was established in 1949 and acts as an inter-university research institute promoting collaboration and participation in graduate education. The **NIG** also serves as the center for a range of genetic resources, including the **DNA Data Bank of Japan** (**DDBJ**)[7]. The **DDBJ** is a nucleotide sequence database that collects, annotates and then releases the original and authentic **DNA** sequence data. The **DDBJ** aims to only release the sequence data after annotation. Given the large amounts of data deposited and the time it takes to provide sufficient annotation, there exists a significant backlog. This has prompted the **DDBJ** to explore new ways of annotating DNA sequences.

The Sanger institute[8] is a genomic research institute set up in 1992 that employs large-scale sequencing, informatics and analysis of genetic variation to increase understanding of gene function in health and disease. Responsible for sequencing over a third of the human genome itself, the Sanger Institute concentrates largely on human health. The Cancer Genome Project uses human genome sequences together with high-throughput techniques to identify cancer-critical genes [91]. It is worth mentioning the **Human Genome Project** (**HGP**), although it is already covered in section 1.1, as an example of a large scale, genomic collaboration involving the assimilation of data from disparate public submission. Many of the problems and characteristics of biological data that we describe here can be exemplified within the **HGP**.

[6]NIG, http://www.nig.ac.jp/index-e.html
[7]DDBJ, http://www.ddbj.nig.ac.jp/
[8]The Sanger Institute, http://www.sanger.ac.uk/

2.1.1.2 Protein Data

Proteomics, first termed in 1997, is the large-scale study of proteins in order to establish their structure and functionality[1][8]. A *proteome* refers to the entire complement of proteins produced by an organism or system[95]. The *proteome* will change to reflect the requirements of the host cell at any particular time.

The **Plasmodium Genome Resource** (**PlasmoDB**)[9] is part of an **NIH/NIAID** (**National Institute of Allergy and Infectious Diseases**) funded bioinformatics resource center to provide Apicomplexan database resources[10]. **PlasmoDB** is the official database of the **Plasmodium falciparum Genome Database** (**PFDB**) consortium incorporating both finished and draft sequence data as well as annotation emerging from Plasmodium sequencing projects. **PlasmoDB** currently houses information from five parasite species and provides tools for cross-species comparisons [3].

The **Protein Information Resource** (**PIR**) was established in 1984 by the **National Biomedical Research Foundation** (**NBRF**) to aid the identification and interpretation of protein sequence information supporting both genomic and proteomic research. Multiple protein databases and analysis tools have been made freely available to the scientific community including the **Protein Sequence Database** (**PSD**) and the first international database, **PIR-International** [96]. In 2002, the **PIR**, along with its international partners[11], were awarded a grant from the **NIH** to create **UniProt**, a single worldwide database containing protein sequences and function, by merging the **PIR-PSD**, **Swiss-Prot** and **TrEMBL** databases.

Swiss-Prot[12], created as a PhD project in 1986 and developed by the **SIB** and the **EBI**, is a curated protein sequence database that endeavours to provide a high level of annotation such as protein function, structure and post-translational modifications. Each protein entry provides an interdisciplinary overview of relevant information bringing together experimental results, computed features and conclusions[10].

The **Protein Data Bank** (**PDB**), established in 1971, is a repository for the 3D structural data of large biological molecules and is overseen by the **Worldwide Protein Data Bank**. The **PDB** provides support in various areas of structural biology and many journals and funding agencies require the submission of structural data to the **PDB** before application. The **Human Protein Reference Database** (**HPRD**)[13]

[9]PlasmoDB, http://plasmodb.org

[10]The **Apicomplexa** are a large group of unicellular, parasitical protists[43].

[11]**EBI** and the **Swiss Institute of Bioinformatics** (**SIB**)

[12]Swiss-Prot, http://www.expasy.ch/sprot/

[13]**HPRD**, http://www.hprd.org/

represents a centralized platform to visually depict and integrate information pertaining to domain architecture, post-translational modifications, interaction networks and disease association for each protein in the human proteome[58]. All the information in **HPRD** has been manually extracted from the literature by expert biologists who read, interpret and analyze the published data[58].

There are a multitude of protein structure databases. **Proteopedia** is a collaborative, 3D wiki-style encyclopaedia of proteins, highlighting functional sites and ligands[50]. The **Macromolecular Structure Database** (**MSD**) is a European collaboration for the collection, curation, management and distribution of data about the macromolecular structure of a protein. Derived from the **PDB**, it was conceived as a single access point for protein and nucleic acid structures and related information[11].

The **Biomolecular Interaction Network Database** (**BIND**)[14] is a database designed to store the complete description of protein to protein interactions, molecular complexes and pathways[2]. The **Database of Interacting Proteins** (**DIP**) catalogs experimentally determined interactions between proteins, combining information from a variety of sources to create a consistent set of protein to protein interactions[97].

2.1.1.3 Other Data Organisations

The **National Biological Information Infrastructure** (**NBII**) was established as a result of a 1993 report [75] recommending the development of a national biotic resource information system to coordinate information about biodiversity and ecosystems. Currently, the **NBII** concentrates on increasing the access to data and information on the national biological resources. The **NBII** links diverse, high quality databases, information products and analytical tools maintained by the partners, government contributors, academic institutions, non-governmental organisations and private industry. In a nut shell, the **NBII** works on new standards, tools and technologies that make it easier to find, integrate and apply biological resource information[79].

The **Kyoto Encyclopaedia of Genes and Genomes** (**KEGG**), initiated in 1995 by the **Japanese Human Genome Programme**, consists of a collection of online databases warehousing data on genomes, enzymatic pathways and biological chemicals[55]. **KEGG** can be considered to be a "computer representation" of the biological systems and the database can be utilised for modelling and simulation as well as the browsing and retrieval of data[54]. The **MetaCyc** database contains information on experimentally determined metabolic pathways. It can be used to computationally predict metabolic pathways of organisms from their sequenced genomes and contains extensive data on

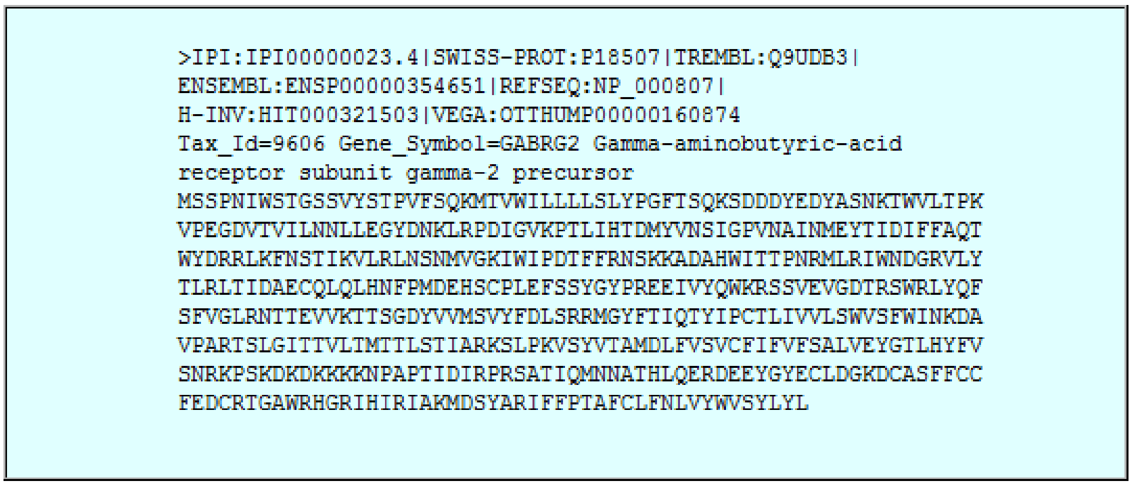
[14]BIND, http://binddb.org

individual enzymes, describing their structure, cofactors, activators and inhibitors as well as other information[56].

2.1.2 Data Formats

There are many different data formats and to mention all is beyond the scope of this document. We shall discuss the data formats considered to be most important and primary to this research.

Sequence formats are the way in which biological data such as amino acids, proteins and DNA sequences are recorded in a computer file. Sequence formats always consist of a set of ASCII characters and are arranged in such a way as to represent aspects of the data such as the ID, name, comments, etc. To submit data to a database, that data must be in the same format that the particular database expects to receive. There are numerous sequence formats in the biological domain, too many to discuss their differences here but, in general, each database will have its own sequence format. A sequence does not strictly require any sort of identification but most will have a header that contains at least one form of an ID, usually at the top of the sequence. Figure 2.1 illustrates part of a single sequence entry in the FASTA format.



```
>IPI:IPI00000023.4|SWISS-PROT:P18507|TREMBL:Q9UDB3|
ENSEMBL:ENSP00000354651|REFSEQ:NP_000807|
H-INV:HIT000321503|VEGA:OTTHUMP00000160874
Tax_Id=9606 Gene_Symbol=GABRG2 Gamma-aminobutyric-acid
receptor subunit gamma-2 precursor
MSSPNIWSTGSSVYSTPVFSQKMTVWILLLLSLYPGFTSQKSDDDYEDYASNKTWVLTPK
VPEGDVTVILNNLLEGYDNKLRPDIGVKPTLIHTDMYVNSIGPVNAINMEYTIDIFFAQT
WYDRRLKFNSTIKVLRLNSNMVGKIWIPDTFFRNSKKADAHWITTPNRMLRIWNDGRVLY
TLRLTIDAECQLQLHNFPMDEHSCPLEFSSYGYPREEIVYQWKRSSVEVGDTRSWRLYQF
SFVGLRNTTEVVKTTSGDYVVMSVYFDLSRRMGYFTIQTYIPCTLIVVLSWVSFWINKDA
VPARTSLGITTVLTMTTLSTIARKSLPKVSYVTAMDLFVSVCFIFVFSALVEYGTLHYFV
SNRKPSKDKDKKKKNPAPTIDIRPRSATIQMNNATHLQERDEEYGYECLDGKDCASFFCC
FEDCRTGAWRHGRIHIRIAKMDSYARIFFPTAFCLFNLVYWVSYLYL
```

FIGURE 2.1: A single UniProtKB/Swiss-Prot entry in Fasta format[15].

Some formats, such as **Genetics Computer Group** (**GCG**), **plain** and **Staden** must contain only one sequence per file. Other sequence formats can hold multiple sequences in one file, either concatenated or in an aligned set of sequences. **EMBL**-formatted sequence files can contain several sequences, each starting with an identifier line(ID), followed by further annotation lines. The start of the sequence is marked with "SQ" and the end of the sequence is marked with "//". A sequence file in FASTA format can also contain multiple sequences but marks the beginning and end of each sequence differently. A GenBank sequence separates its multiple sequences differently still.

Some databases such as **GenBank**, **EMBL** and **DDBJ** share similar data formats, albeit with different headers. In order to achieve transparency between them, they have introduced a syntax called **INSDSeq** consisting of the letter sequence of the amino acid sequence as well as the letter sequence for the nucleotide bases in the gene or decoded segment.

Microarrays have been one of the most important breakthroughs in experimental life sciences. They allow snapshots to be made of gene expression levels at a particular genomic stage[16]. Microarray data can be accessed through **Array Express**[17], a public repository for microarray-based gene expression data. Although in the past, much progress had been made and many significant results had been derived from microarray studies, the major limitation for this technology was the lack of standards for presenting and exchanging the microarray data [12]. The **Minimum Information About a Microarray Experiment** (**MIAME**) format describes the minimum amount of information necessary to ensure that the microarray data can be easily verified and enable the unambiguous interpretation and reproduction of such data[12].

The effective and efficient delivery of health care requires accurate and relevant clinical information. The storage of clinical information has traditionally been limited to paper, text or digitised voice. A computer cannot easily manipulate data in these formats. Clinical terminologies are also large, complex and diverse with respect to the nature of medical information that has been collected over the past 130 years of the discipline. There are numerous schemes which have proved successful in supporting the collation and comparison between collections,but it is also hard to reuse schemes for purposes other than for what they were originally developed and this causes the proliferation of even more schemes. **Galen** and the open source version **OpenGalen**[76] provide a formal model of clinical terminology in an attempt to address these problems.

Mass spectrometry is a powerful analytical technique used to identify unknown compounds and quantify known components, even in very minute quantities. It is also used to establish the structure and chemical properties of molecules. Mass spectrometry can be used to sequence biopolymers such as proteins and oligosaccharides, determine how drugs are used by the body and perform forensic analyses such as chemical identification to determine drug abuse[18]. Due to the large amounts of information that can be generated by mass spectrometry, computers are essential, not only to control the mass

[16]Sourced from http://www.ebi.ac.uk/Databases/microarray.html
[17]Array Express, http://www.ebi.ac.uk/microarray-as/ae/
[18]Athletic steroid abuse among others.

spectrometer but for spectrum acquisition, storage and presentation. Tools are available for spectral quantitation, interpretation and compound identification via online spectral libraries.

2.1.3 Biological Data Management

Over the past 25 years, biology and biological research has become largely dominated by what is referred to as the **genomic era**. It was originally believed that the mapping of the human genome alone would herald a new world of discovery for the life sciences. By the time of its completion, relatively little had been uncovered due to the lack of understanding of the genome. Arguably, this is still very much the case today. Large amounts of proteomic and genomic data have been collected into data warehouses, but there is still a lot to be done to enable its full understanding. The volume of the data presents a significant part of the problem and much research has been devoted to the problems of biological data and its management. Section 2.1.3 aims to present the most significant advances in these areas.

2.1.3.1 Data Access

There are a number of data sources from which scientists and researchers may wish to retrieve and access biological data, some of which have already been mentioned but there are many others[82]; **RefSeq**, **Molecular Interactions Database** (**MINT**), **IntAct**, **NCBI Taxonomy**, **Gene Ontology** (**GO**), **LocusLink**, **Entrez Gene**, **Homolo-Gene** and many more.

Generally, there are three types of biological database. The *primary* databases, such as **Swiss-Prot** or **GenBank**, contains information about the sequence or structure alone. A *secondary* database contains information that has been derived from a primary database. These include, amongst others, the **Structural Classification of Proteins**[19] (**SCOP**) database developed at Cambridge University, **CATH**[20], a hierarchical classification of protein domain structures database developed at the University College London (**UCL**), and **PROSITE**[21], a database of protein domains, families and functional sites developed at the Swiss Institute of Bioinformatics. *Composite* databases combine data from a variety of different primary database sources, often with heterogenous data formats, eliminating the need for searching through multiple resources. Raw data is extracted from multiple *primary* databases, converted into *composite* database

[19]SCOP, http://scop.mrc-lmb.cam.ac.uk/scop/
[20]CATH, http://www.cathdb.info/
[21]PROSITE, http://www.expasy.ch/prosite/

format (*data integration*), the discrepancies and errors are removed (**data cleaning**) and the data is enriched with information from relevant literature or from experimental results (*data annotation*)[61].

Most biological data management is conducted within specialist *composite* bioinformatic databases and are loosely referred to as *data warehouses*. These specialist bioinformatic databases contain detailed information such as functional and structural properties and expert-enriched information essential for the in-depth analysis of relevant data[61] and can be regarded as a subject-oriented, integrated, non-volatile, expert-interpreted collection of data in support of biological data analyses and knowledge discovery[80]. This level of detail is usually lacking in the primary databases such as **Swiss-Prot** or **GenBank**, which instead must concentrate on the volume of data. Specialist data warehouses generally exhibit several preferred characteristics; increased detail of annotation, cleaner data, integrated data from multiple sources, integrated searches, and specific analysis tools.

2.1.3.2 Data Integration

Data integration of geographically dispersed, heterogenous and complex biological databases is a key research area [98]. Amongst the most common problems are integrity, consistency, redundancy, connectivity, expressiveness and updatability[7] and **BIOZON** endeavours to address these problems, offering biologists new knowledge resources. Data integration consists of *wrapping* data sources and either loading the retrieved data into a data warehouse or returning it to the user. Wrapping a data source refers to the retrieval of data from a source and translating it to a common integrated format[64]. One of the principle issues of data integration remains the data format. Ideally, a simple self-describing format is best but many biological databases still provide data in flat files which are poor data exchange formats. Worse still, as we saw in section 2.1.2, each biological database often uses a proprietary format and tools must be created to deal with the heterogeneity. **Bio2X**[98] is a system that gets around this problem by converting the flat file data into highly hierarchical XML data using rule-based machine learning techniques.

A wide variety of biological experimental techniques are available from the classic methodologies to modern high-throughput techniques such as gene expression microarrays. Looking to the future, more and more disparate methods will be developed, continually pushing integrative technologies and driving research in this area. The **Multi-source Association of Genes by Integration of Clusters** (**MAGIC**) is a general framework that uses formal Bayesian reasoning to integrate heterogeneous types of high-throughput biological data[89]. There have been many other systems designed for the specific purpose of biological data integration but the problem of seamlessly integrating data whilst taking into account source content metadata, source-access cost, and query evaluation remains[64].

The **EnsMart** system[57][39] is a self-contained addition to the **Ensembl** software and data that provides a generic data warehousing system for fast and flexible access to biological datasets as well as integration with third-party data and tools. **EnsMart** is a system capable of organising data from individual databases into a single query-optimised system. The generic nature of **EnsMart** allows the integration of data in a flexible, efficient, unified and domain-independent manner[57] and one of the noted benefits of the **EnsMart** system is the ability to engineer your own version of the system with very little informatics expertise. This is useful when using data and queries not explicitly offered by the datasource, and especially important in light of the increasing interdisciplinary nature of this research area.

Data Cleaning involves the detection and removal of errors from data in order to improve the quality of the data. For some time data cleaning, as an identifiable implementation was ignored and there was very little peer-reviewed information on data cleaning techniques[90]. Recently, it has received more attention and for good reasons. The quality of data can deteriorate or simply require data cleansing for many reasons. Data may be conceived in a poor state and therefore require cleaning immediately. More often, data cleaning is applied to data that has undergone some transformation. For example, when multiple databases are integrated in data warehouses or federated database systems, there is often a need for data cleaning due to databases containing differently represented redundant data. The same data is represented in both integrating databases, however the data is represented with slight differences (spelling, word-order, etc.) and the merging tool regards the two data items as two individual items. Data cleaning tools are used to provide accurate and consistent data, consolidating different data representations and elimination of duplicate information[18]. Furthermore, data cleaning is imperative in excluding spurious genotyping results[65]. **Bio-Ajax** in one such framework which uses existing data cleaning techniques in order to improve data quality in biological information systems[48].

2.1.3.3 Data Annotation

Data annotations are semantically rich metadata applicable to a particular application domain that help further clarify *features of interest*[6], which are the very data items that the user wants to annotate[37]. Examples of data annotations include comments, descriptions, definitions, notes and error messages and they are usually attached to specific parts of the dataset, often in the form of Web accessible documents[37]. There is often the requirement for bespoke annotation application due to the unsuitability of the metadata schema employed by most relational databases [6].

A data annotation can also be a statement about the quality or *correctness* of an item of data. As such, it is important that this information is preserved as the data changes and new versions are created. **DBNotes** is a relational database system that preserves data annotations through transformations and querying, retaining the dataset provenance[21]. Data annotation has been widely recognised as a invaluable tool and, as such, has been implemented in many large-scale projects, including the **International Nucleotide Sequence Database Collaboration (INSDC)** where the **Third Party Annotation (TPA)** project collects and presents high-quality annotation of nucleotide sequence[22]. Annotation can be added to features of interest by those who are not necessarily responsible for the submission, requiring only high-quality data, resulting from experimental and inferred analysis, discussed and provided through peer-reviewed publications.

The **Distributed Annotation System** (**DAS**) is a client-server, open-source project used to share and collect genomic annotation information from a number of remote servers to a single local point[28]. The **DAS** can allow many layers of information to be gathered at a single point, integrating **DAS** annotation data from multiple sources in a simple, single view[68]. The **Otter Annotation System**[81] is one example of a manual annotation system developed as an extension of **Ensembl**.

2.1.3.4 Systems Biology

Systems biology is a biology-based interdisciplinary field, focussing on biological system interactions. It is usually defined as the antithesis to the *reductionist* paradigm, seeking to put together and integrate multiple complex systems into a single, understandable and measurable environment. More specifically, systems biology concentrates on the interactions between the components of biological systems, endeavouring to explain how these interactions relate to the function and behaviour of that system. Given the interpretation as a paradigm for obtaining, integrating and analysing complex data from multiple experimental sources using a variety of interdisciplinary tools, systems biology has been applicable for a number of key scientific fields including; genomics, proteomics, transcriptomics, translatomics and interactomics.

Systems biology often involves the development and construction of *mechanistic models*, mathematical models that combine theory-based methods and use computational tools to handle the large number of parameters. Systems biology also encapsulates the reverse-engineering and reconstruction of various systems based on the quantitative characteristics of their component parts[26][36]. One of the major challenges for systems biology is the involvement and engagement of specialists from different disciplines and computer science is optimally placed for aiding this endevour[30].

2.2 Current Versioning Techniques

In the following section, we describe the history behind traditional versioning and some of the tools that were created. But why is versioning important? Our research aims to estimate the impact that experimental components cause through updating. Central to this problem is ability to detect whether a component has changed and, more importantly, quantitatively analyse how the component has changed. Versioning is concerned with change, albeit traditionally text-based change, but it is a good place to start.

2.2.1 Current Tools

There have been numerous approaches to versioning files and data and some are described below. There were several early file systems that provided versioning, such as the **Cedar File System** (**CFS**)[38] and **3D File System** (**3DFS**)[62] but these systems were not transparent. That is to say, users had to manually create versions using special commands and tools. Users of **CFS** had to change a copy of the file on their local system and a file was only versioned when it was transferred to the remote server.

Unix and Unix-derivative operating systems have a long history of incorporating version management providing tools such as the original *diff* and *patch* programs and these led to a number of versioning tools, the most popular being **Revision Control System** (**RCS**)[88] and **Source Code Control System** (**SCCS**)[77]. **SCCS**[77] was one of the first versioning tools that stored differences between file versions as strings of *deltas* to show the progress history of the file. **RCS**[88] supported both forward and reverse deltas, allowing **RCS** the ability to either store an initial copy with subsequent changes or store the latest copy with previous changes. The **Concurrent Versioning System** (**CVS**)[45] is also not transparent as users have to use specific commands to control the versioning of their files. **CVS** utilises a client-server architecture, whereby a client connects to the server in order to check-out a complete copy of the project, work on this copy and then later check-in their changes. **CVS** has become quite popular in the open-source world, largely due to its open-source development and subsequent widespread use. **Rational ClearCase**[22] is another versioning tool requiring specific administration and is also relatively expensive. BitKeeper[47] was developed more recently and aims to provide architecture to support globally distributed development. To work on a repository, the developer must first make a complete mirror of the original repository to work on, an idea similar to the **CVS** *sandbox*. When the developer wants to *push* or check in a version back to the tree, their changes are merged with the existing versions, which have since been pushed in from other developers.

Subversion is a version control system with a working model similar to that of **CVS**, and was intended not only to replace **CVS** but provide a versioned network filesystem over **WebDAV** (**World Wide Web Distributed Architecture and Versioning**)[23]. **Subversion** versions files and directories, provides support for metadata and can efficiently handle binary files. **V-Grid** is a framework for generating Grid data services with versioning support from **UML** models that contain structural description for the datasets and schema[63].

There has been a considerable amount of research on the versioning of **XML** data and documents. For this purpose, conventional versioning tools such as **RCS** and **SCCS** are inappropriate, due largely to the considerable computing overhead that occurs during the retrieval of a version[20]. Moreover, neither **RCS** nor **SCCS** preserve the logical structure of the original document and this somewhat negates the benefit of a hierarchical document such as **XML**.

There are also various snapshotting tools available such as **WAFL**[49] and **Ext3cow**[72]. With the snapshot method, incremental or whole snapshots of the file system are made periodically. When required, the entire file system can be restored from any recorded point in history. Snapshotting makes no allowances for how often a file may change moreover, it simply works on a discrete time period of data capture. The consequence of this is that files which do not change regularly will be continually 'snapshotted' regardless. Conversely a file that is being regularly updated may have updates that are not captured with the snapshot method suggesting that it may be inappropriate for biological data versioning. Also, snapshotting requires that an entire snapshot must be purged if the disk space is to be reclaimed.

[22]Rational Clearcase, www.rational.come/products/clearcase/index.jsp

[23]**WebDav** aims to provide a standard infrastructure for asynchronous collaborative authoring across the Internet.

Versioning employing the **copy-on-write** technique[24] is used by **Tops-20**[25], **VMS**[26], the **Elephant File System**[78] and **CVFS**[84][86]. An ideal situation would be to have versioning automatically integrated into the operating system. However, this has yet to be implemented.

The techniques described above have been implemented by high-level applications or at the filesystem level. There is an alternative technique that pushes the versioning functionality closer to the disk by taking advantage of modern, block-level storage devices [31]. Versioning at the block-level has several advantages over versioning at the filesystem or application levels. Firstly, it provides a higher level of transparency and is completely filesystem independent [31]. It also reduces complexity in the higher layers, in particular the filesystem and storage management applications [53]. Thirdly, it off-loads expensive host-processing overheads to the disk subsystem increasing the overall scalability of the system [53].

However, there are some distinct disadvantages to versioning at the block level. Having offloaded complexity from one level to another, it is still picked up somewhere and the ramifications of this transference is unclear. The consistency of the data may be affected with the use of the same volume as the filesystem but perhaps the most relevant disadvantage with respect to biological data versioning is the problem of versioning granularity. Since the data versioning is occurring at a lower system layer, information about the content of the data and metadata is unavailable as access is only available to full volumes as opposed to individual files.

2.2.2 Object Versioning

We must also decide the best way to deal with very large datasets. Conventional versioning systems are not designed to record such large datasets with so many versions. Given the amount of biological data that may be versioned, the prospect of simply storing multiple versions of the data as well as the associated metadata, presents a significant challenge. This subsection aims to explore the methodology behind current versioning techniques and describe some of the differences between various approaches.

There are two main methods for dealing with object versioning[59]. The first technique stores versions of a particular object as complete objects and is referred to as *complete versioning*. This approach is relatively easy to implement, although the waste of storage

[24]**Copy-on-write** is an optimisation technique that employs transparent concurrent use of a resource, applying versioning techniques only when required.

[25]From DEC, Digital Equipment Corporation, *TOPS-20 user guide (version 4)*, January 1980

[26]VMS File Systems Internals. *Digital Press*, 1990

space becomes more detrimental as the number of versions increases. The second technique stores a single object and each version is maintained as a difference between the current version and the previous version. This approach is more difficult to implement but it is more suitable for representing data that may be subject to continuous and dynamic change[59]. Using this approach, changes in objects are handled using a version management system. Each version reflects a change in the object's attributes and/or behaviour. Subsequent changes in the object will generate related dynamic attributes and temporal links to the updated versions. Storage is a serious consideration given the vast amount of biological data that may be effected by version management and this type of version management reduces the required storage space, since the current object is only stored once in its entirety[59].

Linear versioning is a technique where one version is stored as a complete object and the rest of the versions are represented as iterative differences between the versions. This approach is based on one-to-one versioning. That is to say, each parent or base object will have only one child or derived object. Linear versioning can be classified into two versioning strategies[59]. The first allows the current version to be calculated from the initial base version, with the addition of the subsequent versions representing the changes over time, and is referred to as *forward oriented versioning.* The second strategy stores the current version as the base version and previous versions are calculated as differences to the current version. This is known as backward oriented versioning[23]. The chosen method varies greatly depending on the type of manipulation that the data is to be subject to. Forward oriented versioning takes longer to retrieve the current object as differences must be added to the base version to get the current version but it takes less time to retrieve earlier versions. Backward oriented versioning provides faster access for the newest versions. As a result, this strategy is usually more suitable for most applications[59].

Branching is a technique where one version is stored as a complete object and all other versions are stored as differences between that version and other versions. As all versions are just one *difference* away from the current version, the *branch forward versioning* strategy provides the same access time for all versions.

2.2.3 Ontology Versioning

There are large amounts of heterogeneous biological data currently available to scientists. Unfortunately, due to its heterogeneity and the widespread proliferation of biological databases, the analysis and integration of the data presents a significant problem. Biological databases are inherently distributed because the specialised biological expertise required for data capture is spread around the globe at sites where the data originates. Biologists currently waste a great deal of time searching for available information and in order to make the best use of this data, we need to integrate the different kinds of information in a way that makes sense. The problem is further exacerbated by the variations in terminology used by different researchers at different times. An *ontology* provides a common vocabulary to support the sharing and reuse of knowledge. The **Open Biomedical Ontologies** (**OBO**) library[27] is a repository of controlled vocabularies, which has been developed in order to provide for improved communication across different biological and medical domains.

Ontologies are often considered as the basic building blocks of the *Semantic Web* as they allow machine supported data-interpretation reducing human involvement in data and process integration [17]. Ontologies provide a reusable piece of knowledge about a specific domain. However, these pieces of knowledge are often not static, moreover they evolve over time [60]. The evolution of ontologies causes operability problems, which hamper the effective reuse. Given that these changes are occurring within a constantly changing, decentralised and uncontrolled environment like the Internet, support is required to handle the changes. This is especially prudent with respect to the semantic web, where computers will be using the data. Humans are, however unlikely, more likely to spot erroneous data due to unexpected changes. One must also consider the nature, complexity and quantity of dependencies that exist between data sources, applications and ontologies, as changes in one area may have far-reaching effects[60].

Traditional versioning systems, for the use of text and code, enable users to compare versions, examine changes, and accept or reject changes. An ontology versioning system, however, must compare and present *structural* changes rather than changes in the text representation of the ontology. The PROMPTDIFF ontology-versioning environment reportedly addresses these challenges [67]. PROMPTDIFF includes an efficient version-comparison algorithm that produces a structural diff between ontologies.

2.2.4 Schema Evolution

Schema evolution refers to the evolution of a database schema in response to changes in the modelled structure of the data. Schema evolution affects not only the existing data but also the derived queries, applications and embedded experimentation therein. The traditional approach for schema evolution is to provide an open canvas for the schema which will fit any future change but this assumption is almost always inappropriate,

[27]OBO, http://obo.sourceforge.net

especially considering web information systems which, due to their distributed and co-operative nature, are subject to even stronger change. The evolution of the database has a strong impact on applications accessing the data, therefore there is a requirement for accurate and smooth evolution. There have been various surveys of schema evolution in object-oriented, relational databases[73]. Dynamic schema evolution is the ability of the database schema to evolve by incorporating changes to its structure without loss of existing data and without significantly affecting the day-to-day operations of the database[73].

The evolution of a database schema raises the issue of monitoring and managing those changes. Perhaps it is important to retain details of the previous schema versions if the data changes often. In such cases, defining the history of the data warehouse schemata is crucial for dependant users or applications[42]. Several schema versioning frameworks have been developed but the application of such systems can have significant semantic issues[44].

2.3 Scientific Data Versioning

The following section endeavours to present the current state of data versioning, specific to the case of scientific data. We begin by looking at some current successful examples followed by a critique detailing the unsuitability of such examples.

2.3.1 Successful Examples

This section looks at specific successful examples of scientific data versioning. Of particular importance are areas that seek to version data that is similar in structure to biological data. During the life cycle of a piece of software, different versions may be developed depending on the state of development and the purpose of the intended release. In order to effectively support software development, these versions as well as the relationships between them, should be stored[27]. The different versions are likely due to the continuing development or maintenance of that software. A specific set of versions may be needed to fulfill the requirements for a certain release of the software. For example, a shareware release of a software item may require a previous set of versions with limited functionality. A beta release may require the latest set of versions, irrespective of the level of testing undergone. The versioning of software files and projects is often conducted by a file versioning system, such as **CVS**[45]. **CVS** utilises a client-server architecture, whereby a client connects to the server in order to check-out a complete copy of the project, work on the copy and then later check-in their changes. Basic software versioning techniques typically provide versioning of large granularity objects at the file level, whereas it may be more useful and appropriate to version at a finer granularity level at the object and instance level.

An interesting example of successful data versioning in a scientific domain, presented by Barkstrom[4], involves large-scale scientific data production for NASA's Earth observing satellite instruments. This requires the production of vast amounts of data from a wide variety of data sources[28][4]. It is noted that while software versioning requires tracking changes principally in the source code, versioning of the data requires the tracking of changes in the source code, the data sources and the algorithm parameters and changes in any of these items can induce scientifically important changes in the data[4].

Shui et al. [83] present an XML-based version management system for tracking complex biological experiments. The framework uses generic versioning operations such as *insert* and *delete*, and defines three more; *update*, *move* and *copy* in order to describe changes in the XML-based description. The framework can store every single component of an entire experiment in XML. A change to any component will result in a new version. Users can then query the system to return sets of data so they can see the differences between sets of results according to the materials used. The title of the publication [83] reports to track complex biological experiments although the conclusion of the same publication clearly states that the framework only tracks changes to laboratory based data.

bdbms is an extensible prototype database management system for supporting biological data[29], extending traditional **database management systems** (**DBMS**) to include annotation provenance tracking, data dependency tracking and content-based authorisation for data curation, amongst other improvements. **bdbms** concentrates on data annotation, in particular its provenance between versions and introduces *Annotation SQL* (*A-SQL*), which allows the annotation and provenance data to be seamlessly propagated with minimal user programming. **bdbms** goes some way to identifying inter-data dependencies, highlighting the problem of data changes that cause subsequent invalidations[29]. **bdbms** presents the invalidated possibilities to the user, who can make the decision to act or not but the user is not given any information about the severity of the impact or the estimated cost of updating the data. The **bdbms** reports that a component *may* have become invalidated and this may be sufficient in an environment where detection alone is required.

[28]In fact, Barkstrom[4] quotes the production values at tens of thousands of files per day from tens or hundreds of different data sources.

2.3.2 Unsuitability of Present Techniques

When examining the unsuitability of various versioning techniques, it seems prudent to examine the overall requirement for versioning. The benefits of versioning can be grouped into three categories[84]; recovery from user mistakes, recovery from system corruption, analysis of historical change and the merging of disparate work sources. The first two items, although important in their own right, are not specifically applicable to the problem of biological data versioning. The analysis of historical change is important as it is specifically the history of the data that is significant to this research. Analysis of the history can help answer questions about how a file reached a certain state [84]. For example, **CVS**[45] and **RCS**[88] keep a complete record of committed changes to specific files. It is reasonable to assume that due to the various needs for versioning, different tools will be suitable for different aspects of versioning.

It would take too long to explain how each of the aforementioned versioning technologies are individually unsuitable for the versioning of biological data. It is important to first note that, with a few noted exceptions [4] [83], the versioning technologies described above have not been designed with the intention of versioning scientific data. It is therefore unlikely that the versioning tools will provide a sufficient solution for the prescribed problem. In short, current versioning technologies do not take into account the uniqueness of biological data and the constraints that versioning such data creates. This does not necessarily preclude these versioning tools from being used or modified for the purpose. The unsuitability of some of the more common versioning tools are described below.

When considering the unsuitability of a versioning tool, it is necessary to examine the function of the tool and measure that against the required functionality to solve the problem. In the case of biological data and the problem of versioning such data, we must consider a scenario where multi-versioned data can be processed in a variety of ways, producing multi-versioned results. When contemplating a problem of this kind, it seems obvious that a traditional file-versioning technology or a *snapshotting* tool will not provide the required level of flexibility. Traditional document version management schemes, such as **RCS** and **SCCS**, are line-oriented and suffer from various limitations and performance problems. Both **RCS** and **SCCS** may read segments that are no longer valid for the required version[20]. Also, **RCS** and **SCCS** do not preserve the logical structure of the original document. This makes structured-related searches on **XML** documents difficult. It may require that the entire original document be reconstructed before such a search can take place[20].

XML versioning provides a closer match for the required purpose[20]. It is suggested that **XML** document versioning provides a sufficient solution for managing the versioning of biological data experiments[83] but this approach only addresses part of the problem by tracking the changes to the data. If biological data is to be adequately managed and versioned, the semantics of the experimental process must be incorporated into the solution. Given the self-describing nature of **XML**, it seems a promising method for describing structured data.

Ultimately, there is no single technique that is completely adequate for our purpose and this is due largely to the complexities that can exist within scientific experimentation, in particular biological or medical. It is the definitions and the interaction between the experimental components that contributes to the complexity and any system that aims to provide change management must have, at minimum, the capability to fully record the entire experimental environment along with the behaviour and relationships in between.

2.4 Ontologies and the Semantic Web

There is a large amount of heterogeneous biological data currently available to scientists but due to its heterogeneity together with the widespread proliferation of biological databases, the analysis and integration of the data presents significant problems. Biological databases are inherently distributed because the specialised biological expertise required for data capture is spread around the globe at the sources where the data originates. To make the best use of this data, different kinds of information must be integrated in a way that makes sense to biologists. As the semantic web matures, the need for life sciences data integration grows, a problem that is further exacerbated by the lack of widely-accepted standards for expressing the syntax and semantics of the data[19].

Biologists currently waste a great deal of time searching for available information in various areas of research. This is further exacerbated by the wide variations of terminology used by different researchers at different times. An ontology provides a common vocabulary to support the sharing and reuse of knowledge. The **OBO** ontology library[29] is a repository of controlled vocabularies, which has been developed to provide improved communication across different biological and medical domains.

[29]**OBO**: Open Biomedical Ontologies - http://obo.sourceforge.net

2.4.1 The Gene Ontology

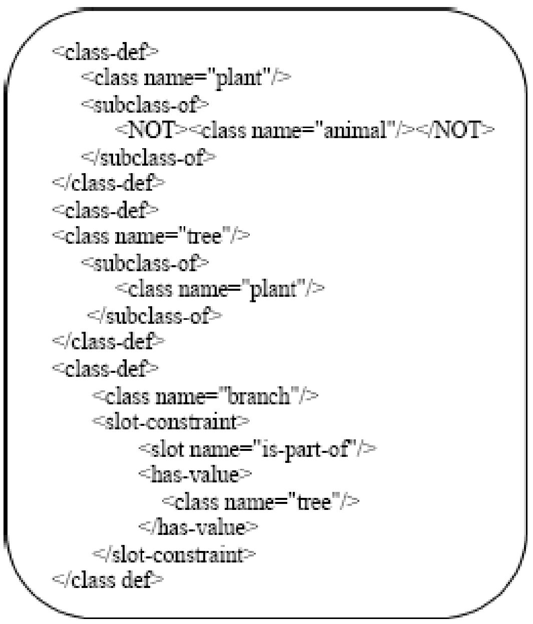
The **Gene Ontology** (**GO**) project[9][16] provides structured, controlled vocabularies and classifications covering several domains of molecular and cellular biology. The project is driven and maintained by the *Gene Ontology Consortium* whose members work collectively and, with the help of domain experts, seek to maintain, expand and update the **GO** vocabularies. Collaborations of this nature are difficult to maintain due to geography, misunderstandings and the length of time required for standardisation and collective agreement. The **Gene Ontology** (**GO**) project is a collaborative effort that aims to address two aspects of information integration; providing consistent descriptors for gene products in different databases and providing a classification standard for sequences and sequence features.

The project began in 1998 as a collaboration between three model organism databases: **FlyBase** (*Drosophila*), the **Saccharomyces Genome Database** (**SGD**) and the **Mouse Genome Database** (**MGD**). Since then, the **GO** consortium has grown to include many databases. The benefits of using **GO** increase as the number of participating databases increases. The **GO** project has grown enormously and is now a clearly defined model for numerous other biological ontology projects that aim to achieve similar results.

2.4.2 A Note on XML and RDF

XML (**eXtensible Markup Language**) and **RDF** (**Resource Description Framework**) are the current standards for establishing semantic interoperability on the Internet. However, **XML** only describes document structure. **RDF** better facilitates interoperation because it provides a data model that can be extended to address sophisticated ontology representation techniques[25]. **XML** is intended as a markup-language for arbitrary document structure. An **XML** document consists of a properly nested set of open and close tags, where each tag can have a number of attribute-value pairs. One of the important aspects of **XML** is that the vocabulary is not set, but rather can be defined per application of **XML**. The following example **XML** shows a part of a defined ontology.

XML is foremost a means for defining grammars. Any **XML** document whose nested tags form a balanced tree is a well-formed **XML** document. A **DTD** (**Document Type Definition**) specifies the allowed combinations and nesting of tag-names, attribute-names, etc. using a grammar formalism. The purpose of a **DTD** is to define the legal building blocks of an **XML** document. It defines the document structure with a list



```
<class-def>
    <class name="plant"/>
    <subclass-of>
        <NOT><class name="animal"/></NOT>
    </subclass-of>
</class-def>
<class-def>
<class name="tree"/>
    <subclass-of>
        <class name="plant"/>
     </subclass-of>
</class-def>
<class-def>
     <class name="branch"/>
     <slot-constraint>
          <slot name="is-part-of"/>
          <has-value>
             <class name="tree"/>
          </has-value>
     </slot-constraint>
</class def>
```

FIGURE 2.2: The **XML** representation of part of a defined ontology

of legal elements. A **DTD** can be declared inline in your **XML** document, or as an external reference.

RDF was designed by the **W3C**[30] in order to provide a standardised definition and use of metadata. **Resource Description Framework**, as its name implies, is a framework for describing and interchanging metadata. A *Resource* is anything that can have a URI (Universal Resource Indicator) such as a web page or an element of an XML document. A *PropertyType* is a resource that has a name and can be used as a property. A *Property* is the combination of a Resource, a PropertyType and a value. **RDF** is carefully designed to have the following characteristics[31]:

Independence: Anyone can use RDF to invent their own types of metadata. The PropertyType can be anything and is not domain-specific.

Interchange: RDF properties can be converted into XML.

Scalability: As RDF properties are essentially made up of three distinct parts (see above), they are easy to handle and look up, even in large numbers. It is important that metadata is efficiently captured. With the large amount of data, it is important that the overhead caused by metadata is as minimal as possible.

[30]The **World Wide Web Consortium** (**W3C**) is an international community developing standards to ensure the long term growth of the web. http://www.w3.org/

[31]According to http://www.xml.com/pub/a/98/06/rdf.html

In particular, **XML** falls down on the issue of scalability. Firstly, the order in which elements appear in an **XML** document is significant and can change the meaning of the document. When it comes to semantic interoperability, **XML** has disadvantages and since **XML** only deals with the structure of the document, there is no way of recognising or extracting semantic meaning from a particular domain of interest. The **Systems Biology Markup Language** (**SBML**) is an **XML** language for representing biochemical network models[30] in order to describe biochemical reaction networks. **SBML** is a software-independant language for describing models that occur in many different areas of computational biology including cell signaling pathways, metabolic pathways and gene regulation among others[52].

2.5 Data Provenance

The widespread nature of the Internet and the ease with which files and data can be copied and transformed has made it increasingly difficult to determine the origins of a piece of data. The term *data provenance* refers to the process of tracing and recording the origins of data and its movement between databases. Provenance is not considered a significant factor for many kinds of data but scientists focussed on the *accuracy* and *timeliness* of the data consider the provenance of data as a big issue[14]. Provenance allows us to take a quantity of data and examine its *lineage*. Lineage shows the steps involved in sourcing, moving and processing the data [71]. In order to provide full data provenance, all datasets and their transformations must be recorded.

In 2002, Frew and Bose [34] propose the following requirements for provenance collection; a standard representation so lineage can be communicated reliably between systems; automated lineage recording which is essential since humans are unlikely to record all the necessary information manually; unobtrusive information collecting is desirable so that current working practices are not disrupted.

Scientists are often interested in provenance because it allows them to view data in a derived view and make observations about its quality and reliability [15]. Goble [40] presents some notable uses for provenance;

Reliability and quality: Given a derived dataset, we are able to cite its lineage and therefore measure its credibility. This is particularly important for data produced in scientific information systems.

Justification and audit: Provenance can be used to give a historical account of when and how data has been produced. In some situations, it will also show why certain derivations have been made.

Re-usability, reproducibility and repeatability: A provenance record not only shows how data has been produced, it provides all the necessary information to reproduce the results. In some cases the distinction between repeatability and reproducibility must be made. In scientific experiments, results may be different due to observational error or processing may rely on external and volatile resources.

Change and evolution: Audit trails support the implementation of change management.

Ownership, security, credit and copyright: Provenance provides a trusted source from which we can procure who the information belongs to and precisely when and how it was created.

There are three further purposes for provenance[32] from the viewpoint of the scientist[99];

Debugging: Experiments may not produce the desired results. The scientist requires a log of events recording what services were accessed and with which data.

Validity Checking: If the scientist is presented with a novel result, they may wish to perform expensive laboratory-based experiments based on these results. Although sure that the workflow design is valid, they may still want to check how this data has been derived to ensure it is worthy of further investigation.

Updating: If a service or dataset used in the production of a result has changed, the scientist will need to know what implications that change has on those results.

2.5.1 Life Sciences

Among the sciences, the field of molecular biology has generated a wealth of biological data and is arguably one of the most sophisticated consumers of modern database technology [24]. Molecular biology supports many hundreds of public databases, but only a handful of these can be considered to contain *source* data in that they receive experimental data and many of the databases actually reference themselves. This sounds contradictory until you take into account that much of the value associated to a data source comes from the expert curation and annotation of the data.

Most implementers and curators of scientific databases would like to record provenance, but current database technology does not provide much help in this process as databases are typically rigid structures and do not allow the kinds of *ad hoc* annotations that are

[32]These are not entirely exclusive to those described by Goble [40].

often needed to record provenance [14]. There have been some attempts to formally monitor data provenance. *Query inversion* is a process that attempts to establish the origin of data by inverting the query on the output tuple but even for basic operation, the formalisation of the notion of data provenance is a non-trivial problem [14] but there are, however, data models where the location of any piece of data can be uniquely described as a path[13].

One of the most significant problems associated with data provenance can be seen with the following example[33]. Suppose document A cites a component of document B and then suppose the owner of B wishes to update it, thereby invalidating the citation in A. Whose responsibility is it to maintain the integrity of B? This is a common problem in scientific and, in particular, biological databases. The usual procedure is to release successive versions of a database as separate documents.

2.5.2 Workflow Enactment

It is not only the biological science domain that is concerned with data provenance. Large-scale, dynamic and open environments such as grid and web services build upon existing computing infrastructures to supply dependable and consistent large-scale computational systems [87]. Within both scientific experiments and business transactions, the notion of lineage and dataset derivation is of paramount importance since without it, information is potentially worthless. There are tools that provide provenance recording at an infrastructure level for service-oriented architectures such as the Grid and web services[87] and some also propose methods that uses provenance for determining whether previously computed results are still up to date[87].

Provenance capability in a grid or web service environment has two principal functions: to record provenance on dataset transformations executed, e.g. during workflow enactment, and to expose this provenance data in a consistent and logical format via a query interface.

Provenance information needs to be stored with two possible solutions for storage[87].

1. Data provenance is held alongside the data as metadata.
2. Data provenance can be stored in a dedicated repository, made accessible as a Grid or web service.

The first solution requires the holders of any such data to maintain the integrity of the provenance records as transformations take place, and it imposes significant changes to

[33]Taken from [14].

any existing data storage structures. Such a kind of provenance can be useful for a user to remember how a result was derived, and what steps were involved. It is unclear that such provenance can be trusted by any third party, since the data and provenance owner has to be trusted to have recorded provenance properly.

Workflow enactment is the automation of a process during which documents, information or tasks are passed from one participant to another for action, according to a set of declarative or procedural rules [87]. In grid applications, this task is often performed by a *workflow enactment engine*, which uses a *workflow script*, such as **WSFL**[34] or **BPEL4WS**[35], to determine which services to call, the order to execute them in and how to pass datasets between them.

2.5.3 ^{my}Grid and the Taverna Workbench

Grid applications require versioning services to support effective management of constantly changing datasets and implementations of data processing transformations. ^{my}Grid, as a pilot e-science project, aims to provide middleware services not only to automate the execution of *in silico* experiments as workflows in a Grid environment, but also to manage and use results from experiments [85]. The ^{my}Grid project is currently being developed using several molecular biological scenarios. ^{my}Grid is being used to automate complex and tedious *in silico* experimentation[36] by wrapping each web-based analysis tool and data resource as a web service. Within this process, often discarded intermediate results are assigned identifiers and published locally, so that they become resources of the personal web of science [46]. If provenance information is to be shared within ^{my}Grid, we need to overcome their heterogeneity and agree a common understanding (or semantics) as to what the contents of each data item and service represents and the relationships we provide between resources [99].

The following figure from Zhao [99] shows the architecture of the provenance providing components in ^{my}Grid and how provenance data are drawn together.

This following is a quote taken from the ^{my}Grid user guide (bundled with the ^{my}Grid download) and provides an adequate description of the Taverna workbench.

[34]The **Web Services Flow Language** (**WSFL**) is an **XML** language for the description of web services compositions.

[35]**Business Process Execution Language for Web Services** (**BPEL4WS**) is a standard executable language for specifying interactions with web services.

[36]Often, the mundane nature of the task means not all possible avenues are explored, because the scientist either misses possible routes or discards apparently uninteresting results.

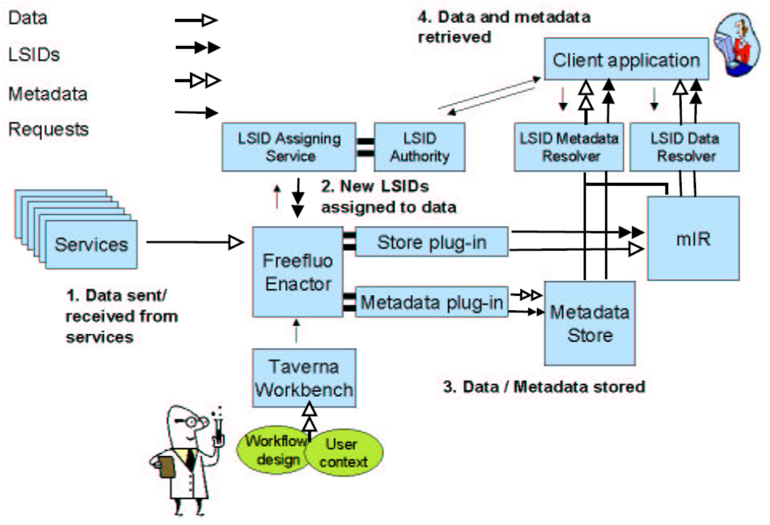


FIGURE 2.3: Architecture for provenance generation and visualisation in ^{my}Grid.

> First the user starts a workflow using the Taverna workflow workbench[37]. The workbench holds organisation in- formation such as user identity, which is passed to the workflow enactor together with input data and workflow specification. As the workflow is run, the enactor stores data (using the mySQL RDBMS), and metadata (in a Jena RDF repository)[38] corresponding to our four views of provenance. Each resource (including person, organisation, data and service) is assigned an LSID and made available via an LSID authority implemented using an open source framework[39]. Client applications can then visualize both metadata and data using the LSID protocol.

The main user interface to ^{my}Grid is the **Taverna** e-science workbench. **Taverna** provides a language and software tools to enable the design and implementation of workflows. In a bioinformatics context, a workflow is the entire process of collecting relevant data, performing any number of analyses over the data and extracting biological results. Often, in bioinformatics the result of one experiment can form the input values of the next. Designing workflows in **Taverna** allows services, or bioinformatics analyses, to be scheduled to run in series. If results are not dependent upon one another, services can be designed to run concurrently, allowing for faster, more efficient processing.

Workflows are designed in **Taverna** by selecting services and inserting them into a workflow diagram. Services are any bioinformatics applications that are available as

[37]Taverna and FreeFluo are both open source projects available from http://taverna. sourceforge.net and http://freefluo.sourceforge.net

[38]Jena is an open source semantic web framework for Java including an RDF repository http://jena.sourceforge.net/

[39]A LSID Java server stack available for download at: http://www-124.ibm.com/ developerworks/projects/lsid

web services. More and more applications are being provided as web services, but 'legacy' applications, either command-line tools, or simple web interfaces, can also be transformed into web services using wrapping tools, such as **Soaplab** and **Gowlab** respectively. This functionality means that any bioinformatics experiment that can be conducted elsewhere, regardless of the number of different processes or data sources required, can be run as a workflow in **Taverna**.

2.5.4 Data Provenance Tools

Data provenance is receiving an increasing amount of attention and this chapter describes some of the tools that are currently available. **Chimera**[32] is a virtual data system for representing, querying and automating data derivation. **ESSW**[33] is a nonintrusive data management infrastructure to record workflow and data lineage for computational experiments. **Tioga** is a proposal to modify an existing database visualiser built over **POSTGRES**, where user functions are registered and executed by the **Database Management System**, to provide fine grained lineage: lineage is computed from ancillary, user supplied weak inversion and verification functions. **CMCS**[70] is an informatics-based approach to synthesising multi-scale chemistry information that uses **WebDAV**. ^{my}Grid[85]is a high-level service-based middleware to support the construction, management and sharing of data-intensive in silico experiments in biology. **Trio**[94] is a system for the integrated management of data, accuracy and lineage. Within **Trio**, database views, modelled as query trees, can be supplied with inversion queries in order to determine the source tables used for that derived data.

Chapter 3

Case Study

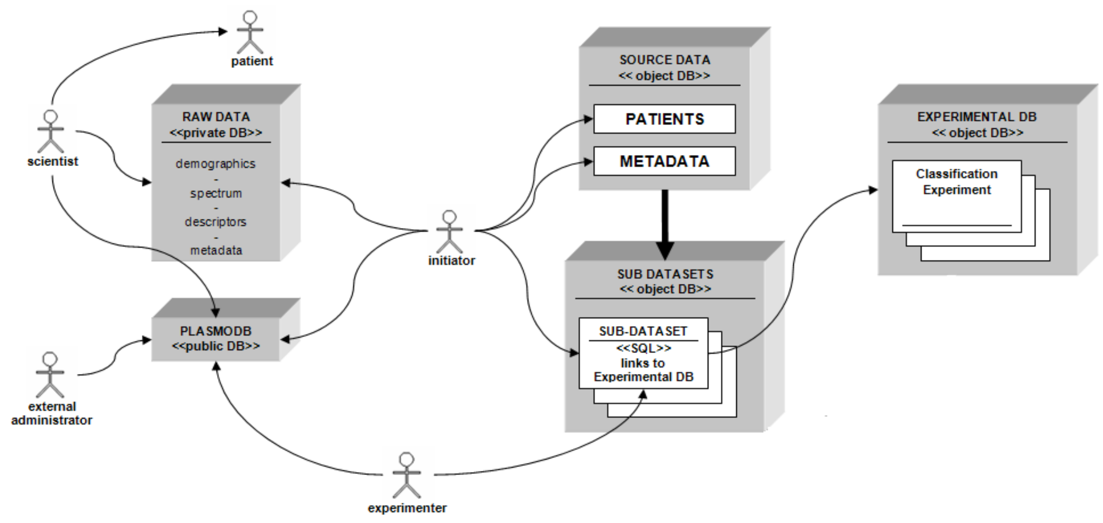
The aim of this section is to provide an analysis of an existing biological data environment. The process uses data from a private collaborative database consisting of clinical and proteomic data from 350 patients. The data is then mined using various machine learning techniques in order to derive patterns and relationships within the data.

We start with an overview of the example which has been illustrated as a non-structured rich picture in figure 3.1. This is the first step for the analysis and is used to identify the areas of interest and sub-systems present in the working example. Some of the parts illustrated are not within the machine domain of the example but do, however, affect the operation of the system[1].

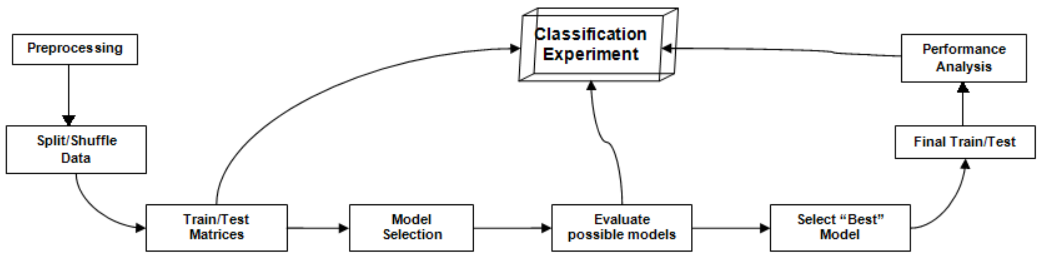
From figure 3.1, we identify five actors that interact with the system either directly or indirectly. The *scientist* maintains the private collaborative database which he/she populates with data from the *patient*. The *scientist* may also publish and submit data to a publicly available database, which is maintained by the *external administrator*, or they may use such a database to validate values from their own data. The *initiator* uses the private database and, possibly, the public database to initiate the bespoke database within the experimentation environment[2]. Once the relevant datasets have been initialised in preparation for experimentation, the *experimenter* can use one of the datasets in an experiment. The precise nature of the experiment depends on the hypothesis, method and parameters employed by the *experimenter*. Some of the generic stages are illustrated in figure 3.1. The experiment is run creating a *ClassificationExperiment* object populated with, not only the results of the experiment, but the model that has been used to attain the results, the method used and any other parameters specific to that

[1]Please refer to the change and impact analyses in sections 6.2 and 6.3.

[2]In this case, ObjectDB is used to store and maintain the initial data and any subsequent sub-datasets. For more information, see http://www.objectdb.com/



(a) An overview of the working example.



(b) An overview of the internal workings of a ClassificationExperiment.

FIGURE 3.1: Analysis of the existing environment.

experiment. The *ClassificationExperiment* object is maintained within an experimental database, which itself is linked to the previously-initialised sub-dataset.

3.1 Example Datasets

The dataset consists of a set of proteomic fingerprinting data generated at St. George's Hospital[3] and the **National Institute for Medical Research** (**NIMR**). The original format for the the data were Microsoft^{TM}Excel flat files. These were edited in Microsoft^{TM}Excel to produce comma-separated variable (**CSV**) files which were then tokenised and marshalled to create **Java Data Objects** (**JDO**s) within **ObjectDB**[4]. **ObjectDB** is a powerful **Object Database Management System** (**ODBMS**) written entirely in Java.

[3]St. George's Hospital, http://www.stgeorges.nhs.uk/
[4]ObjectDB, http://www.objectdb.com/

Participants were selected to form part of a clinical study at St. George's hospital in order to diagnose patients with tuberculosis. The dataset collection consists of data from 349 patients, each with a set of 219 protein quantitations. There are three datasets in the collection; the *Patient* dataset, the *Protein* dataset and a dataset metadata file.

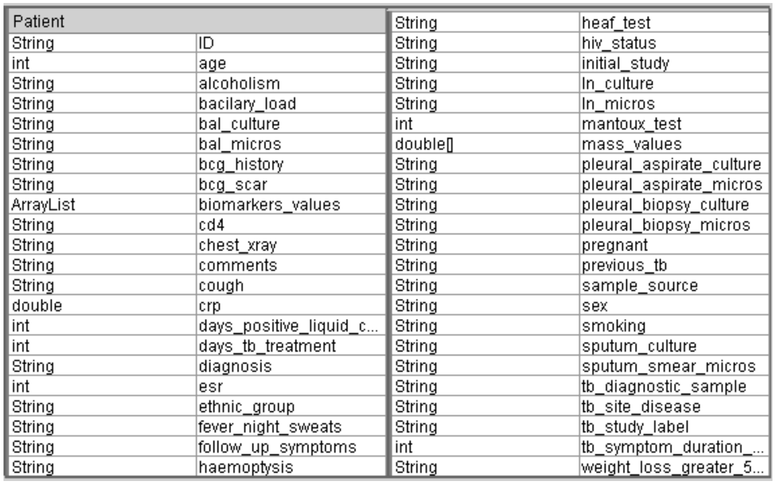
Proteomics is the analysis of complete complements of proteins. It is concerned not only with the identification and quantification of proteins, but also the determination of their localisation, modifications, interactions, activities and function. Proteomics is a useful tool for the analysis of various diseases. A disease can arise when a protein or gene is over or under-expressed, when a mutation in a gene results in a malformed protein or when the protein function is altered through post-translational modifications. In order to understand the biological process and to aid disease diagnosis, the relevant proteins can be studied directly.

There are several factors to consider when determining the validity of proteomic data. Proteomic data, like data obtained from almost any biological experiment, is subject to any number of external factors. A common technique for improving the reliability of a set of data and demonstrate the validity by reducing the effects of any external variables, is to repeat the experiment. How many times an experiment has been repeated lends weight to the validity of the resulting data. It is also important to look at the approach for protein quantitation in order to explain how it may impact the interpretation of any results obtained.

3.1.1 The Patient, Protein and Metadata Classes

The *Patient* class contains the demographic data for each of the 350 patients in the survey. Each patient contains 56 additional *data descriptors* illustrated in figure 3.2. The *data descriptors* combine a set of standard test and questions which can be obtained by relatively simple means. The *Patient* class contains details about a patient which relates, directly or indirectly, to the suspected disease.

The *protein* class contains a protein spectrum for each of the 350 patients within the survey. Each of the 219 protein quantitations form a select protein profile for each of the patients. Protein quantification is the process of establishing the exact quantities of various proteins in a given sample. There are numerous methods for determining protein quantitation, too many to mention here. Protein determination is important for many reasons, in particular for disease diagnosis, as the proteins involved in a particular disease can be identified by matching known *alleles* to those found in a suspected sample. Through ongoing protein determination, we can not only achieve an accurate diagnosis, but continue to monitor protein levels to ensure a successful prognosis of the disease.



Patient		String	heaf_test
String	ID	String	hiv_status
int	age	String	initial_study
String	alcoholism	String	ln_culture
String	bacilary_load	String	ln_micros
String	bal_culture	int	mantoux_test
String	bal_micros	double[]	mass_values
String	bcg_history	String	pleural_aspirate_culture
String	bcg_scar	String	pleural_aspirate_micros
ArrayList	biomarkers_values	String	pleural_biopsy_culture
String	cd4	String	pleural_biopsy_micros
String	chest_xray	String	pregnant
String	comments	String	previous_tb
String	cough	String	sample_source
double	crp	String	sex
int	days_positive_liquid_c...	String	smoking
int	days_tb_treatment	String	sputum_culture
String	diagnosis	String	sputum_smear_micros
int	esr	String	tb_diagnostic_sample
String	ethnic_group	String	tb_site_disease
String	fever_night_sweats	String	tb_study_label
String	follow_up_symptoms	int	tb_symptom_duration_...
String	haemoptysis	String	weight_loss_greater_5...

FIGURE 3.2: The data descriptors for each *Patient*.

3.1.2 Sub-datasets

Sub-datasets are created with the purpose of investigating various hypotheses. For example, tuberculosis and human immunodeficiency virus (HIV) are inextricably linked[66]. HIV progressively weakens the immune system, making the body vulnerable to infections such as tuberculosis. A dataset to investigate the correlation between HIV and tuberculosis is extracted from the dataset, *Patient*, with the following filter;

```
tb_study_label == "TB" && hiv_status == "Positive"
```

to find those patients with tuberculosis and who are HIV positive. Use the following filter;

```
tb_study_label == "TB" && hiv_status == "Negative"
```

to find those patients with tuberculosis and who are not HIV positive. These datasets can then be trained and tested upon using various machine learning techniques to find correlations in the data. Although the number of patients included in the dataset is not large, the number of derived datasets grows considerably and the size and complexity of the experimental environment can reach an unmanageable point. The current example dataset has twelve sub-datasets. Figure 3.3 shows some of the derived datasets from the dataset, *Patient*.

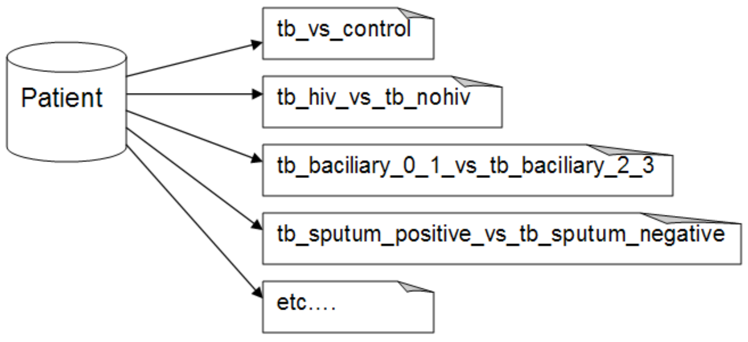

Patient
tb_vs_control
tb_hiv_vs_tb_nohiv
tb_baciliary_0_1_vs_tb_baciliary_2_3
tb_sputum_positive_vs_tb_sputum_negative
etc....

FIGURE 3.3: Some of the derived datasets from *Patient*.

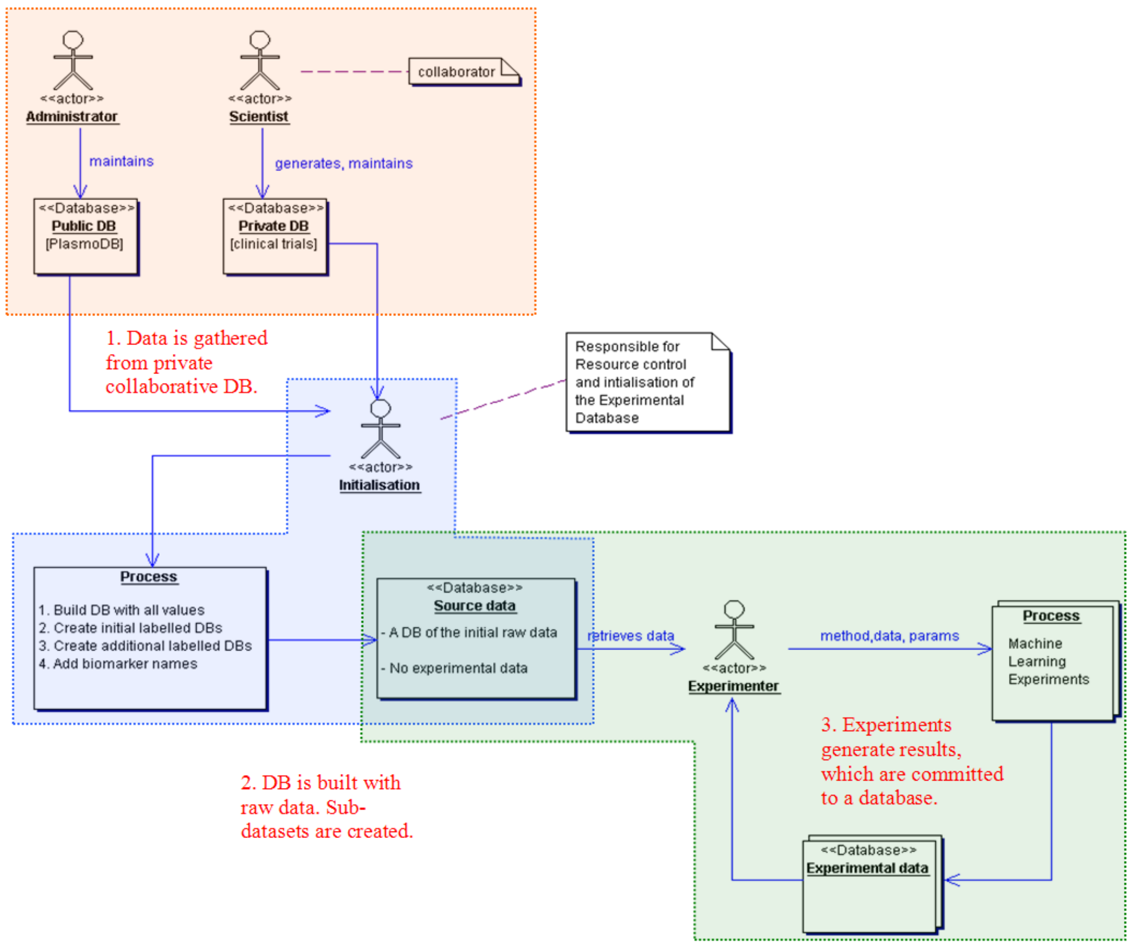
3.1.3 Dataset Characteristics

The example dataset exhibits a high level of complexity due to dependencies on both clinical and molecular data. The semantics of the data are more difficult to identify. There is less metadata than one might expect of a dataset of this size and type. There is little to suggest how the data was gathered. Also, data from various experimental techniques are present in the dataset. These include protein quantitations, chest x-rays, baciliary loads, biopsies and Fine-Needle aspirations (FNAs), although there is no associated metadata that describes any of the details of these tests. It is therefore difficult to assess the error margins or validity of the data obtained.

Further metadata is generated as more datasets are derived and experimented upon. With regards to the case of the tuberculosis-HIV correlation dataset, the following metadata was recorded; the algorithm, algorithm parameters (hypothesis), source and destination datasets, time and date, selected features and the feature selection method, the kernel and kernel parameters, model criteria and validation method. For this dataset, there are 18 classification experiments, each with differing data and metadata. This demonstrates how the amount of data and associated metadata can multiply at a rapid pace. How can an experimenter keep track of these multiple versions of experiments.

3.2 Current Workflow

The workflow for the case study is split into three logical parts. The first is the data collection stage where the experimental data is gathered from a series of private collaborative databases. This is accomplished by a collaborative scientist that generates and maintains the private, clinical data and an external administrator that administers the public database. The resulting raw datasets are then collated and rebuilt to form new sub-datasets, corresponding to the type of required experimentation. At this point, the experimental datasets are fed into the various computational experiments held by the scientist. This phase of experimentation generates sets of results and deals with the storage and administration of these results. Figure 3.4 provides an overview of the entire experimental process, detailing the interactions of the three stages mentioned above. The following sections describe these three stages in more detail.



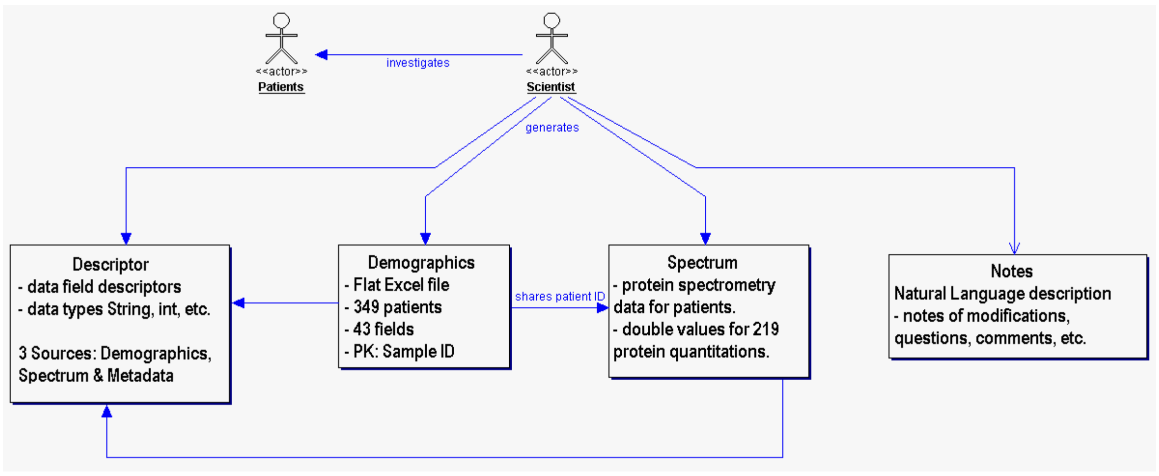
<<actor>> Administrator
<<actor>> Scientist
collaborator
maintains
generates, maintains
<<Database>> Public DB [PlasmoDB]
<<Database>> Private DB [clinical trials]
1. Data is gathered from private collaborative DB.
Responsible for Resource control and intialisation of the Experimental Database
<<actor>> Initialisation
Process
1. Build DB with all values
2. Create initial labelled DBs
3. Create additional labelled DBs
4. Add biomarker names
<<Database>> Source data
- A DB of the initial raw data
- No experimental data
retrieves data
<<actor>> Experimenter
method,data, params
Process
Machine Learning Experiments
3. Experiments generate results, which are committed to a database.
2. DB is built with raw data. Sub-datasets are created.
<<Database>> Experimental data

FIGURE 3.4: An overview of the entire experimental process.

3.2.1 Building the Datasets

The proteomic case study contains two data and two metadata files. The data files contain the patient demographics and protein mass values as described in section 3.1.1. The *Demographics* dataset contains the data for individual patients, including their study label (*training* or *testing*) and their diagnosis (*TB* or not *TB*). The *Spectrum* dataset contains the mass values for 219 testing proteins and is related to the *demographics* dataset by *SampleID*. There are two metadata files. The *Descriptor* file contains such information as field descriptors, field types and accepted input formats. There is an additional *Notes* file that contains general information about the datasets such as comments, questions and notes of data modification.

The first step in the data collection process is the investigation of the patients. Using both interviewing and surveying techniques, demographic and protein data were collected to generate the *demographics* and *spectrum* files and the metadata described above was generated by the administrator at St. George's Hospital.



<<actor>>
Patients
investigates
<<actor>>
Scientist
generates
Descriptor
- data field descriptors
- data types String, int, etc.
3 Sources: Demographics, Spectrum & Metadata
Demographics
- Flat Excel file
- 349 patients
- 43 fields
- PK: Sample ID
shares patient ID
Spectrum
- protein spectrometry data for patients.
- double values for 219 protein quantitations.
Notes
Natural Language description
- notes of modifications, questions, comments, etc.

FIGURE 3.5: The data files created by the collaborative scientist and the relationships between them.

Figure 3.5 describes the data files created by the collaborative scientist and the relationships between those files. Using the techniques above, the raw datasets were generated, ready for initialisation in the experimentation process.

3.2.2 Initialising the Datasets

Having obtained the raw data required for experimentation, the experimental datasets must be generated. Although the datasets have already undergone some refinement, it is unlikely that they are suitable for immediate integration into the experimental process. They must therefore pass through an initialisation process in order to create the experimental datasets and sub-datasets.

There are several stages of dataset initialisation and these are specific to the nature of the planned experimentation. The stages that we describe here are specific to the type of experimentation in the case study although there may, in other cases, exist other dataset initialisation techniques depending on the specific experiment. In any case, we have presented a comprehensive, if not complete, set of dataset initialisation techniques.

The first step is to encapsulate all the data about the patients into a single manageable dataset. The demographics and the protein mass values, linked by *SampleID*, are parsed for each patient into a single dataset, named *Patient*. An experimental dataset containing the dataset metadata, *TBProteomicDatasetMetadata*, is also generated using the demographics, protein mass values and data descriptors.

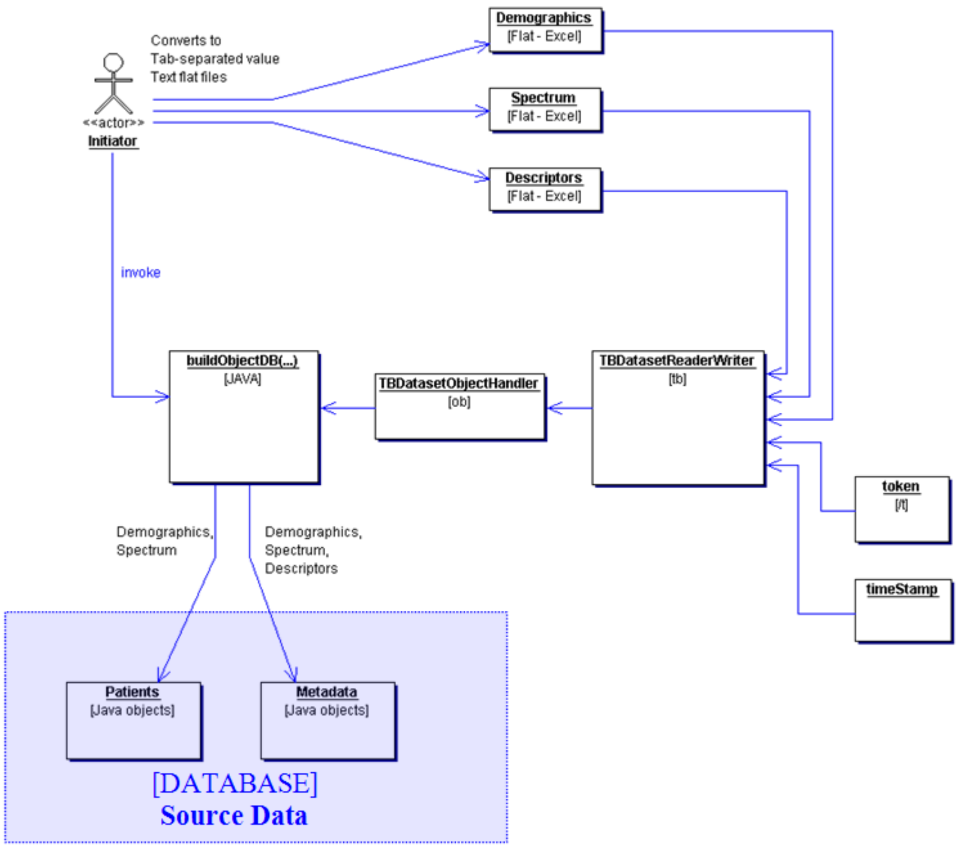


FIGURE 3.6: Primary initialisation of the Experimental datasets.

Figure 3.6 describes the initialisation of the source database using the raw data obtained in section 3.2.1. The raw data files were converted by hand from Microsoft Excel files to tab-separated text files in preparation for parsing. The nature of the parsing separating variable is passed to the system via the *token* file. A *timestamp* is also generated by the system clock for the metadata dataset. The *Initiator* invokes the buildObjectDB() method, parsing the raw data files creating *Patient* objects with the accompanying metadata, implemented as an **ObjectDB** database.

ObjectDB is an object-oriented database written entirely in Java. The primary function of **ObjectDB** is the ability to create, integrate and manipulate Java Data Objects (JDOs) rather than being constricted with a commercial relational database or requiring a bespoke solution. The **ObjectDB** user can manage all aspects of the database with

Java code. This has the benefit that it can easily be integrated into existing computational experimentation architectures, especially if those systems have been established using Java.

Once the Patient dataset has been initialised, the initial study dataset is created. The purpose of the initial study is to create sub-datasets from *Patients* grouped by their surveyed study label (*testing* or *training*) and their diagnosis (*TB* or *not TB*). These new sub-datasets, along with some initialisation variables are committed to the source database. This is illustrated in figure 3.7.

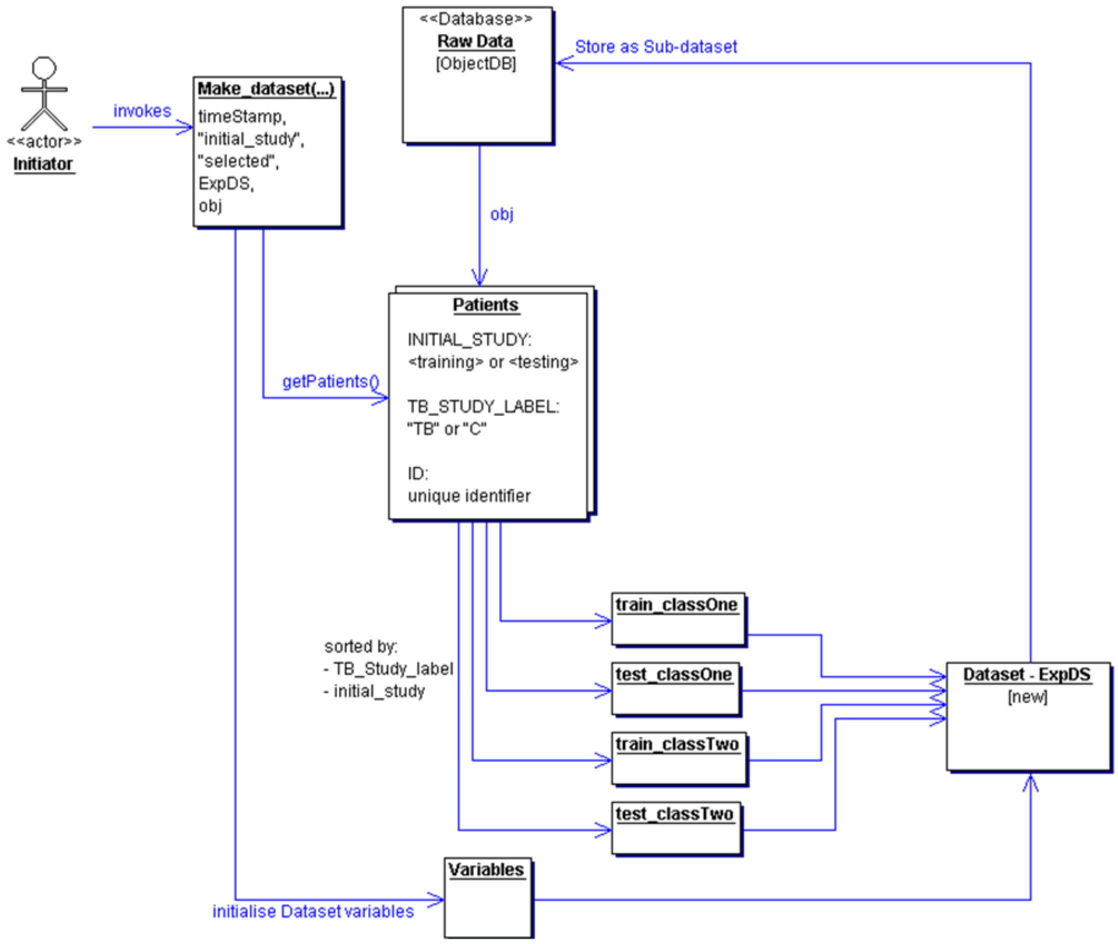


FIGURE 3.7: Initialisation of the *Initial Study* dataset.

Various other experimental sub-datasets are created. For example, a set of four *TB* vs *Control* datasets are generated in a process similar to that which created the initial study above, which can be used to look for patterns for *TB* prediction. For the investigation into the link between tuberculosis and HIV, the datasets can be prioritised using only those two variables. In addition, other features can be selected or deselected depending on what is being investigated and what data can be fed into the various machine learning techniques. The user can specify exactly what kind of sub-datasets are required using the *performQuery* method. This method accepts two (or more) queries in order to generate the various sub-datasets.

Biomarkers play an important role in the case study and are handled in a similar manner to the other data inputs. A biomarker is any kind of substance that can be used as an indicator of a biological state, namely the existence, past or present of a living organism. Biomarkers can also be used as an indicator of a particular disease. The presence of a certain antibody, identified by the biomarker, may indicate the presence of a disease but more importantly, a biomarker can indicate a change in expression or state of a particular protein associated with a particular disease. This type of investigatory work is very important during both the diagnosis and prognosis of many diseases.

In this case study, biomarkers are imported via a tab-separated text file. The biomarker headers are added to the case study metadata as an indication and record of the biomarkers that have been used and the biomarker values are added to the individual patients. An illustration of this process can be found in figure 3.8.

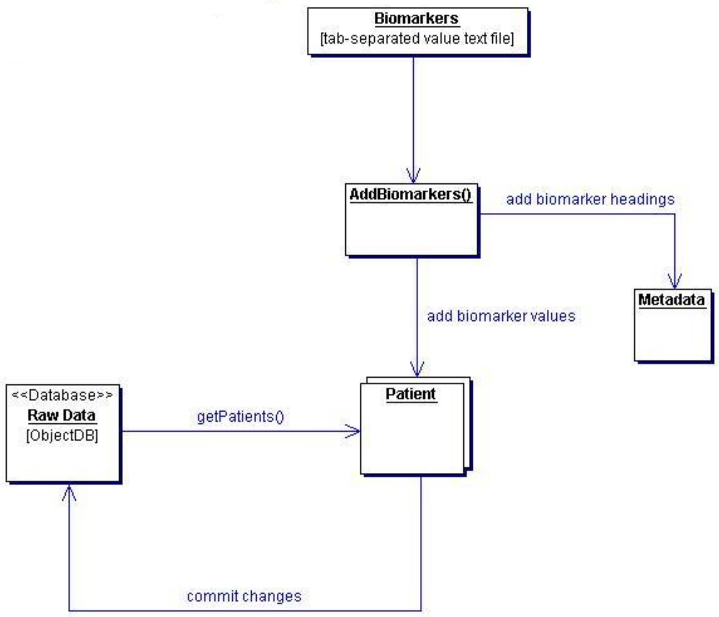


FIGURE 3.8: The initialisation and addition of biomarkers.

3.2.3 Experimentation

The experimenter begins experimentation by running an initiating run command from Matlab. The experimenter passes a single integer variable which represents the type of experiment that is to be initialised. The types of available experiment in this case study are pre-specified, although more can be added depending on the nature of the required experimentation. The variables for the experiment are initialised and, depending on the experiment, the corresponding *run* procedure is called.

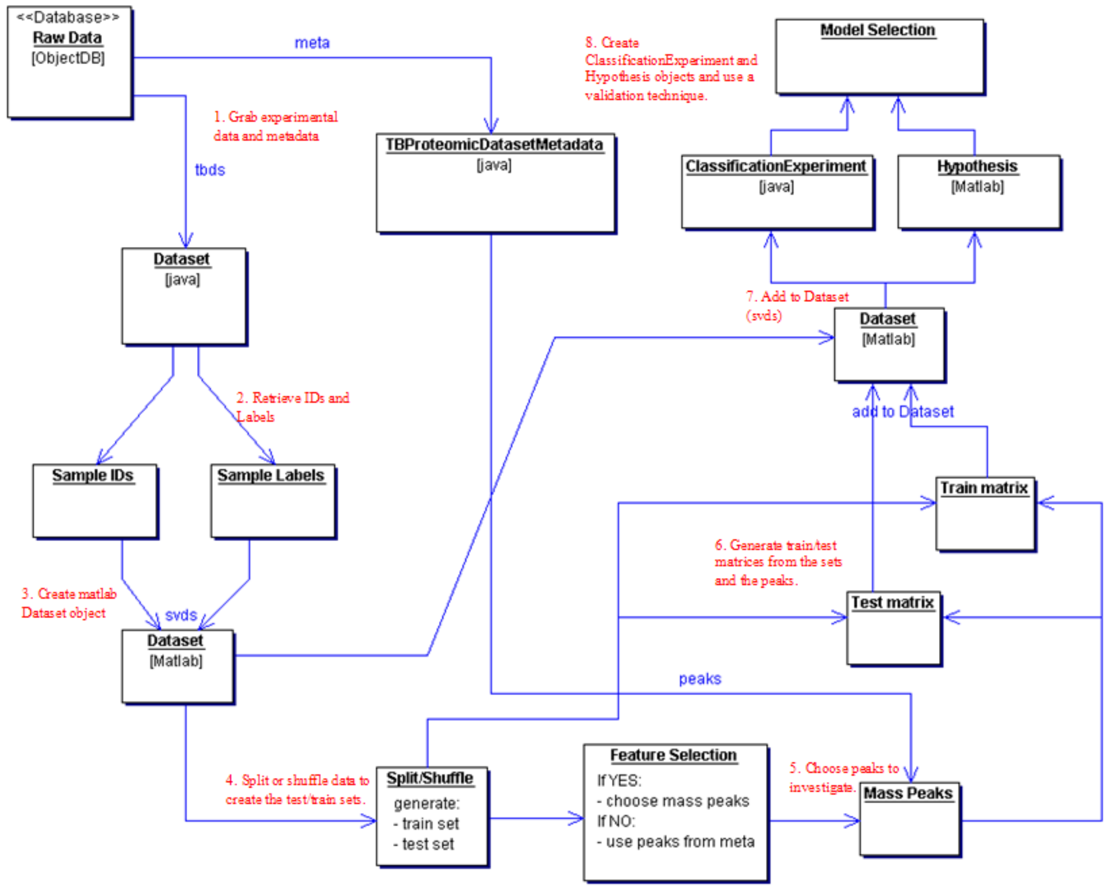


FIGURE 3.9: The processes involved in running a basic experiment.

The following is a description of the processes illustrated in figure 3.9. The relevant dataset (specified in the matlab function, RUN) is retrieved and split in sample IDs and labels. These are used to create a matlab object called *dataset*. The *dataset* is then split or shuffled according to pre-set variables to generate the train and test sets. If the experimenter has enabled feature selection, the mass peaks of those features are selected, otherwise the mass peaks of all the features are selected from the metadata. The mass peaks, together with the train and test sets are used to generate train and test matrices, which are in turn added to the matlab dataset. This dataset object is used to create a *ClassificationExperiment* java object and a *hypothesis* matlab object. These are then passed to a validation process.

3.3 The Case Study Analysis

The case study analysis falls into two main areas; an identification of the possible sources and sites where change can occur and explicit description of the impacts that the identified changes can bring about. The results of the analysis are described in section 3.4.

3.3.1 Sources of Change

We are dealing with a system that exhibits a high degree of change over a variety of levels. There are multiple points of possible change and there can be various consequences as a results of those changes.

The change and impact analyses are based around the schematic analysis of the working example carried out and described in the previous section. The changes were identified using a working knowledge of the example and postulating the presence of possible changes within the system. The change analysis documents the points at which changes can occur within the system. It describes where the change occurs, the actor responsible for the change, the reason for the change and provides a real-world example.

The purpose of the change analysis was to investigate the areas of potential change and link those changes to a real-world example. It is not sufficient to merely claim that the changes occur and that we must deal with them. The changes must be identified, located, described and rooted within a real context. Only then can we begin to hypothesise solutions to the problems.

3.3.2 Impact Flow

Having identified the sources of possible change, we can look at each change and analyse the impacts that the change may have on the system and its component parts. This was completed using the analysis of the working example together with the change analysis to hypothesise the possible impacts that each change may have.

The impact analysis examines each source of change in the change analysis and describes the impacts on the system and its actors as a result of those changes. The impacts are measured in five categories[5];

Data Impacts: The impact to the values and/or structure of the data.

[5] It is important to note that, unlike the previous reports, the impacts report is not limited by the limitations of the system that is being analysed. For example, a change that should impact the system but doesn't due to an implementation limitation is considered nevertheless.

Metadata Impacts: The impact to the system metadata.

Performance Impacts: The impact to the performance to the system after as a result of the change.

Safety/Security Impacts: The impact on safety or to any security policies as a result of the change.

Resource Impacts: The computational, financial or human cost incurred as a result of the change.

3.4 Case Study Analysis Outcomes

Within the change analysis, we identify four types of change that occur within the case study; content, semantic, protocol and experimentation. These are described in the following sections.

3.4.1 Content

Data can change for many reasons and this section refers to changes to the values of the specific data points.

Over time, new data is added to existing databases. This is especially true for large, publicly available databases where data is continually added. These additions are not always reflected immediately as the release of updated versions remains an administrative issue. Depending on your experimental sub-datasets and the nature of your queries, these unannounced additions may or may not affect your results. It is important to note that there are generally no semantic links between previous versions of the database. Moreover, it is not possible to distinguish what results, if any, have been added. In a similar fashion to added data, there may be data that is re-examined or refuted and corrected. These kind of changes usually occur with no warning or explanation.

The impact of the addition of new data depends almost entirely on the queries used to generate the experimental datasets. If the added data does not fall within the limits of the experimental query, then clearly there will be no impact on the experiment. Consider a scenario where the experimenter may wish to include the additional data because of its importance even though it falls outside the limits of the original experimental query. In such a case, it may be necessary to widen the experimental query and the implications and impacts of this action should also be considered.

Some samples from the data may become invalidated, either through external research or via local experimentation. Either way, these data samples are usually removed to allow valid experimentation to continue. In an ideal scenario, these removals or alterations are fed back into the database higher up, although this is certainly not usual practice. Subsequent versions, therefore, will continue to contain the invalidated samples and the process of removing them must be repeated.

By modifying the source data, the results that have been achieved through the use of that data immediately become out of date. This doesn't necessarily mean that the data has been sufficiently impacted be considered defunct, moreover it does not reflect the most current state of the source data. The experimenter may have performed many experimental iterations with the source data generating complex interactions between multiple sets of results and data objects. Any data that relies, either directly or indirectly, on the source data is potentially out of date.

One important problem with this type of change is that modifications are not reflected in the metadata; there is no way of detecting whether a change has occurred. This effect is lessened somewhat in the case of a collaborative data source where changes initiated within the group can more easily communicated. Details of the modified results and the reasons behind the modifications should be available. This is aided further due to the fact that collaborative data sources are often much smaller and, therefore, it is easier to keep track of changes.

When considering the case of automatic disease diagnosis, any modifications of the data can present significant concerns to the validity of the experimental model. If the data has been modified in order to correct errors, that may have an effect on the existing results and any diagnoses that have previously been made. In such a case, it may be very important to re-run the previous experiments in order to reflect the current state of the data source, regardless of the cost and effort involved.

The initiator of the experimental datasets may be called upon to either create additional experimental sub-datasets or alter an existing sub-dataset. The nature of the query to create this dataset carries an effect on the following experimentation. During the alteration of an existing sub-dataset, presumably to fix a problem with an experiment based on that dataset, care must be paid in assessing the impact on any other dependent experiments. If an effect is established, a new sub-dataset is preferred over an alteration. The impact of the creation of sub-datasets is difficult to assess. There is no way of automatically creating experimental sub-datasets and, therefore, the process requires some human interaction. The level of this interaction depends on the nature and complexity of the experiment as well as the availability of personnel who are qualified and familiar enough with the experimental process to create the datasets.

3.4.2 Semantic

Semantic changes refer to a change that is semantic in origin such as the modification of data headings or the parameters of experimentation. Any kind of database schema change is encapsulated in this section.

There may be good reason to alter the properties of an existing experiment in order to correct an error, update or change a parameter or to alter the functionality or hypothesis of the experiment. For example, the matlab **run()** file is an executable that contains a long list, a *catalog*, of experiments from which the experimenter chooses the desired experiment. Each experiment has its own set of parameters which are enabled during initiation of the selected experiment. It may be necessary to change some of these, either to correct an error or for experimental tweaking. In either case, the change will affect future instantiations of that particular experiment and the experimenter must make sure that the correct, or desired, parameters are in operation for each experiment to be executed.

A change in the experimental parameters denotes a change in the way data is processed. A change in the experimentation process will often generate an effect on the data or results. As a consequence, experiments that have subsequently been initiated using those results will need to be re-run in order to reflect the changes. The metadata for any modified experiment and for any associated results will be immediately affected. Also, this metadata should be used to track the impact of the modified process to all results that have been spawned from that process.

There is often a significant human and computational effort in recomputing results and this plays a large part in deciding whether to propagate the impact. The most important aspect, arguably, is the time and effort required by an actor to recompute. This human and computational effort must be taken in consideration just like any other regular impact.

During experimentation, results may yield new discoveries, generating new or refined hypotheses and creating new direction for experiments and research. These new discoveries may require refinement of the experimentation and may possible require the investigation of additional facets of the existing data. These additions will alter the schema of the dataset, affecting the functionality of the existing dataset-dependent experiments. For example, a link between tuberculosis and pneumonia may be suspected after analysis of some recent experimentation. It may be called for the collaborative scientist to revisit the data demographic (in this case, the patients) and test for pneumonia and then add it to the raw data for the experimenter to incorporate into the experimental process. We have to examine the effects of such an action to the experimental environment.

Databases are queried in order to generate experimental datasets. These datasets are fed into experiments to generate results. In reality, the data is fed into the experiment, usually line by line. The data lines are then internalised within the system and experimentation continues. Under this regime, it makes no difference to the order of the lines of data supplied. When we consider implementing a system to detect changes and, therefore, versions of a dataset, it may cause considerable difficulty if the dataset were reorganised. This is especially true if the order of both the columns and the tuples are reorganised. It should make no difference to the experimentation, as exactly the same data is being used, but it can pose a significant problem when trying to detect whether the datasets have changed.

3.4.3 Protocol

A protocol change is generated from the running and operation of the system. This is not a change to either the data or the implementation of that data. It refers to the operation of a part of the system that may have an effect on another part of that, or related, systems.

There may be cause for an actor to alter their working environment so as to cause a change to the way data is transferred to other users of the system. As a simple example, it may be more productive for the scientist to produce comma-separated variable files rather than Microsoft Excel files. In such a case, this change would have to be announced to the experimental community so that it can be accommodated for, ideally before the change takes place. The smooth functionality of the system relies on each part having full knowledge of how it interacts to the other parts of the system. Any changes to these interaction should be announced in a proper and timely fashion in order to enable the successful continuation of the experimental process.

It is unlikely that changes in policy will directly affect existing results. An explicit requirement of a policy change should always be that it does not directly affect the content of the data. A policy change involving the data format, however, could require a propagation of that policy to existing results and this could cause substantial effort. Any change to the policy is likely to be reflected in the metadata enabling policy changes to be tracked throughout the system. It is important to consider other aspects of a policy change. The performance of a system can be impeded or even broken by a policy change if that change affects some previously unaffected area. For example, a policy change adding extra levels of authentication could theoretically cause a system to pause for a password, perhaps indefinitely if not picked up. This is especially problematic if the policy change is unannounced.

Protocol changes, such as the example described above, are generated and governed internally within the experimental process. Other changes to protocol are conceived externally and affect the experimental environment without warning. For example, it may become necessary for the external administrator to alter the policy regarding the access of the database. This may include changes to data formats, version-release schedules, data-inclusion policies and metadata. Such changes occur, not frequently usually but with a certain regularity that requires local management.

From time to time, it may become necessary to update, maintain or debug the experimental code in order for it to remain efficient and up-to-date with new, emerging technologies. Theoretically, this should not affect the external interface for the experimental methods, as the process is being improved internally. It may, however, alter the way the experiments are performed internally and this may have affects on the results that would be very difficult to estimate, especially if the experiment uses third-party or sealed processes.

Code maintenance can have varied performance-related effects. Most maintenance routines aim to improve some factors of the code, some of which focus on maintainability. Efficient maintenance can be an extremely difficult process, especially if it is done properly. The higher the emphasis on maintenance factors such as portability, readability and low-coupling, the easier and more desirable future maintenance becomes. The difficulty in maintaining software usually relates directly to the time spent ensuring the code is easy to maintain.

Any system that offers safety-critical operations, an automated diagnosis tool for example, must be thoroughly tested before use. Any bugs or errors must be exposed and corrected before the system is used to diagnose real patients. When the code for such a system undergoes maintenance or is changed in any way, it must be retested to the same extent as before to make sure that maintenance has not affected the system durability. Not only does this make maintenance undesirable but enforces the requirement to manage any possible change correctly.

3.4.4 Experimentation

An experimental change occurs as part of the experimentation process such as the addition of experimental technology or the addition or removal of an existing experiment.

Over time, revelations and discoveries from results will, ideally, generate new hypotheses and propel research in directions. This will likely require the addition of new experiments to test these hypotheses. Each new experiment will have its own unique parameters and metadata. The emergence of new technology, once feasible to include, may require the building of new experiments.

Generally, the effort required to add a new experiment is small as it does not affect the operation of the existing experiments. It is often the case that new experiments differ from existing experiments by parameters only. In this case, impact is low. If the new experiment requires new technologies, the impact is greater. This obviously precludes the effort involved in conceiving the new experiment.

There may be new machine learning techniques emerging that the experimenter would wish to implement within the experimentation process. This could involve a re-writing of some parts of the process-centric code. Furthermore, the experimenter will likely wish future experiments to reflect these changes in implementation, especially if the change is a result of a bug fix. In that case, it is desirable to rerun the previous invalidated experiments using the new implementation.

Theoretically, the incorporation of new technology should not immediately affect the resulting data. This depends entirely on the nature of the improvement, whether it be a process or semantic improvement. The semantic impact, however, is more considerable and depends on what metadata is being currently recorded. Changes and updates in the code that do not reflect change in the data or the experimental process are unlikely to impact the metadata.

Chapter 4

Hashed Data Model

4.1 Introduction

There is little point spending time and effort to develop a model if it turns out that it is based on data which is out of date. Many models require large amounts of data from a variety of heterogeneous sources. We have already suggested that the integration of this data is a significant challenge, backed up by much research. But furthermore, this data is subject to frequent and unannounced changes. It may only be possible to know that data has fallen out of date by reconstructing the model with the new data but this leads to further problems. How, when and why does the data change and when does the model need to be rebuilt? At best, the model will need to be continually rebuilt in a desperate attempt to remain current. At worst, the model will be producing result that are out of date[1]. The **Hashed Data Model** (**HDM**) was conceived to manage this state of data change.

Systems biology focuses on understanding the interactions between biological systems through the use of high-throughput analysis, computational modelling and experimentation. Large amounts of complex data from multiple, heterogeneous sources are obtained and integrated using a variety of tools. These data and tools are subject to frequent change, much like other biological data. Reconciling these changes, coupled with the interdisciplinary nature of systems biology, presents a significant problem. We require a system that can identify changes and, if possible, describe the impact of the change on results that have been previously computed.

The recent advent of automated and semi-automated data-processing and analysis tools in the biological sciences has brought about a rapid expansion of publicly available data.

[1]A model that produces results that are erroneous as a result of being out of date, can be regarded as being invalidated.

Many problems arise in the attempt to deal with this magnitude of data; some have received more attention than others. One significant problem is that data within these publicly available databases is subject to change in an unannounced and unpredictable manner.

Consider a systems biologist building a model and conducting experiments '*in-silico*'. In the event that the data used has changed, it may be important that the biologist be made aware of this change in order to establish the impact on the results they have previously attained. Couple this change with the experimental, protocol and other changes that can occur, it is clear that there exists a complex environment of possible changes that can affect the scientist's results. Managing this change in a way that can benefit the experimenter is a considerable problems. Some results may be more important than others, may require more effort to repeat or may be less effected by certain changes than other sets of results.

Chapter 4 presents the HDM in order to help e-scientists identify, track and manage these types of change. The HDM is a tool that abstracts an experimental dataset and metadata to a model that can be used to detect changes between versions of datasets. The characteristics of the HDM allow the user to potentially retain a unique version of the dataset for every single experiment.

4.2 Requirements for the Hashed Data Model

We have developed the Hashed Data Model (**HDM**) as a prototype to demonstrate a first attempt at the versioning and tracking of change within a dataset. The **HDM** is specified according to a set of requirements elicited through an analysis of the case study, described in section 3. Given the prototypal nature of the **HDM**, the requirements were not specified formally, or to the same extent as the **ExperimentBuilder**, and they are presented below. We cover only the basics here; we describe the way the **HDM** is intended to interact with the user, the main features of the system, some of the non-functional requirements and some of the data characteristics of the **HDM**.

4.2.1 System Overview

We start by identifying the users of the system. As a prototype, we intend the **HDM** to be used primarily by scientists who are interested in developing a change management framework. In this respect, the significant secondary purpose of the **HDM** is the elicitation of further requirements for a future change management application. The **HDM** is designed to be used by one scientist a time and there is no requirement for concurrency within the **HDM** at this point in time.

We shall start with a brief *scenario* in order to identify some of the important *use cases* associated with the **HDM**. This scenario deals with the basic concepts of generating and comparing **HDM** text files[2].

> The scientist starts the **HDM** application. They enter their unique user name and password and provide a brief description of the dataset that they intend to use with the **HDM**. They also enter the filename of the dataset. The scientist enables the **HDM** to use *fine-grained checking* and provides a *tracking factor* of 5. They then click on the button, 'Generate HDM'. the **HDM** verifies the username and password against an internal list and, if verification is successful, returns to the user a textual representation of a "Factor 5 HDM" text file. The scientist saves this file to their personal repository.
>
> The scientist then brings up the *HDM Comparison* screen. There are two text boxes. In the first, the scientist enters the filename of the **HDM** text file they have just generated. In the second, they enter the filename of the **HDM** text file of the same dataset that they generated 6 months ago. They click on the *Compare* button. The **HDM** returns a report of this comparison detailing the differences between the two **HDM** text files. The scientist digests the report to see whether, in the last 6 months, the dataset has changed, how much it has changed and where these changes have occurred.

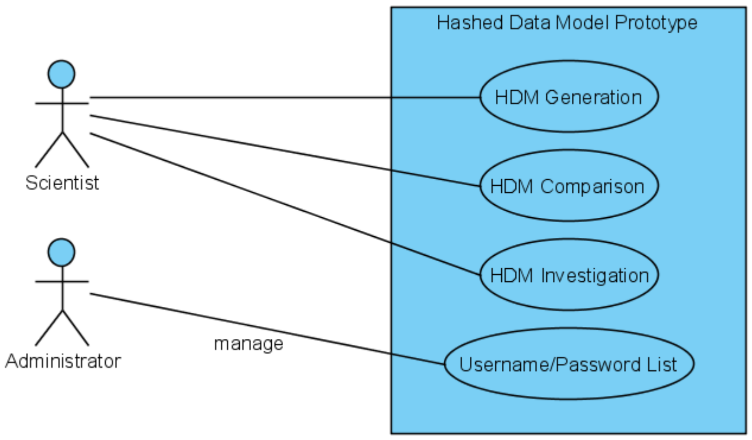


FIGURE 4.1: A use-case diagram for the HDM prototype

[2]The **HDM** text file refers to the textual representation of a dataset version, generated by our application.

From the scenario above we can build a use-case diagram and this is illustrated in figure 4.1. From figure 4.1, we can see there are four distinct use cases and these are described below.

HDM Generation

1. Scientist starts a browser session.
2. Scientist navigates to page housing the **HDM prototype**.
3. Scientist enters username and password and, optionally, provides a description of the dataset being used.
4. Scientist chooses whether to enable fine-grained investigation.
5. If fine-grained investigation is enabled, then the scientist enters a tracking granularity value.
6. Scientist hits *Generate HDM* button.
7. If the username and/or password are incorrect or cannot be found within the internal user list the **HDM Prototype** will return a null result.
8. Providing the username and/or password can be found in the internal user list, the **HDM Prototype** displays a textual representation of the **HDM**.

HDM Comparison

1. Scientist starts a browser session.
2. Scientist navigates to page housing the **HDM prototype**.
3. Scientist navigates to the section labelled *"HDM Comparison"*.
4. Scientist chooses the **HDM** text file of the original (older) dataset.
5. Scientist chooses the **HDM** text file of the current (new) dataset.
6. Scientist hits *Compare HDM* button.
7. **HDM Prototype** displays a textual representation of the comparison of the two **HDM** text files.

HDM Investigation

1. Scientist completes either a *HDM Generation* or *HDM Comparison* scenario.
2. Scientist clicks on the section labelled "**Investigate HDM**".
3. Scientist can investigate the various properties of the **HDM** within a tree-like structure and can drill up or down.

Username/Password List Management

> The trigger for this use case is a change of operational requirements for the **HDM** relating to either the addition or removal of a valid user of the system. It is not worthwhile to examine the steps required in such an action as there will be no direct functionality in the **HDM Prototype** to facilitate these actions. It is important to recognise only that the **HDM Prototype** contains a list of approved users and passwords and the occupants of this list are managed by an administrator. Currently, there are plans for only one level of security; i.e., a user has access rights or does not.

The **HDM Prototype** is required to provide access to users, primarily scientists, who often do not have extensive computer experience. In order to successfully use the **HDM Prototype**, users will require a basic familiarity with computers, involving but not exceeding basic web browsing and file management skills. There is no platform-specific experience required to use the **HDM Prototype**.

4.2.2 Functional and Non-functional Requirements

Under normal conditions within a requirements specification, functional and non-functional requirements would be separated and discussed in separate sections. For the sake of simplicity, we will discuss all the requirements together in this section. As previously mentioned, the requirements for the **HDM Prototype** were not formally specified and rather than presenting them formally, we discuss them here in order to provide a concise, clear view of what we wanted to achieve with the **HDM Prototype**.

At the most basic level, there is the requirement to provide the functionality described in the use cases in section 4.2.1. The use cases primarily elicit the functional requirements for a specification and we shall describe some of them briefly here.

HDM Generation

Prerequisites The user is connected to their service provider and can navigate to the **HDM Prototype**; the user has a dataset (in the correct format) to generate the **HDM** text file.

User Input The user enters (or browses for) the filename of the dataset for which they wish to generate the **HDM** text file. The user can then decide how finely they wish to measure the submitted dataset. The degree of *granularity* with which they measure the dataset during the generation phase defines how accurately the **HDM Prototype** can measure changes to the datasets during the *HDM Comparison* phase. We will have a numerical *granularity factor* to allow the user to specify how finely they wish to measure the dataset. At this point, the user will hit the "*Generate HDM*" button.

Response The **HDM Prototype** returns the **HDM** text file to a window in the web browser and the user can choose to save it to their personal repository. If the user chooses to save the **HDM** text file, the filename is added to the "*compare-to*" text box of the *HDM Comparison* section of the **HDM Prototype**[3]. As well as a text form, the generated **HDM** is presented to the user in a *drillable* form within a table to allow the user to investigate the HDM on a more in-depth, intuitive level.

HDM Comparison

Prerequisites Again, the user is connected to their service provider and can navigate to the **HDM Prototype**; the user must have, as a minimum two **HDM** text files to compare to one another. If the user has recently generated and saved a **HDM** text file, that filename is automatically placed in the "*compare-to*" text box of the *HDM Comparison* section of the **HDM Prototype**.

User Input The user enters the filenames of the two **HDM** text files. The user has the option of employing a comparison that takes advantage of any *fine-grained investigation* measures that have been enabled on either of the **HDM** text files[4]. The user then clicks on the "*Compare HDMs*" button.

Response The **HDM Prototype** returns a report detailing the differences between the two **HDM** text files. The accuracy of this report will depend on the granularity of the two text files. This report is available for the user to save for their own records. Similarly as for *HDM Generation*, the comparison is also available to the user as a *drillable*, structured report.

[3]It seems reasonable to assume that if a user has just generated a **HDM** text file and then wishes to perform a comparison immediately after, they will want to compare some previously generated **HDM** text file (the original) to this one (the current).

[4]It may be necessary to de-select the option to perform a *fine-grained* comparison, depending on how finely each of the **HDM** text files have been generated

HDM Investigation/browsing

Prerequisites The user must have at least one **HDM** text file they intend to view.

User Input User either conducts a *HDM Generation* or a *HDM Comparison* or enters the file name of the **HDM** text file they wish to view, whereupon the user clicks on the "*View HDM*" button.

Response The **HDM Prototype** returns the generated **HDM** to the user in a *drill-able*, structured form within a table.

There is an assumption inherit in almost all the use cases described above. We intend to use a web platform to distribute the **HDM Prototype** either through a password-protected Internet website or via a private laboratory repository. As we are building a prototype, it in unlikely that we will have many users at a time and we are, therefore, not including specific concurrency functionality[5].

Using a browser-based service also manages the problem of hardware constraints, at least on the part of the user who requires only a platform capable of running a browser. The computation of the **HDM Prototype** takes place at the server so what of the server requirements? It is important that the prototype does not require any kind of special computing requirements in order to run. Fortunately, we do not foresee any problem in this regard as the weight of computation is relatively low and should be ably handled by any household desktop computer.

The **HDM Prototype** requires the presence of either a dataset, for *HDM Generation*, or a **HDM** text file, for *HDM Comparison.* Needless to say, without either of these things, there is no way of interacting with the **HDM Prototype**. There is no persistence within the **HDM Prototype**; when a user completes an interaction with the **HDM Prototype** and closes the browser window, their session is lost and any data in use is discarded. We include this requirement in order to deal with one important aspect of security. Users of the **HDM Prototype** are working with various types of data, mostly biological or medical. Data of these types is invariably of a highly sensitive nature and, therefore, subject to increased security. In order to persuade scientists to upload their highly-sensitive data to our web application, something some will be unwilling to do and most will be wary of, we must assure them that their data is secure. We will achieve this by not retaining any kind of persistent state of the data within the **HDM Prototype**. The data is uploaded, parsed and the **HDM** text file is returned to the user, whilst the original dataset is immediately discarded by the **HDM Prototype**.

[5]Methods dealing with concurrency may be present in the **HDM Prototype** due to added functionality from the developmental **IDE** (Integrated Development Environment).

The **HDM Prototype**, as its name suggests, was conceived to be a relatively simple, light-weight application designed to be implemented as quickly as possible. As a result, there is limited importance placed on user facilitation and the visual aspects of the project. From the identified use cases, we can define some requirements for the **HDM Prototype**. *HDM Generation* and *HDM Comparison* are distinct operations to be implemented within the boundaries of the **HDM Prototype**. We can, therefore define two distinct sections of the **HDM Prototype**, navigable by the user.

4.3 Building a Data Model Using Hashes

Given that the data changes with a frequency and strength that requires management of those changes, we can conclude some requirements for a data change management framework. Firstly, we should be able to detect whether a change has occurred or not within the dataset. Secondly, we must be able to conclude the significance of the detected changes. Some changes will have greater impact than others and it is important to measure this impact in order to decide how to manage the change. Thirdly, we would like to know the location of the change, adding to the desiderata for the significance of the change and providing provenance information for the dataset.

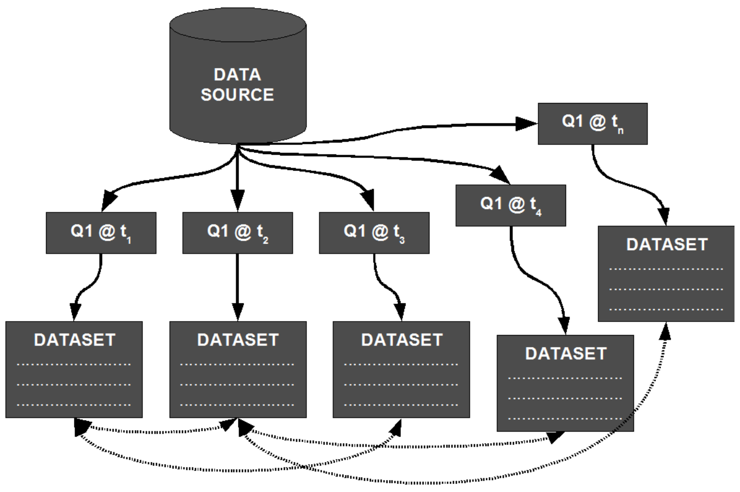


FIGURE 4.2: Query 1 over five time periods on the same datasource.

Figure 4.2 illustrates the querying of datasources to obtain experimental datasets. A dataset is created by querying the data source at t_1 with *Query 1.* At some later time, t_2, we use the same query to create another dataset. We would like to know whether, during t_1 and t_2, the dataset has changed. This appears to be a straightforward problem but when we consider that from a single dataset we may possibly have thousands of experiments, each conducted at a distinct time, we face the possibility of managing many versions of the same dataset. There may be many experiments querying a single datasource, and therefore generating multiple versions, over a range of time periods. There is considerable overhead in administering and maintaining these dataset versions as well as providing analysis between them. The storage overhead for n versions of a dataset of x bytes in size can be represented as xn. The computational complexity for comparing n versions can be represented by the triangular number T_n[6].

$$\sum_{k=1}^{n-1} k$$

$$= \tfrac{1}{2}n(n+1)$$

$$\text{(an } n^2 \text{ problem)}$$

In order to detect differences within the datasets, we must compare one dataset to another. Given the high number of experiments that can come from a single dataset, it is possible that we may have to consider many versions of the dataset at any given time. It is, therefore, not feasible to store a complete version of the dataset for each experiment; consider a dataset with 10,000 dependant experiments. Are we to store the precise details of the dataset used in each experiment, regardless of the differences between these experimental datasets. An abstraction of each version of the dataset could be made from which to detect change.

Hashes could be used to model the datasets. A hash is a way of creating a small digital fingerprint of an item of data. Successful hash functions typically have two important characteristics. The hash function should be deterministic. The same item of data, if unchanged, should always produce the same hashed value. Secondly, any change in the data will be very likely to produce a different hashed value. The ideal hash function has very low collisions although these can not be removed completely.

If we apply a hash function to the entire dataset, we can look for any changes in the dataset. A change in the hash value will depict a change, however small or insignificant. This model will allow us to detect changes to the dataset but provides no information about the size, location or impact of the change. The problem lies with the nature of hash functions, which have been primarily developed for security purposes. It is generally considered to be a positive aspect of a hash function to exhibit entirely different hashes for data items, regardless of their degree of similarity. This aims to reduce the feasibility of a successful comparative analysis attack on the hashed value. For the purpose of the

[6]More formally, a triangular number is a number obtained by adding all positive integers less than or equal to a given positive integer n.

HDM, we require a comparative hash function such that the degree of similarity among datasets is mirrored by the similarity of the hash values.

By selecting important aspects of the data and recording their relationships, we can build a model using nodes and relationships as illustrated in figure 4.3(a). Figure 4.3(a) demonstrates how the various aspects of the dataset are used to form the HDM model[7]. A textual representation of the HDM illustrated in figure 4.3(b) corresponds to the graphical HDM representation in figure 4.3(c). In figure 4.3(c), each node of the graph represents a hash value contained within the HDM. The structure within the HDM is hierarchical so the hash value of each node is constructed using the node itself including any children of the node. We can now detect the aspects of the data that have changed and, equally as importantly, identify those aspects that have been preserved. Upon detecting a change, owing to the hierarchy of the HDM, we can also pinpoint the node from which the change originates.

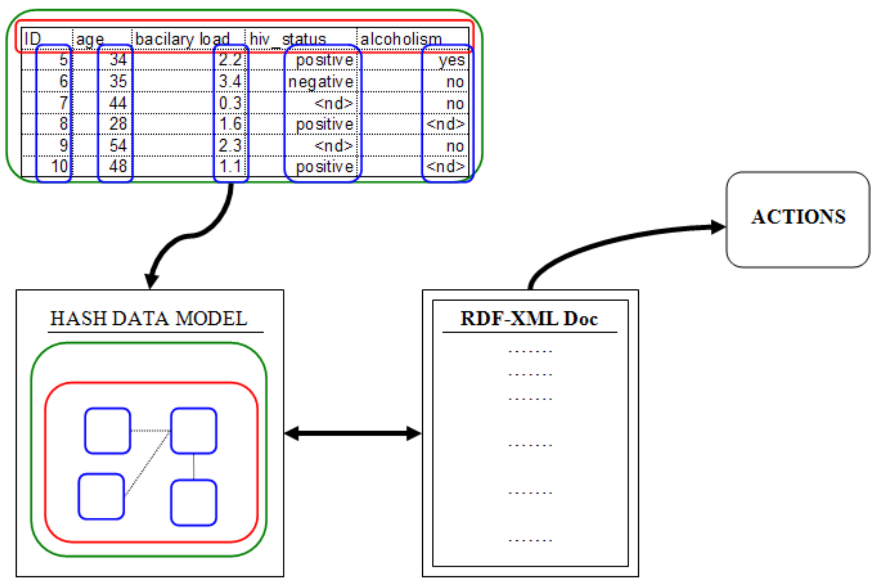
We present the algorithm used to generate the Hashed Data Model below. This algorithm was subsequently used within the HDM Prototype.

[7]The HDM illustrated is only an example of a possible HDM.

The HDM Building Algorithm:

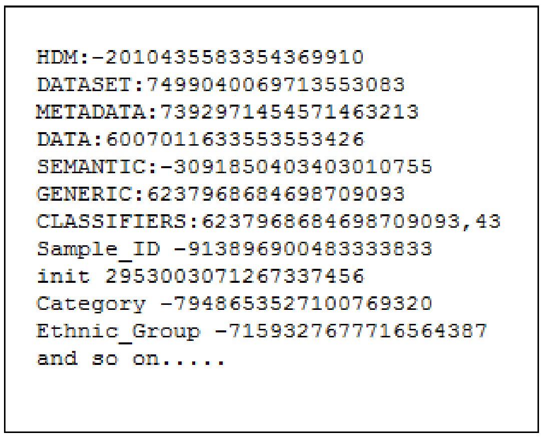
```
begin proc HDMBuild($md, dataset, gran$) $\equiv$
        $hdm := newHDM();$
        comment: Add generic metadata (hashed).
        for $each(metadata : md)$ do
            $hdm.addMetdata(Hash(metadata));$ od
        comment: Add dataset (hashed).
        $hdm.addDataset(Hash(dataset));$
        comment: Add hashes of columns.
        for $each(column : dataset)$ do
            comment: Create a new Column().
            $col := newColumn();$
            comment: Add the column (hashed).
            $col.setHash(Hash(column));$
            for $i := 1$ to $col.rowNumber()$ do
                $varcounter = 0;$
                $varcolString;$
                comment: Add a number of rows according to the granularity.
                while $counter \neq gran$ do
                      $colString+ = row;$
                      $counter + +;$
                      $i + +;$ od
                $col.addRowBlock(Hash(colString));$ od
            comment: Add the Column to the dataset of the hdm.
            $hdm.getDataset().addColumn(col);$ od
        **return:**$hdm;$
  .
end
```

It is important to note that the HDM is primarily concerned with the detection of changes in datasets. The latest instantiation of the prototype provides some tracking and impact estimation. The HDM is not a complete solution for managing changing biological data but rather provides the first step for such a framework.



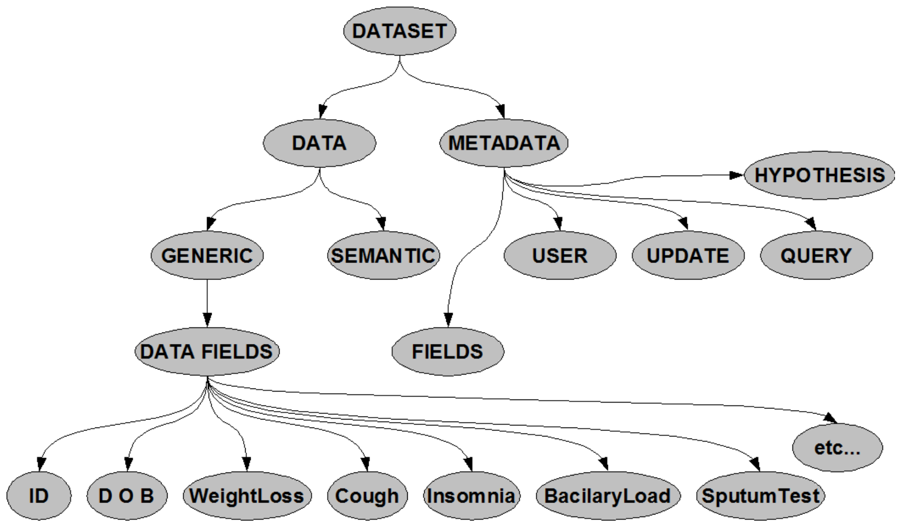
ID	age	bacilary load	hiv_status	alcoholism
5	34	2.2	positive	yes
6	35	3.4	negative	no
7	44	0.3	<nd>	no
8	28	1.6	positive	<nd>
9	54	2.3	<nd>	no
10	48	1.1	positive	<nd>

(a) Generation of HDM

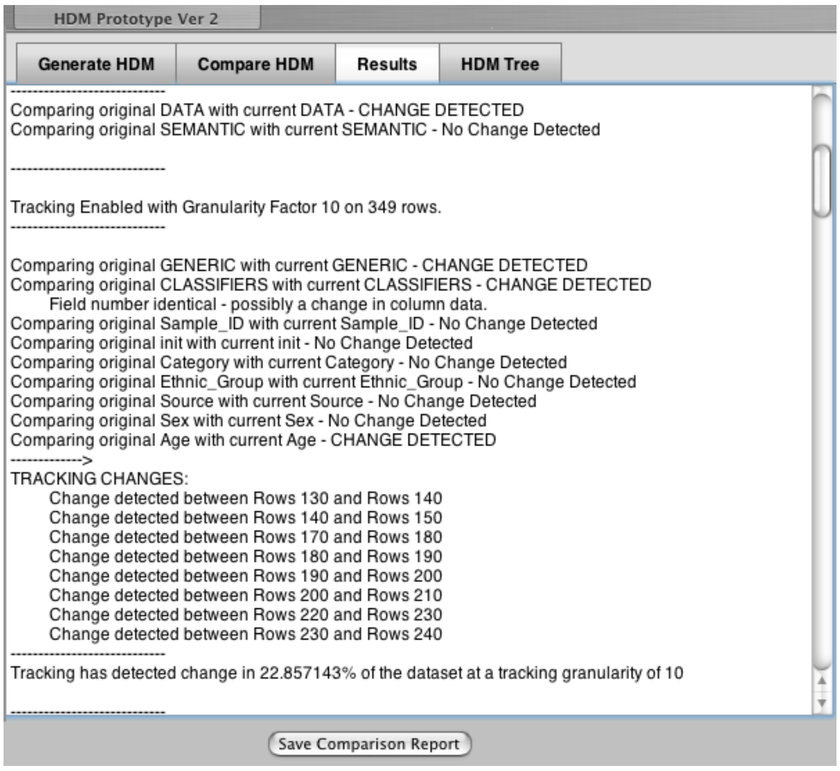


```
HDM:-2010435583354369910
DATASET:7499040069713553083
METADATA:7392971454571463213
DATA:6007011633553553426
SEMANTIC:-3091850403403010755
GENERIC:6237968684698709093
CLASSIFIERS:6237968684698709093,43
Sample_ID -9138969004833333833
init 2953003071267337456
Category -7948653527100769320
Ethnic_Group -7159327677716564387
and so on.....
```

(b) HDM Text File



(c) HDM Graph



```
----------------------------
Comparing original DATA with current DATA - CHANGE DETECTED
Comparing original SEMANTIC with current SEMANTIC - No Change Detected

----------------------------

Tracking Enabled with Granularity Factor 10 on 349 rows.
----------------------------

Comparing original GENERIC with current GENERIC - CHANGE DETECTED
Comparing original CLASSIFIERS with current CLASSIFIERS - CHANGE DETECTED
      Field number identical - possibly a change in column data.
Comparing original Sample_ID with current Sample_ID - No Change Detected
Comparing original init with current init - No Change Detected
Comparing original Category with current Category - No Change Detected
Comparing original Ethnic_Group with current Ethnic_Group - No Change Detected
Comparing original Source with current Source - No Change Detected
Comparing original Sex with current Sex - No Change Detected
Comparing original Age with current Age - CHANGE DETECTED
------------->
TRACKING CHANGES:
      Change detected between Rows 130 and Rows 140
      Change detected between Rows 140 and Rows 150
      Change detected between Rows 170 and Rows 180
      Change detected between Rows 180 and Rows 190
      Change detected between Rows 190 and Rows 200
      Change detected between Rows 200 and Rows 210
      Change detected between Rows 220 and Rows 230
      Change detected between Rows 230 and Rows 240
----------------------------
Tracking has detected change in 22.857143% of the dataset at a tracking granularity of 10

----------------------------
```

FIGURE 4.3: Tracking output via the **HDM** prototype console.

4.4 The HDM Prototype

The HDM prototype is a simple, light-weight, generic web-service that demonstrates the workings of the Hashed Data Model. It was designed using Java Studio Creator and can be deployed as a WAR file on a variety of Tomcat or Java's SunAppServer 8 servers. The purpose of the prototype is to demonstrate the workings of the HDM. Although the initial stimulus for the HDM prototype was to provide change management for biological and life sciences data, the HDM has been designed to accommodate data of all types. Input is limited only by the format of the dataset. The prototype accepts data in a tab-separated format with each row beginning a new line. The HDM works by detecting changes in specific columns so it assumed that the first line contains the column names, otherwise default column names are used which hold no semantic meaning.

The user submits the dataset with some sample metadata and the web service returns a textual representation of the Hashed Data Model. An extract of this textual representation can be seen in figure 4.3(b). The signed long value next to each heading refers to the hash value generated as part of the HDM. As the cycle of experimentation continues, an experimenter may generate many of these HDM text files and may wish to compare two datasets in order to detect changes. If a change has occurred, it can be investigated in the results report illustrated in figure 4.3.

HDM nodes that exhibit a change in their hash value also affect changes in the parent nodes, as their hash values are determined as a sum of themselves and their child nodes. The prototype also looks for other changes such as row and column size which can potentially disable the change detection functions. The HDM prototype provides the user with an adjustable level of tracking granularity for the column data. We split the data into data windows and apply a hash function to each portion of data. We can then determine the approximate size and location of the changes as well as their overall impact to the column data. The benefits of this approach are determined by the implementation of the hash and the size of the data windows. If the windows are too large, we find the hashed values to be as defunct as before, detecting a change but giving no information as to its size or location. If the data windows are too small, we may be storing as many hashes as there are values in the dataset.

Upon initialisation of the HDM, the user enters a tracking granularity value, hereafter referred to as the granularity factor. The HDM breaks each column of the dataset into groups of rows, the group size determined by the granularity factor. The algorithm generates an 8-bit alphanumeric character for each data group and appends it to the HDM, forming a representation of the column data. At the point of comparison, we can detect changes and infer details about the size and location of the change. Given that there are a finite number of alphanumeric values available and that some are used as special characters to mark the HDM text file, we conclude a collision probability of around 1%.

The hashing algorithm employed within the HDM is the simple RS Hash[8].

The RS Hashing algorithm:

```
begin proc RSHash($string$) $\equiv$
        $b = 378551;$
        $a = 63689;$
        $hash = 0;$
        for $i := 1$ to $string.length$ do
            $hash := hash * a + string.charAt(i);$
            $a := a * b;$
        od
  .
end
```

[8]A simple hash function from Robert Sedgwick's book, *Algorithms in C*.

So far, we have described how the HDM prototype manages changes that do not affect the semantics of the dataset. Where a dataset has either gained or removed a column, the HDM prototype will flag the change in the report but cannot detect changes in either the gained or removed column. The order of the columns in either dataset does not affect the operation of the prototype. Additions and removals of rows are harder to manage. If a row is appended to the end of the dataset, the prototype simply flags a change in the number of rows and ignores the additional row, as there is no original row against which to compare it. A removal of a row will affect the detection and tracking of all subsequent rows as they will become misaligned against the original dataset. The same occurs with a dataset that has reshuffled its rows.

We originally intended to represent the Hashed Data Model in an existing XML schema, either as a separate document or merged with the original data document, perhaps using a XML-generation tool such as **Bio2X**[98]. Many data representations in XML become unmanageably large due initially to the size of the raw data, but the problem is magnified given the overhead in representing the data in XML format. In such situations, it may be very useful to have HDM values of various significant portions of data in order to either promote investigation of a certain area or deem it unnecessary. This could significantly increase the speed of parsing and working with large sets of structured data.

The **Hashed Data Model** is not a complete solution for change management. It does, however, provide a reasonably elegant and lightweight solution for detecting data change from one version to the next. The **HDM** may be particularly appropriate for datasets that change very frequently as one only has to store a single, small text file as a representation of the version and from that, we can detect change and infer some level of difference. But the **HDM** does not go far enough to support a full, complex experimental environment[9]. In the following chapter, we present the **ExperimentBuilder**, which endeavours to provide a total change management framework within a complete experimental environment.

[9]Indeed, the **HDM** currently only handles datasets with no experimental component support planned at present.

Chapter 5

Experimental Environment

The purpose of the experimental environment is to provide an abstraction of the real-world environment in which a scientist may be conducting experiments. We begin by describing the motivation for our implementation of the environment based both on our previous work and an overview of some of the shortcomings in competing products. Section 5.2 describes the requirements that have been derived, in part by an analysis of the case study, but also with a view of the limitations found with the **Hashed Data Model**. We describe the experimental components that our system will employ and illustrate the requirement for multi-version components. Section 5.5.2 describes our methodology for versioning data[1]. The persistence of the experimental environment, containing multi-versioned components, was a significant challenge and our approach is described in 5.6. Finally, section 5.7 describes the **ExperimentBuilder**, our application for representing the experimental environment and the components therein. We provide a brief overview of the main design components of the **ExperimentBuilder** together with explanations of our versioning methodologies (those which have not already been discussed) and a description of some of the key elements of the user interface.

5.1 Motivation for the Experimental Environment

The motivation for our implementation of the experimental environment is derived from three distinct sources;

1. A comprehensive review and analysis of the case study.
2. Our previous work with the Hashed Data Model with an understanding of the limitations.

[1]In this context, we refer to data in tabular formats.

3. A review of the surrounding literature, in particular similar implementations with a discussion of their respective shortcomings.

The first two items have been covered in chapters 3 and 4. The following is a brief overview of some existing, competing strategies together with a description of their respective shortcomings. This should help us to identify how our product should seek to either further the functionality of such approaches or aim to provide a differing service.

We have already mentioned some of the problems associated with biological data in chapter 2 and we shall not repeat them here. Suffice to mention that, aside from the ubiquitous problems of the volume, heterogeneity and complexity of the data, we have concluded that it is the complexity of the processes that generate the data that cause significant problems when attempting to identify or estimate the impact of changes to the data. It is the complexity of the metadata that is important for us. But let us start nearer the beginning.

We are looking for changes in the data, so how do some existing data providers do this? Some data providers, such as the Protein Data bank (**PDB**), allow access to data in flat files[2]. They may also allow time-stamped snapshots of previous versions of the database to be downloaded from the archive, however, there is no information to suggest how the data has changed from one version to the next. Many data providers, although mainly the smaller secondary data providers, make their data available in this way.

Larger primary data providers, such as **Swiss-Prot** and **Ensembl**, go a little further and allow the user to inspect previous versions and view the differences between them. With the use of a *diff* generator, the changes between versions are highlighted in the browser. But this is only really useful for investigative purposes and, even then, on a very small scale. If we experience a problem within our experiment and we are lucky enough to know the data item that is causing the problem, then we can use this *diff* tool to investigate further. This is of only limited use, especially as our experimental environment grows and problems are dispersed over multiple nested layers of experimentation. The data version is still without any indication of change or dependency on other data. **Ensembl** complicates the issue somewhat further with the inclusion of stable identifiers for genes, proteins, exons and others. When the data undergoes any significant change, it may result in a dramatic change in the specific data model. In such circumstances, the old set of IDs are completely retired and a new set are created. Given that this may have potentially disastrous effects on any previous work, **Ensembl** have developed a tool called the **ID History converter** which maps changes in the ID from the earliest version to the latest.

[2]For the sake of simplicity, we will refer to data that is made available in a single batch as a *flat file*.

The **ID History converter** from **Ensembl**, although a small data-dependant tool, clearly highlights an identified requirement to interact between various versions of a datasource, not just the latest.

We have already discussed the merits of the **Taverna** workbench for ^{my}Grid in chapter 2. It is worth mentioning **Workflow Monitor**; a tool, developed for **Taverna** 1.x, designed to help prevent workflow decay[3]. Workflow decay describes the situation where a historically successful workflow will no longer execute. The decay may be caused for many reasons but workflows generally fail to execute due to the unavailability of one or more component processors (web services). As a standalone tool, the **Workflow Monitor** provides a useful explanation of the errors encountered by a workflow but, more importantly raises the issue that there may be multiple causes for the breakdown of a workflow, involving one or more of the incumbent processes.

The workflow engine from **Taverna** allows us to piece data and processes together to form a repeatable experimental workflow. In fact, the workflow is only repeatable if the data and processes do not change in a way that cause the workflow to break down. We intend our experimental environment to manage these problems, albeit outside of the Taverna framework. It is fair to conclude that much of the inspiration for our implementation of the experimental environment comes from workflow engines such as **Taverna** but, as you will see, we intend to do something quite different. We aim to establish workflow, not for the purposes of re-enactment but to detect dependencies and estimate the propagation of impact as the various workflow components evolve.

We will revisit some of these considerations in chapter 8 as we discuss the relative merits and demerits of our approach.

5.2 Requirements for the Experimental Environment

The purpose of this section is to document and communicate the requirements that have been elicited both from the system clients and extracted through the analysis of the current working example. It is intended to bridge the gap from an understanding of the case study to the presentation of the **ExperimentBuilder**, our solution for dealing with the elicited requirements. Through an extensive collaborative analysis of the changes and impacts within the working example, a set of functional and non-functional requirements were established and are presented here.

[3]More information can be found at http://www.omii.ac.uk/wiki/WorkflowMonitor.

5.2.1 Overview, Purpose and Scope

The overall purpose of the **ExperimentBuilder** is the creation and management of computational *in silico* experiments with all predictable aspects of the experimentation. The system runs as a Java application running in conjunction with a **MySQL** experimental database containing the environmental data. The database can either be held locally or on a public or private server. All aspects of the system have been designed to be platform independent.

Experimentation often occurs as the solution for problems arising either from theoretical hypotheses or hypotheses generated from previous bouts of experimentation. During this pre-experimental time, there may be various and often numerous scientists, technologists and other collaborators involved who should be accounted for during the requirements gathering stage.

The **ExperimentBuilder** should allow an experimenter to create heterogeneous experiments based on a variety of differing hypotheses and specify them in such a way as to accurately model the real-world experimental environment. The primary purpose of the **ExperimentBuilder** is to provide the tools necessary to construct the experimental environment complete with all required parameters. The secondary purpose of the **ExperimentBuilder** is provide tools to enable the management, tracking and impact estimation of changes that occur within the experimental environment both during and post-experimentation. When a change occurs within the experiment, or there is a change to some experimental condition, a propagation of this change occurs and manifests as impact throughout the dependant portions of the environment. The purpose of the **ExperimentBuilder** is to monitor the experimental changes, model and estimate the resulting impacts. It should then present the results in a way that allows experimenters to understand the impacts that exist within their environments along with their respective significants.

It is worth noting that the **ExperimentBuilder** does not embody the experimental processes directly. There are no technical or logical links between the **Experiment-Builder** and the real life experimental environment. The **ExperimentBuilder** should run along side and mimic the real life experimental environment. In this respect, the **ExperimentBuilder** should contain only experimental data and metadata, making the system a passive representation of the environment, incapable of re-enacting any stage of the experimentation process.

5.2.2 Scenarios

We will describe some *scenarios* below in order to elicit some requirements for the **ExperimentBuilder**. We aim to provide a considerable amount of functionality within the **ExperimentBuilder** and hope to cover most of these aspects in the three *scenarios* described below.

Adding an ExperimentBuilder Component The *scientist* identifies an experimental component, for example a dataset[4], that has yet to be added to the **ExperimentBuilder**. The *scientist* submits the component to the *populator*[5], providing some associated metadata and a list of specific, identified dependant components. The *populator* initiates the **ExperimentBuilder**, restoring the individual experimental environment of the *scientist* (loads it into the **ExperimentBuilder**). The *populator* then specifies the addition of a new component within the **ExperimentBuilder**, supplying the filename of the component along with the metadata provided by the *scientist.* After submitting the new component, the **ExperimentBuilder** provides the *populator* with the opportunity to define the dependents of the new components; any existing component within the environment that may have an impact relationship to the new component. After making these definitions, the component is formally added to the experimental environment.

Create an Experiment and Add a Component The *scientist*, conducting experiments, is either observed by the *populator* recording the experimental details, or conveys their experiments directly to the *populator.* In either case, the *populator* will create an experiment within the **ExperimentBuilder** based on metadata about the experiment that they have received from the *scientist*[6]. The experiment will likely contain some, if not many, experimental components. If the *populator* finds a component within the experiment that is not represented in the **ExperimentBuilder** environment, they must add it first, as per the scenario above. Components are logically added to the experiment within the **ExperimentBuilder** until that experiment fully represents the *real-world* experiment conducted by the *scientist.*

Update Components and Investigate Impact At some point in time, components will become out-of-date and new components will be available. Component updates are identified by two main sources; the *collaborator* who will build a new private dataset

[4]In the case of datasources, a *scientist* may work with a *collaborator* who's purpose is to generate and manage the experimental data. It may be the *collaborator* who is called upon to reveal the environmental relationships of a particular data-related component.

[5]For the purposes of these scenarios, we regard the *scientist* and the *populator* as two different individuals though, in many cases, the two roles may be performed by the same person. We are, however, specifying the *role*, rather than the individual *actor.*

[6]Note that the scientist has no direct involvement in the construction of the experimental environment. This is to ensure that the environment is constructed correctly and consistently.

and flag the changes to either the *scientist* or directly to the *populator*; or the *scientist* will identify the changes directly through ongoing work within their experimental environment, conveying these changes to the *populator*. Once the *populator* has been made aware of the requirement to update, and has a copy of the updated component, they can replace the old component with the new within the **ExperimentBuilder**. The **ExperimentBuilder** replaces the *current version* of the component with the new version so that any subsequent employment of the component will use the updated version. The **ExperimentBuilder** also analyses the experimental environment searching for uses of the component and flags these potential impacts to the *populator*. The **ExperimentBuilder** presents an estimation of the cost of updating the component for each experiment within the environment. The *populator*, in collaboration with the *scientist* and perhaps the *collaborator* decide whether to update the experiment or not, based on the estimated cost of doing so, provided by the **ExperimentBuilder**. If they decide to update, it will likely yield different experimental results[7] and these differences are submitted to the system by the *populator* and the scenario begins again.

5.2.3 Use Cases

From the *scenarios* described in section 5.2.2, we extract several important *use cases*, described here with the aid of diagrams. We present two diagrams; the first illustrates the construction of the **ExperimentBuilder** environment; the second describes the detection of change and the measurement of the resulting impact.

There is a general prerequisite for all the following use cases in that we rely on the *scientist* having already established the real-world experimental environment. We understand that this may change over time and there are use cases to describe this evolution but we do not explicitly cover the *construction* of the environment as it strictly falls outside of our system boundary.

Add Component

Prerequisites The *real world* equivalent of the experimental component has been added to the *scientist's in silico* environment. Components are added to the **ExperimentBuilder** only once, so we assume that the component has not been added already.

User Input The *scientist* identifies a component that has yet to be added to the **ExperimentBuilder**. This may be required through the addition of a new component or during the construction of the **ExperimentBuilder** for an existing

[7]Experimental results are also encapsulated as components within the **ExperimentBuilder**.

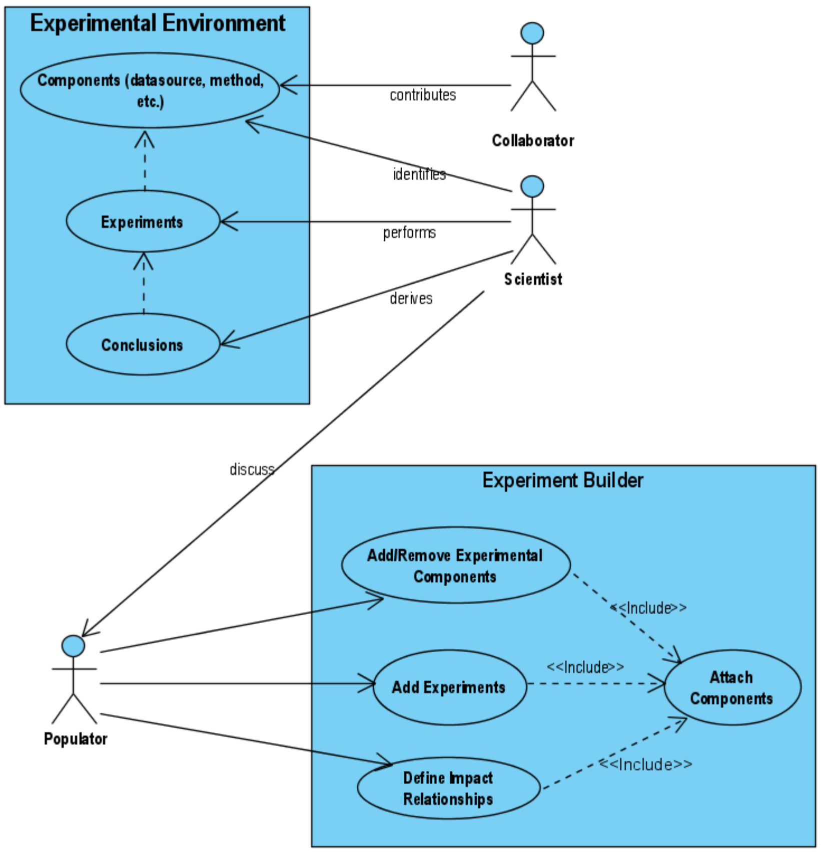


FIGURE 5.1: A use case detailing the relationship of the real-world environment and the *in silico* experimental environment.

experimental environment. The *scientist* passes the details, including metadata and dependant component information, to the *populator*. The *populator* restores the experimental environment and then defines the component within the **ExperimentBuilder**, adding it to the environment. At the point of addition, the *populator* defines the relationships between the new component and any existing components within the environment.

Response There is no explicit response from the **ExperimentBuilder** other than a confirmation of the addition and an update to the existing *in silico* environment.

Add Experiment

Prerequisites The *scientist* has created the experiment within their experimental environment and wishes to add it to the **ExperimentBuilder**[8].

User Input The *scientist* passes the experimental details to the *populator* who initiates and restores the appropriate **ExperimentBuilder** environment. The *populator*

[8]A *scientist* may conduct many experiments that are not included in the **ExperimentBuilder** as the additions occur at the request of the *scientist*. It is, after all, their change management tool.

defines a new experiment, entering details such as *hypothesis*, *experimenter*, *date*, and other any other requested descriptors.

Response The **ExperimentBuilder** creates a shell of the experiment within the *in silico* environment, ready to be populated with components from the experimental environment.

Attach Component to Experiment

Prerequisites The experiment and the component to be added exist within the *in silico* environment.

User Input Using experimental details supplied by the *scientist*, the *populator* selects the appropriate experiment, highlighting it to be modified. The *populator* then selects the component and attaches it to the experiment.

Response The **ExperimentBuilder** requests from the user, a value for each registered parameter of the component being added.

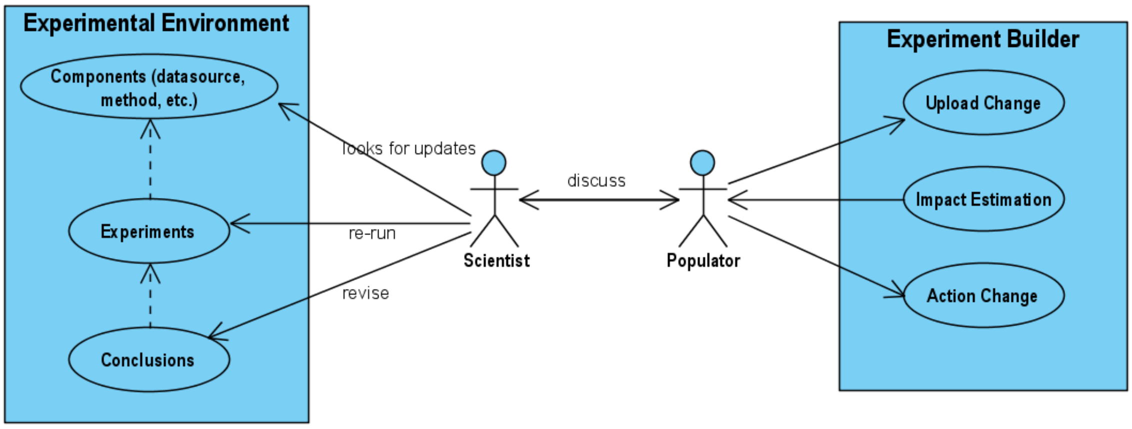


Figure 5.2: A use case diagram describing the updating of an experimental component and the resulting impact analysis.

Upload Change

Prerequisites The *scientist*, working within their own experimental environment, continually looks for change within their workspace. The prerequisite of this use case is the identification of a component that either has changed or is *suspected* of change and has been passed to the *populator*.

User Input The *populator* uploads the new version of the component to the **ExperimentBuilder**.

Response The **ExperimentBuilder** provides the user with an *Impact Estimation* report. This report contains three main items of information. It shows how the change in the new component directly affects each other component that carries a dependant impact relationship. It also shows components that are indirectly affected by the change; i.e. components impacted by the directly affected components and the components that are impacted further. Most importantly, it shows the experiments that contain affected components and an estimation of how much the results of those experiments are affected by the changed components. The **ExperimentBuilder** displays this change as a real world cost to the *scientist*[9].

[9]We aim to present a *real world* impact; i.e., one that contains an estimation of the actual effort involved in recomputation, whether that be time, money, or machine cycles.

Action Change

Prerequisites The *scientist* is presented with the *Impact Estimation* report.

User Input The *scientist* must decide, possibly considering other sources of contribution such as *collaborators*, other *scientists* or *managers*, based on the *Impact Estimation* report whether it is necessary or worth the effort required to re-run the specified experiment. When a decision has been made, the experiment may be re-run using the updated component.

Response The *scientist* indicates to the *populator* whether the experiment has been updated or not, presumably passing to the *populator* an updated set of results. At this point, the **Experiment Builder** will reflect the updated results and then trigger a possible update for any dependant components. This generates a new *Impact Estimation* report based on the update. This report is then passed to the *scientist* and the update cycle continues. There may be several iterations of this updating process and successful implementation relies, in particular, on a streamlined relationships between *scientist* and *populator*. In many cases, these roles would be taken by the same person, which would obviously aid this difficulty.

5.2.4 System Characteristics

The primary actor for the consideration of the **ExperimentBuilder**, at least chronologically, is the *scientist*. The *scientist* is responsible for the construction of the real-world computational experiments and it is the work of the *scientist* that must be replicated in the **ExperimentBuilder**. It may also be the responsibility of the *scientist* to populate the **ExperimentBuilder** with experimental details or this task may be delegated to someone else, identified as the *populator*. In the use cases above, we describe the *scientist* and *populator* as individual roles though they may often be performed by the same person.

The *scientist* may execute their experimentation with several other contributors and, in this context, are referred to as *collaborators*. The **ExperimentBuilder** has the capability of managing multiple datasources so we must also consider the outside influences from such datasources and the people who maintain them, referred to as *administrators*.

The **ExperimentBuilder** has been conceived as a result of an in-depth analysis of an existing biological problem, involving primarily demographic and protein quantification data and due to this, the **ExperimentBuilder** will be designed, at first, to accept data of these types. It is imperative, however, that the **ExperimentBuilder** accepts other types of data, in particular genomic and microarray data and these forms of data will also be accommodated within our system.

We aim to provide data support for;

- Demographic/Tabular data
- Genomic Sequence Data (multiple formats)
- Microarray Data

The main purpose of the **ExperimentBuilder** is to encapsulate the experimental environment in a way that allows and facilitates the tracking and investigation of change. To this end, we must concentrate on some specific aspects of the graphical user interface that allows investigation of the required level. A significant part of the **ExperimentBuilder** is the accommodation of varying types and sources of data and should provide an interface for visualising multiple versions of a variety of datasets. The user can group, sort and search datasets according to preference and move forward and backwards between versions of a specific dataset. And the same should be true for methods and tools within the **ExperimentBuilder**.

As well as static investigation of the experimental components, the paramount reason for the **ExperimentBuilder** is the investigation of change within the experimental environment. We also, therefore, allow investigation of the impact relationships between experimental components. In other words, by selecting a component, the user will be able to see all related components[10]. We provide a view of related components together with indications of the strength of the relationship.

We also describe experiments within the **ExperimentBuilder**, which can be created and grouped according to preference. As experimental components are added to experiments, the *change profile* of the experiment evolves, taking into account the impact characteristics of the components that are added. Over the life of an experiment, we expect it to change to reflect the changing requirements of the experimenter and we also expect the **ExperimentBuilder** to accommodate for this dynamic. We manage the change with a selection of tools provided by the **ExperimentBuilder**, which we have identified by the extensive analysis of an existing example environment, as described in chapter 3. The tools and main features of the **ExperimentBuilder** are described in the next section.

[10]A related component is one that either affects the component or is affected by it.

5.3 Main Product Features

We describe below the main features of the **Experiment Builder** as they would be presented to a potential customer or user of the system.

Bespoke Component Construction Model existing components within the **Experiment Builder** or design your own bespoke components. Any dataset or tool, whether *off-the-shelf* or bespoke, can be added to the **Experiment Builder**. Define component details, including parameters and even component code, where applicable. Components are added once to the environment and can be grouped or sorted, ready to be used within a single, or possibly many, experiment(s).

Multi-experiment Environments Scientists can define their environments within a single **Experiment Builder** session using multiple, differing experiments. Components can be added to experiments from repositories within the **Experiment Builder**, specifying independent parameters.

Full Impact Profile for Entire Experimental Environment Every dataset, tool and experiment is connected in the **Experiment Builder**, explicitly defining how each impacts the other. Select any component and see how it relates to all other components in the environment.

Change Tracking Make a change in one component and watch that change ripple through the environment. Depending on the nature and strength of the change, it will propagate through the system following a specific path. The **Experiment Builder** identifies this path and estimates the impact.

Impact Estimation/Real Cost Advice The **Experiment Builder** provides the user with an estimation of the impact, not only to components directly impacted by a change, but also to components that are indirectly affected[11]. By measuring all the possible impacts within a system, direct or indirect, we can provide a *total impact estimate* for any given change. But the **Experiment Builder** goes further than this. Rather than providing an arbitrary value associated with the impact, the **Experiment Builder** provides a real-world cost of the impact as a *total update estimate*[12][13].

[11]A component can be thought of as being indirectly affected if the impact reaches it via more than one level of component relationship; a component that carries impact from a component that has been impacted can be thought of as *indirectly impacted*.

[12]This is the real cost of updating all the required components as a result of the initial change.

[13]Please note that the *total update estimate* requires user input to estimate the real effort involved in updating.

Persistent Environment The **Experiment Builder** provides an entirely persistent environment allowing multi-versioned datasets and tools as well as the relationships between them to be persisted and restored at will. Scientists can share environments, for example, choosing to restore only the components without the experiments. In this way, labs with multiple scientists can work from a shared component repository, eliminating the need for repeating identical shared components, whilst retaining one unique experimental environment per scientist.

5.4 Modelling Experimental Components

One of the main limitations of the **Hashed Data Model** and its implementation was the lack of integration with the real experimental environment. During the description of biological data and the accompanying experimentation, several key characteristics that pose problems for its tracking and management were highlighted. The **HDM** provides a solution for data-change tracking as a light-weight cross-platform application but fails to take into account the complexities and challenges posed by a complete and fluid experimental environment.

To provide an accurate representation of the environment, the individual experimental components must be modelled in such a way as to model the real environment. This is important for two reasons. Firstly, if change and impact are to be tracked and estimated accurately and correctly, there needs to be persistent model of the environment within which to model and present the changes. Secondly, as the system relies on the user entering metadata about experimental components, it is clearly preferable they they do so within a natural and realistic environment.

Perhaps we should start by defining an experimental component. There are many aspects of a single experiment from off-the-shelf components to bespoke scripting applications to singular parameter changes. It is not possible to foretell the exact nature of the experiments, rather we must allow experimenters to populate the environment as per their own experimental design. We must therefore design the environment to handle differing experimental components, affording experimenters the variation demanded from their experimentation. We have identified a set of the most commonly occurring experimental components with which experimenters can reflect their designs. These include items such as datasources, datasource wrappers, methods and tools.

A **Datasource** in the **ExperimentBuilder** is defined as a body of text-based data representing information of a biological, annotational or demographic nature. In fact, the

ExperimentBuilder will accept any data regardless of the information it represents, as long as it is in the correct data format.

A **Datasource Wrapper** is any kind of tool that operates on a datasource before the data is used in the method. Some datasources will contain data that can not, in the present state, be used with the chosen method. In such cases, the data must be altered in some way or another in order for it to be integrated into the experiment correctly although it is generally accepted that the content or the *meaning* of the data is unchanged.

A **Method**, as defined in the context of the **ExperimentBuilder**, is a data transformation tool that takes, as input, some data and generates, as output, further data referred to as **Results**. The differences between methods are vast and numerous but each broadly conforms to the same input/output profile. The **ExperimentBuilder** allows the user to specify the methods themselves, define the inputs and outputs and tweak the parameters necessary for the specification of the method.

A **Tool** is defined in the **ExperimentBuilder** as any application or process within the experiment that is used during any stage of the workflow either on its own or in conjunction with any other component. A tool that is used in conjunction with a datasource may be more aptly described as datasource wrapper rather than a tool and the distinction in this case is left to the individual user.

Experimenters specify and add components in order to build their experiments. These components, however, must be added in a defined manner so that experimental integrity can be maintained. For example, we have defined an experiment so that it can contain exactly one method. Thus the definition of an experiment[14] is, principally, that it contains one method. There are other constrictions that the experimenter must adhere to. A tool that operates on a datasource should, in most cases, be entered as a datasource wrapper although in some cases, it may make more sense to encapsulate the process as a tool.

The experimental components above were conceived after careful analysis of the case study as described in chapter 3. The case study uncovers many real avenues of possible experimentation and these, as well as many others, are fully accommodated by the available components provided in the **ExperimentBuilder**. As well as revealing the candidates for components, we have also revealed information about the inter-component relationships.

[14]This definition is confined within the boundaries of our experimental environment. We do not intend this definition to represent all experiments.

5.5 Multi-version Components

In the previous section, we specified several aspects of the experimental environment and introduced the idea of modelling experimental components within that environment. The modelling becomes considerably more complex as more versions of the components become available. The management and application of datasets with multiple versions is arguably the biggest consideration when designing such an experimental environment. Indeed, one of the principal problems identified in the current experimental approach is the proliferation, variety and frequency of unpredictable and unannounced changes to datasets. These changes can and regularly do cause significant impact to other experimental components as well as the experiment results. The experimental environment must be able to reflect these impacts as well as provide ample support to reflect the dynamic and fluid characteristics of the datasets.

5.5.1 Component Behaviour

Having modelled experimental components, we must prepare for the possibly many subsequent versions that may follow. Subsequent versions of components appear for many reasons and must be handled by our framework. Experimenters must estimate the impact of these subsequent versions to their existing experimental setup. Should a new version generate a significant impact, the experimenter may be faced with the prospect of updating the experimental environment either in part or, as a worst-case scenario, rebuild entirely from scratch. To avoid costly rebuilds, the impact must be accurately estimated. In order to do this, the modelling of components must be accurate including the modelling of multi-version components.

Experimental components behave differently and these differences are even more pronounced when considering multi-version components. Databases, both public and private, can change in a variety ways and some with more considerable effects than others. The changes propagate to the experimental datasets based primarily on their individual generating queries. Changes occur in databases for many reasons. Some important changes are highlighted in our case study described in section 3.4, but we will describe the main sources of change here.

Data Change Changes to the data itself is the most obvious way of affecting the resulting experimental datasets. These changes occur for many reasons and some of these are described in section 3.4. It is important to note that it is not sufficient to measure changes in the source database. Changes to the source database may occur on an almost continual basis, particularly if the database is very large and/or accessed or contributed to by a large number of people. That does not necessarily mean that these changes will follow on to the experimental datasets. To analyse change in the experimental datasets, we must investigate both the source data and the generating experimental queries. Semantic changes to the dataset, such as changes to the dataset schema, can be more difficult to assess and manage as there can be less associations with previous versions. For such changes, it may be necessary to look at or employ schema evolution techniques.

Data Policies Policy change can appear anywhere at any time and usually occurs in order to fix an existing problem. In such cases, the policy change may be welcome. Policy changes are most disruptive when they appear unannounced, for example within the source database or a third-party tool, causing the system to break. A change of data format or additional authentication can easily result in system breakdown and there is very little that can be done in order to prevent or prepare for this. Although prevention is difficult, policy change highlights a requirement for good debugging and error explanation so that when the system does break, the task of bringing it back online remains as painless as possible. When policy changes are announced in a timely fashion, warning potential users, they have far less impact on the system integrity.

Software Updates Updates to software, both bespoke and third-party off-the-shelf software, can cause significant disruption to the experimental environment. We must first establish whether the update is actually worth the resulting impact. A bespoke software update has an implied acceptance of the impact, mainly because the update would not be taking place unless it had already been considered worthwhile. We must establish the impact of both updating the software process and not updating the software as both scenarios can generate impact. An update to a commercial third-party piece of software carries two considerable challenges. The first is to discern whether the update is both critical, or at least important, for the successful continuation of experimentation and applicable to the experimental environment. Software updates may include many minor updates. If these updates are not applicable to the experimentation, there is no point updating. The second challenge is to identify the effects and impacts of not updating the software. Some software suppliers enforce the upgrading of software with penalties for the refusal of upgrades such as discontinuation of support or, in some cases, complete withdrawal of the product. Software providers may attempt to complete the upgrade process transparently and this poses a significant problem. Whenever a process changes, the system must be rebuilt and retested to ensure correct functionality.

Parameterisation Changes to experimental parameters can causes significant impact to an environment, not least due to the ambiguity of effects of parameter change. An experiment can contain any number of parameters reflecting the precise nature and hypothesis of the experiment. If any of these parameters are changed, it is unclear whether this represents a completely new experiment or simply a tweaking of an existing experiment. It seems wise to leave this distinction to the experimenter with an inherent knowledge of the experiments and how they should be maintained. It does, however, present a lack of clarity when managing the experimental environment.

Advent of New Technology Similar to the case of a software update, the advent of new technology creates a case for updating or enabling the system to accommodate the new advances. Whether this update is worth the resulting impact should be evaluated by the experimenter and system administration and depends largely on the potential benefits to the experimental processes. New technologies will appear from time the time and it is up to the controllers of the experimental system to determine a sensible approach to managing this advancement. Clearly it is not efficient to rebuild and retest the system in order to incorporate every technological advancement as soon as it becomes available. It seems prudent to develop a long term plan of experimental environment version release, with obvious room for manageable variation, so experimenters can retain some degree of control over their system.

So, how do the different types of change affect the dependant components? There are multiple observed changes and effects and different changes are likely to produce different effects. It is, however, very likely that there are other non-observed changes and non-observed effects present within the environment. These cannot be explicitly managed because we have not, as yet, explicitly identified them. We must, therefore, build our system to accommodate effects based on, but not limited to, our defined and observed effects from the case study. Consider, also, the size of some experimental environments. We should not expect users to document change and effect in the terms of the specific types described above as it would be very inefficient. We require a system that manages all types of change with the ability to measure, without prejudice to type, the estimated impact within the environment.

How do the changes affect the experimental outcome? This is a difficult question to answer as it usually requires specific knowledge of the individual environment. A scientist may have some anecdotal evidence on how changes have particular effects but this is largely inadequate. Effect and impact are hard concepts to estimate and the complexity of the environment complicates the issue further. A small change may have a small

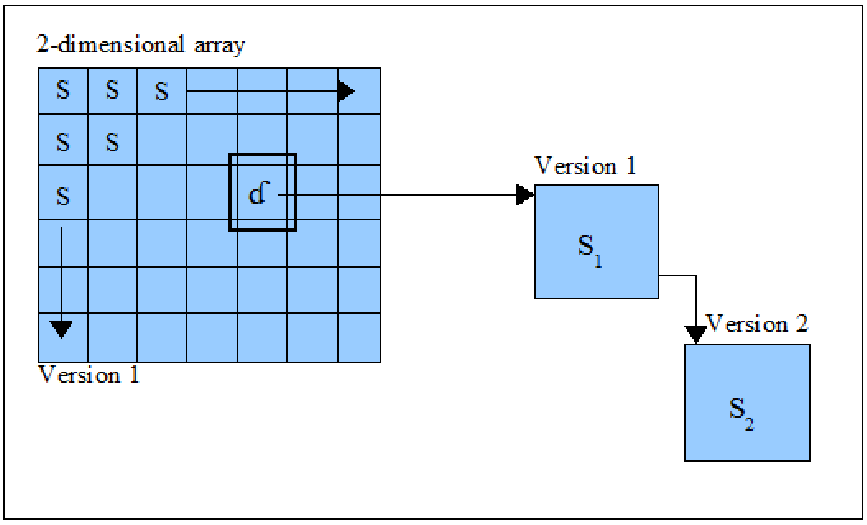


FIGURE 5.3: Versioning methodology using DBVS.

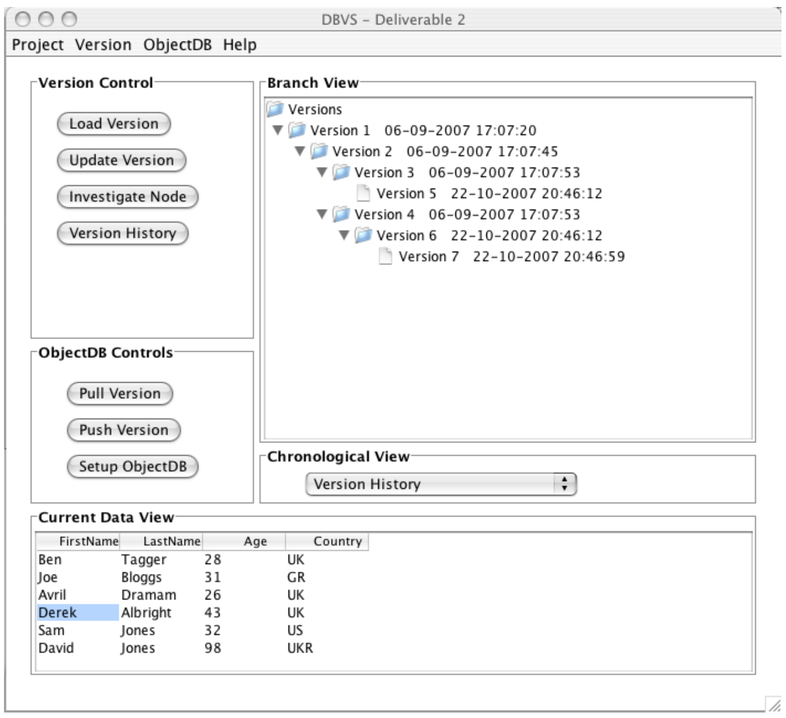
impact but when combined with another small change, the impact may be much greater. In addition to this, the change may not come from the directly affected component. Change may arrive at a dependant from a component that is itself a dependant of a changed component.

5.5.2 Modelling the data change

The **HDM** Prototype provides an interesting and novel way to model scientific data, with some sample metadata, and a first step towards monitoring changes among multiple datasets that can be generated during biological experimentation. The **HDM** has several shortcomings, most notably the restriction of granularity specification and the inability to model the complete experimental environment.

The **DataBase Versioning System** (**DBVS**) is primarily concerned with the versioning of data. The **DBVS** provides users with change-tracking for individual data items. We have already identified the need to support multiple versions of a single dataset and we also need the ability to investigate the changes on a fine-grained level. There are two principal requirements for the **DBVS**; individual change tracking; support for multiple versions. These two requirements contradict one another with respect to storage limitations. We require the retainment of every single change in the most economical way.

The **DBVS** stores the initial dataset as a 2-dimensional array of string variables. When subsequent datasets are loaded, the **DBVS** looks for changes between the two versions. For each data point that has changed, the string representing the value is discarded and replaced by a linked list of objects, each representing a single version. Using this



```
Versions
▼ Version 1   06-09-2007 17:07:20
    ▼ Version 2   06-09-2007 17:07:45
        ▼ Version 3   06-09-2007 17:07:53
            Version 5   22-10-2007 20:46:12
        ▼ Version 4   06-09-2007 17:07:53
            ▼ Version 6   22-10-2007 20:46:12
                Version 7   22-10-2007 20:46:59
```

FirstName	LastName	Age	Country
Ben	Tagger	28	UK
Joe	Bloggs	31	GR
Avril	Dramam	26	UK
Derek	Albright	43	UK
Sam	Jones	32	US
David	Jones	98	UKR

FIGURE 5.4: An extract from the DBVS prototype

methodology, the retention of multiple copies of the same data are avoided, as opposed to many traditional versioning systems, but each version can be retrieved from the same object, encapsulating the **DBVS** data structure. The user can append a new version to any previous version, taking advantage of the application's branching capabilities. Complete versions can be retrieved and the user can also inspect the entire history of any particular node. Due to the nature of the methodology, the storage size of the versioned dataset is also relatively minimal. A dataset with n datapoints at version 1 will contain n objects (at version 1, the object will be of type String). With each subsequent version (v) that generates c changes, the model is only incremented with c objects. Therefore, for any dataset containing n datapoints, a set of v versions with c changes with result in $n + vc$ objects[15]. It is important to note that datapoints that are not subsequently changed, retain only one version and are not affected by the versioning of other datapoints.

DBVS interfaces with **ObjectDB**, allowing the user to push and pull versions to and from the database application.

[15]This example implies that c is constant between versions whereas, there is likely to be differing numbers of changes for different versions.

5.6 Persisting the Experimental Environment

The **Java Persistence API** (**JPA**) is a framework that allows developers to manage a relational database from within their application code. Often considered to be the most significant advance of the Java Enterprise Edition (Java EE 5) so far, the **JPA** offers a simple, yet powerful standard for object-relational mapping (**ORM**). For a number of years, enterprise Java has been using a **POJO**[16]-based programming model and the **JPA** continues this trend, offering the ability to persist **POJO**s with little need to significantly alter the existing object.

There are broadly three processes to consider for the persistence of the **Experiment-Builder**; environmental setup, experimental design and impact estimation. The data contained within the **ExperimentBuilder** falls neatly into these three categories and, as such, are handled in three distinct ways. The user creates the experimental environment whereupon each component is defined within the **ExperimentBuilder** and added to the environment. At the point of component creation, the **Experiment-Builder** extracts keywords from the component generating a *keyword profile*, estimates inter-component relationships based on this profile, and then prompts the user to create a *cost profile.* The final stage, although these stages do not always occur in order, is the definition of experiments using the defined components. In the following section, we describe these processes, in terms of the effect on persistence.

A *Line of Enquiry* (**LOE**) contains between one and many experiments but a **LOE** must contain at least one single *Experiment* in order to qualify as a line of investigation. The *Experiment* contains the various experimental attributes along with the numbers of datasets, wrappers and tools. The user can choose whether to enter attributes for *hypothesis, description* and *notes* but is forced to provide data for at least the *title* and the *date* if it is not to be auto-generated.

In order to specify which components are used within an *Experiment*, we define a *Resource*, which refers to a particular, experimental component and is held elsewhere in the database using;

resType Specifies whether the resource refers to a dataset, wrapper, method or tool.

resID The *ID* of the component in the corresponding component pool, depending on *resType.*

resVersion Once we have identified the component, we must specify the version that has been used. We cannot assume that an experiment contains the latest version of a component as, for many reasons, it may not.

[16]Plain Old Java Object

Two or more experiments may use the same component but they will likely have differing parameters, although this is also not guaranteed. Each *Resource*, therefore, will hold the properties of the parameters specific the use of the component within that *Experiment*[17]. There are two further attributes of *Resource*, *attachedDatasourceNum* and *attachedDatasourceVersion*, that are used only when the resource represents a *wrapper*. As we have already mentioned, a *wrapper* is a functional component that is always applied to a *datasource* and, for this reason, we must specify the datasource to which it is attached.

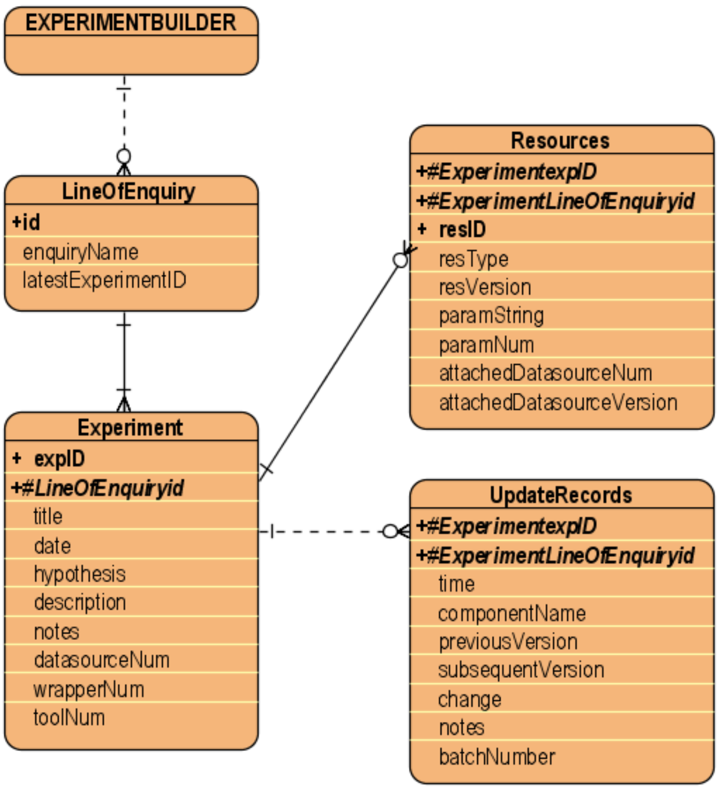


FIGURE 5.5: Storing experiment details and associated components.

When a component is updated, the **ExperimentBuilder** flags the change as a possible requirement to update any experiments that contain the component. The user may decide, upon consideration of the impact to their experimentation, to update the component. We keep an *UpdateRecord* to record each of these updates, tagging them to the *Experiment*. This provides the user with an informal audit trail for the *Experiment*, showing the changes over time. Within an *UpdateRecord*, we describe the component that changed, the time and degree of the change and the previous and subsequent versions of the component. We also supply a *batchNumber*. The *batchNumber* allows us to monitor how the changes were applied to the *Experiment*, namely whether the updates

[17]The parameters are retained in a tokenised string.

occurred sequentially over time, or were conducted as part of the same batch. This may prove important in determining further effects as a result of the updates.

We have implemented two different methods for managing the persistence of experimental components within the **ExperimentBuilder**, one for datasources and datasets and another to deal with the other components such as wrappers, methods and tools[18]. Figure 5.6 illustrates the persistent structure of the experimental datasets.

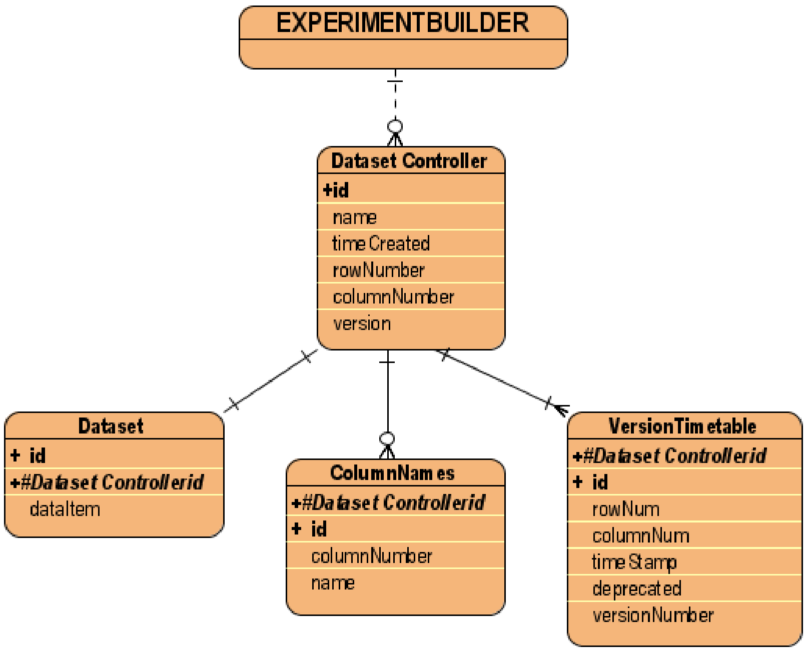


FIGURE 5.6: Persistent representation of the experimental datasets.

A significant function of the **ExperimentBuilder** is the version management of heterogeneous components. The persistence of such components, therefore, is far from trivial. We are implementing a version-enabled persistence framework for the experimental environment and we illustrate the structure of the persistence of version-enabled datasources in figure 5.6. The *DatasetController* describes a single datasource, irrespective of any specific version. We describe the generic metadata for a dataset, including row and column numbers, name, time stamp and the latest version number. Note that there is no specific content data within the *DatasetController*. The content for the datasource is held within *Dataset* and *DataItem*, where each version of each data node is held chronologically, referenced by version. For example, consider the following *DataItem*, represented by the following string.

```
1:56, 2:65, 5:22
```

[18]This is due largely to similarities in structure, although there are significant differences in the way the components interact within the experiment structure.

The premise is very simple. Each data point refers to a unique row and column value. The first integer refers to the data point version, the second refers to the value of the data point at the given version. The example above shows the values at versions 1, 2 and 5. Note that we do not specify values for versions 3 and 4, indicating that the data item in question does not change from versions 2 until version 5. Using this method, the data is represented at every version but there is no data replication, irrespective of the versioning history of the dataset. The information in *Dataset* tells us which data is related to which version but gives no information about the version itself. We use *Version* to describe the versions that exist for the dataset, with *versionNumber* relating to the version specified in *DataItem*.

Components are separated according to type and stored in a component *pool*. The *pools* contain each of the components available within the environment and each of the components are specified further by the *version*. The **ExperimentBuilder** provides the user with the ability to retain the source code for each component[19]. The latest version of this source representation is held within the component table (i.e., *Wrapper*). Each version of the component (i.e., *WrapperVersion*) contains a *codePatchString*, which is a *delta* comparing itself to the immediate subsequent neighbour, referred to as a *backwards delta*.

Each component version is linked to the **Component Impact Manager**, which manages all aspects of cost and impact for the component. Some aspects are version-specific such as the *keyword profiles*, whereas some remain constant regardless of the version, such as the *cost profile*. There are three main sections to the **Component Impact Manager**; the *KeywordProfile* including the *keyword exclusion list*, the *cost profile* and the *impact descriptors*. All *Keywords* are retained in a single table and belong either to the *KeywordProfile* or the *KeywordExclusionProfile*, in which case, the *weight* and *occurrences* attributes are ignored. The *CostProfile* contains the attributes required to generate an accurate estimate of the costs and efforts involved in the update of the component. An *ImpactDescriptor* is not attached to any single component. It specifies an impact relationship between components by identifying the subject and object component versions as well as the *impactWeight* of the relationship.

The **ExperimentBuilder** uses a single persist/restore strategy for managing the persistence of the experimental environment. The default operation of both the persistence and retrieval is to restore or persist the entire experimental environment. In addition to this users can, should they wish, specify the parts of the database they wish to act on by selecting the *Fine Control* option from the menu.

[19]Where applicable. Where the source code is not available, the user can use either a structured text document to represent the component version or alternatively, use any other method for representation.

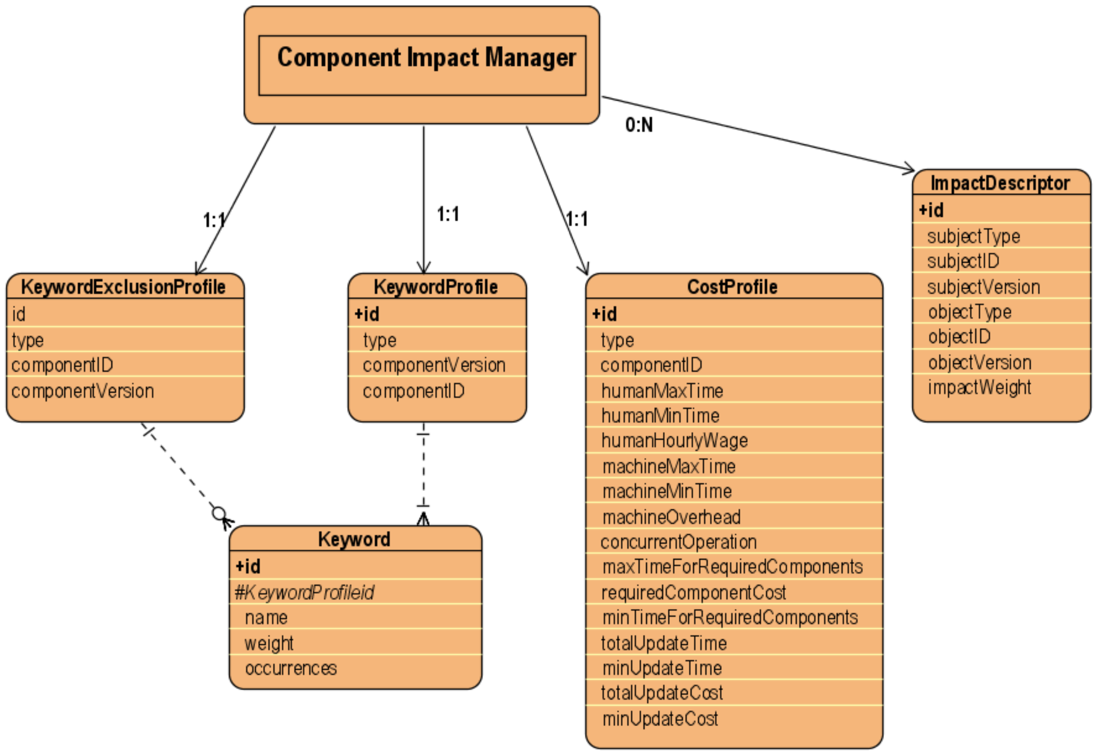
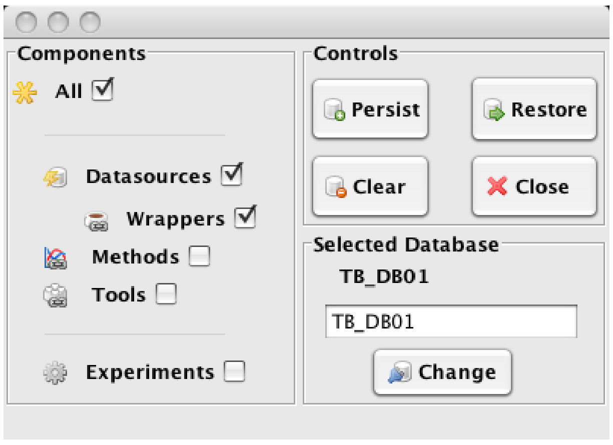


FIGURE 5.7: Defining keyword profiles, cost profiles and impacts.



All [✓]
Datasources [✓]
Wrappers [✓]
Methods [ ]
Tools [ ]
Experiments [ ]

Selected Database
TB_DB01
TB_DB01

FIGURE 5.8: Fine user control of persistence in the **ExperimentBuilder**.

As well as choosing which parts of the database to persist or restore, users can also switch operating databases or clear the current working database. In this way, scientists within the same laboratory can, for example, share the same experimental environment without access to each other's experiments.

5.7 The Experiment Builder

The **ExperimentBuilder** provides the housing for our framework for managing experimental change and understanding the impacts that result from this constant state of flux. We begin by providing an overview of the main system components and we describe some of the more important sub-systems. By the conclusion of this chapter, the reader should have an understanding of the main components and the sub-components of the **ExperimentBuilder** in order to appreciate how the tracking components integrate with the modelling framework.

5.7.1 Design Overview

Figure 5.9 illustrates the main components of the **ExperimentBuilder**, without the complexities of the full application. We describe some of the more important sub-systems in more detail later in this chapter.

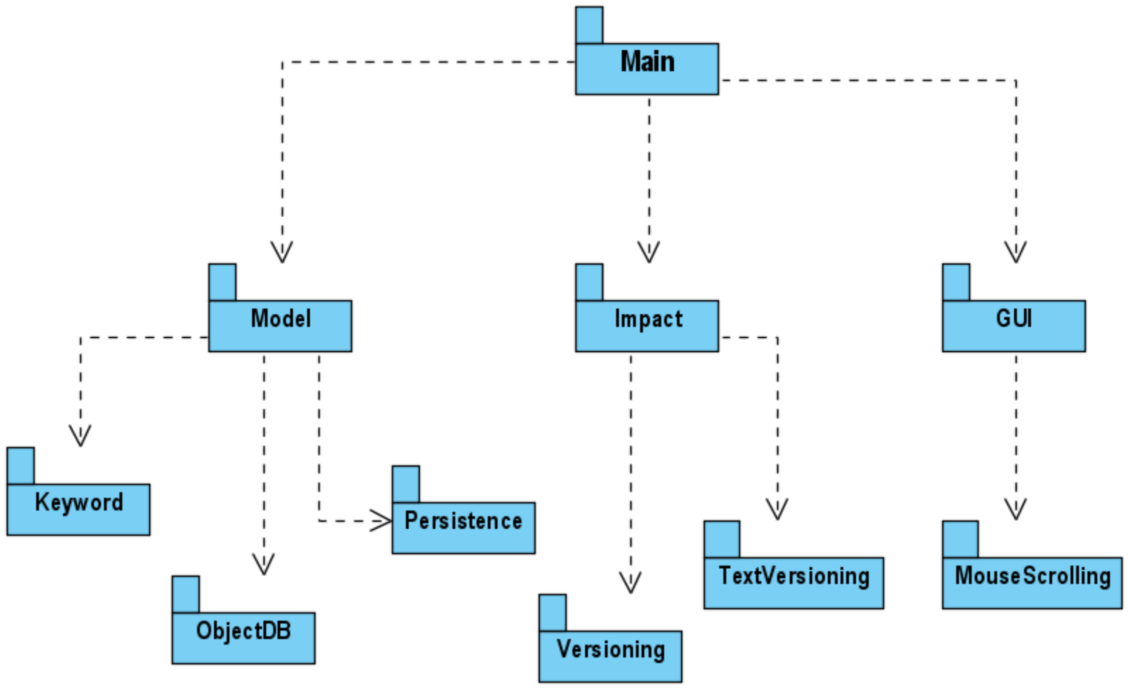


FIGURE 5.9: An overview of the main packages within the **ExperimentBuilder**.

There are broadly three subsystems within the **ExperimentBuilder**; the *Model*, incorporating the persistence aspect and the keyword profiling; the *View* which includes the user interface and the model interaction; and the *Impact* which handles the versioning and the impact estimation for the experimental environment. These sub-systems interact in various ways, too numerous to describe completely.

5.7.2 The Model Package

The **Model** package contains the classes that describe the structure of the experimental environment and the components therein. Within the **Model** package, illustrated in figure 5.10, we can identify two related but distinct areas; the constructing experimental components, such as the datasources and experimental tools; the experimental details such as the experiments themselves, hypothesis and experimental resources.

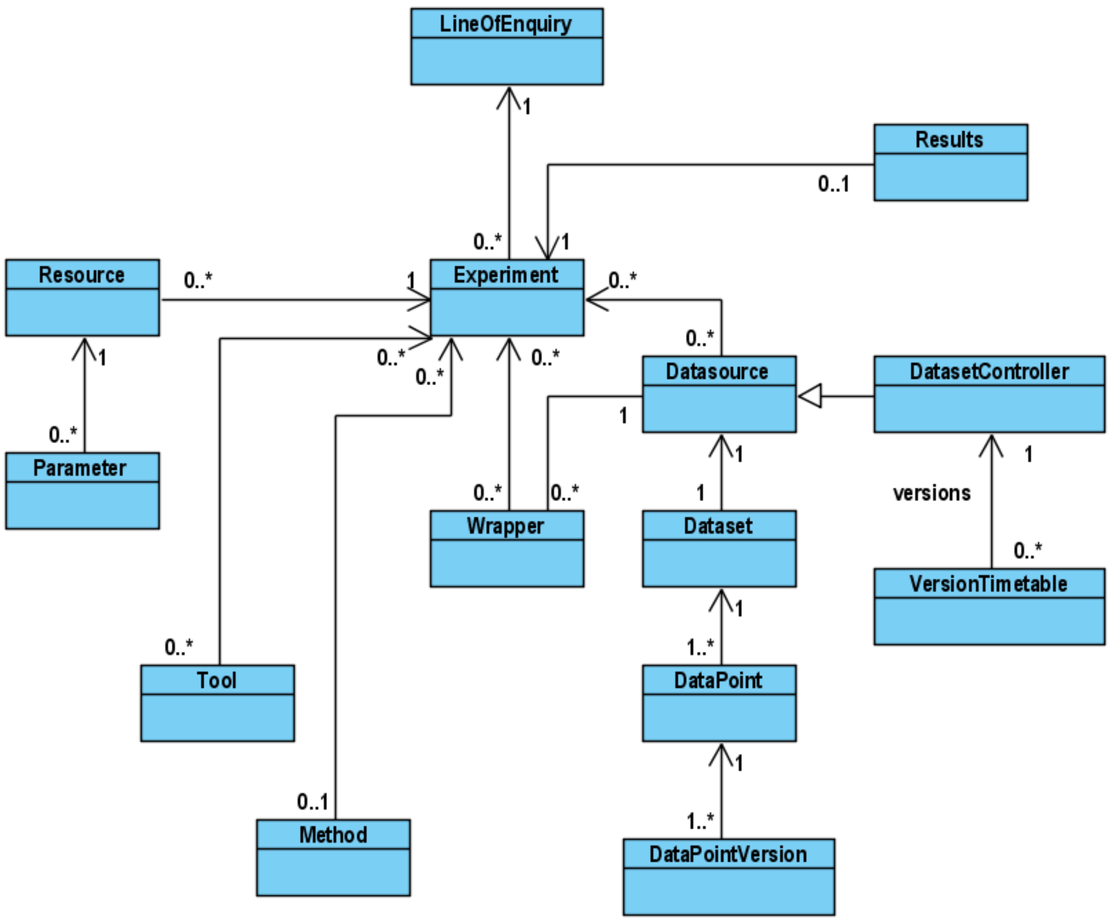


FIGURE 5.10: A class diagram of the **Model** package.

The focal point for the **Model** package is the **Experiment** class. Each **Experiment** can contain any number of experimental components and, conversely, the same component can belong to any number of **Experiments**. Components exist within the environment independently of any specific arm of experimentation but when a component becomes part of an **Experiment**, it is customised with a bespoke set of parameters. We can think of the pooled components as blueprints for the components which are instantiated once added to an **Experiment**.

The **ExperimentBuilder** allows for any number of tools, wrappers or datasources can be added to an **Experiment** although there it is possible to add only one method or set of results. There are a few special notes relating to the application of the components. For example, a wrapper must be associated with a datasource in order to be successfully added; a wrapper, by definition, must *wrap* something.

Datasources are formed by sets of **DataPoints**, each of which can be further defined with a **DataPointVersion** that records any previous versions from the current version. We have also defined the class, **DatasetController**, which serves two significant purposes. Firstly, it provides the handle for a specific dataset version to any **Experiment** or **Wrapper**, allowing only predefined access to the datasource[20]. The second purpose of the **DatasetController** is to record the **VersionTimetable** for the datasource. The **VersionTimetable** contains details of each individual version of the datasource including column date, reason, column changes and impact. Although mainly for historical purposes, the **VersionTimetable** information also tallies with the versioning information held within **DataPointVersion**.

Finally, of note within the **Model** package is the **Resource** class. We have already described how the component classes represent a blueprint for the component before being instantiated within an **Experiment**. In the **ExperimentBuilder**, when a component is added to an **Experiment**, it generates a **Resource** which, as well as carrying a link to the component, contains a list of **Parameters** to specify the component. The **Resource** object contains all the required information to specify the correct version of the component with a complete set of parameters. **Resources** are generally initiated during the persistence of the experimental environment in order to serialise the components in an **Experiment**. During interaction with the **ExperimentBuilder**, added components are stored in temporary component pools to prevent the need for constantly translating between components and **Resources**.

5.7.3 Versioning, Keywords and Impact

This section provides a brief overview of the versioning methodologies employed in the **ExperimentBuilder**. We devote chapter 6 to describing these areas in more detail but it is important to cover the key points here as well in order to complete the picture of the **ExperimentBuilder**.

The **ExperimentBuilder** is an application designed to handle changes to an experimental environment. We also intend to provide a representation of the experimental environment and for that reason, we must not only internally record changes to components, we should also reflect the changes to the user. To this end, we must retain each

[20]For the sake of simplicity, we connect the **Datasource** to the **Experiment** in figure 5.10, whereas it is actually the **DatasetController** that provides the connection for the dataset.

and every version as we have already noted that updating a component is a user choice that may not always be enacted. Our first endevour should therefore aim at the retention of multiple component versions with a cost-effective method for retrieving previous, non-current versions.

Versioning is handled in two distinct ways within the **ExperimentBuilder**. Dataset versioning is handled by our own **Database Versioning System** (**DBVS**), described in section 5.5.2. The **DBVS** uses a brute force comparison method when encountering a new dataset version to discover any changes. These changes are merged within the affected data point, whilst unaffected data points remain unchanged. The **DBVS** retains all versioning information about a data point without the need for generating, managing or curating redundant data.

The **ExperimentBuilder** handles the other components in a more traditional way. All parts of the current version of a component are kept in their complete form. For previous versions, if there are any, each part of the component, with the exception of title, owner and date, are recorded as a delta from the subsequent version. Using this method, we establish a chain of versions from 1 to n, with version n retained in its entirety and versions 1 to n-1 as backwards deltas from the current version.

The **ExperimentBuilder** can call upon any version of any component within the loaded experimental environment and can add any of these to an **Experiment**[21]. It is clear to see how a new version of a component may affect an **Experiment** that employs a previous version of that component. We expect a new version to have a significant relationship to the previous version and this may well be the case. We can measure this relationship by measuring the similarity between the two versions and use the measure to estimate the impact of any dependant experimentation. It is not only versions of the same component that can relate to one another. Relationships can exist everywhere and not only between components of the same type, for example a dataset can relate to a method. There is a difficulty in defining and, more importantly, measuring a relationship between heterogeneous components because of these differences.

Our methodology for dealing with this difficulty is referred to as *keyword profiling.* For each version of a component, we generate a *keyword profile*, a list of descriptive terms describing the component ordered by significance[22]. Once we have constructed a *keyword profile* for each component version, we can search for relationships between components

[21]A version of a component cannot be deleted. Instead, it is marked as *deprecated.* A deprecated component can no longer be added to an **Experiment** and any **Experiment** containing a deprecated component is flagged as such. Other than the limitation of use within the environment, a deprecated component is handled in exactly the same way as an active component.

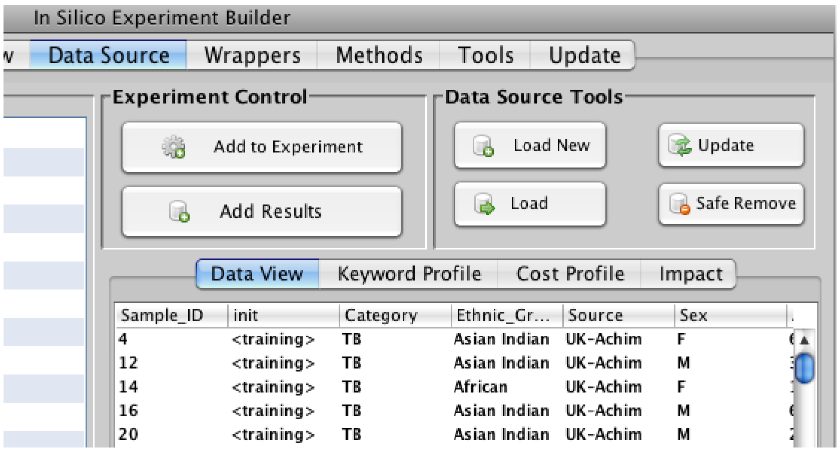
[22]Significance is determined by many factors, including semantic placement of the keyword and the number of occurrences.

that, perhaps, we would not have otherwise identified. This can now be achieved by looking for similarities in the *keyword profiles*, rather than the components themselves to infer about the inter-component relationships. Of course, we can not estimate exactly the nature of the relationship and the **ExperimentBuilder** allows the user to both define new relationships and also tweak the estimated relationships presented by the keyword profiling.

At this point, we build our environment, populate it with components and specify the relationships in between. In order to make the next step and infer impact from the observed changes, we must build a framework that can interpret the change, analyse the relationships of the affected component and model the impact throughout the experimental environment. We have designed several methodologies to estimate the propagation and effects of this impact and these are described full in section 6.4.

5.7.4 User Interface Design

This section describes some aspects of the graphical user interface (**GUI**) of the **ExperimentBuilder** that allow the user to perform some of the actions described above. We present aspects of the **GUI** that relate to environmental building and component construction. Aspects of the **GUI** that relate to versioning, keyword profiling or impact estimation are presented in chapter 6.



Sample_ID	init	Category	Ethnic_Gr...	Source	Sex
4	<training>	TB	Asian Indian	UK-Achim	F
12	<training>	TB	Asian Indian	UK-Achim	M
14	<training>	TB	African	UK-Achim	F
16	<training>	TB	Asian Indian	UK-Achim	M
20	<training>	TB	Asian Indian	UK-Achim	M

FIGURE 5.11: Datasource construction within the **ExperimentBuilder**.

Before beginning to model the experimentation, we establish the environment and this requires constructing components to mirror those in the real world. There are obvious similarities in the construction of the experimental components but there are some specific differences depending on the type of component and the **ExperimentBuilder** provides the tools needed to create user-defined, unique components. Users can add **Datasources**, **Wrappers**, **Methods**, and **Tools** to the **ExperimentBuilder**. Given the requirement to compare heterogeneous components, there are similarities in the way that the component structure and impact relationships are presented.

The **Datasources** tab of the **ExperimentBuilder**, illustrated in figure 5.11, houses the tools for constructing and using datasources. The **ExperimentBuilder** accepts data in multiple formats. The default format for datasource entry is a generic, tab-delimited text file. The first line of the text file must contain a list of tab-delimited column names. This is followed by a user-limited[23] number of tab-delimited data lines.

The **ExperimentBuilder** allows the user to upload entire datasources into the experimental environment but is more suited to handling derived experimental datasets[24] rather than large-scale datasources. This is due to the extensive profiling undertaken when a component is added, to establish inter-component relationships. For very large datasources, this can take a long time and can produce erroneous relationship estimates as the chances of finding false relationships is greater. This is discussed in more detail later in chapter 6.

In comparison with experimental datasets, the other components can be added either in their entirety, if the user has the source code, or the user can generate a representation of the component. This representation is user-defined and can use any selection of available metadata but it facilitates the tracking of the component if a similar method for representing components is used across for all components. In addition to a bespoke, textual representation of the component, the **ExperimentBuilder** offers a user-defined list of parameters for each version of a component.

Having defined the experimental environment, we can begin building the experiments. Users can choose how to add experiments to the **ExperimentBuilder**; experiments can be added sequentially, chronologically or, as is recommended, in a hierarchical structure based on experimental hypothesis. The **ExperimentBuilder** promotes the hierarchical presentation of experiments by allowing the creation of *Lines of Enquiry* (**LOEs**). **LOEs** can be nested within other **LOEs** in order to create a directory-structure of experimentation. This is illustrated in figure 5.12.

Experiments can be added to any *Line of Enquiry*[25] and once they have been defined, the user can start adding experimental components. When a component is added, the user specifies values to any parameters that have been defined for that component. The parameter values are experiment-specific and the instance of the component is created within the **Experiment** as a **Resource**, as in figure 5.10. As components are added,

[23]The dataset size is limited by the manageability and the performance of the **ExperimentBuilder**.

[24]We refer to datasets generated by querying larger datasources.

[25]The only stipulation for an experiment is that is must be added to a *Line of Enquiry*; i.e., it cannot be added to the root.

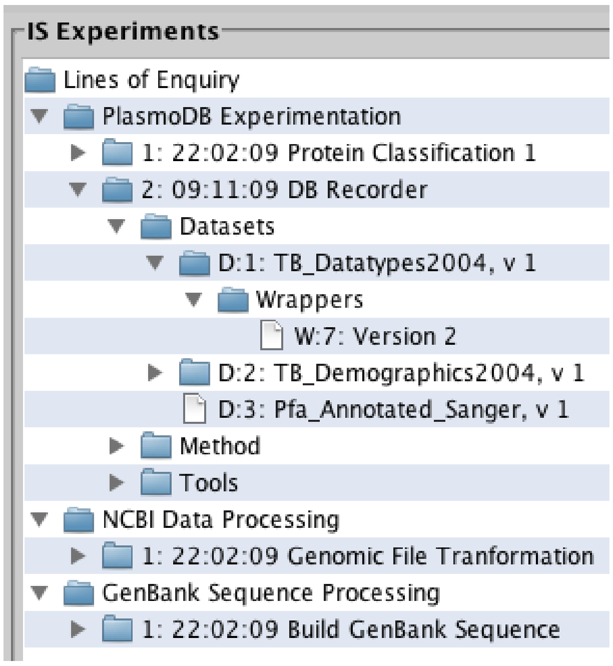

IS Experiments
Lines of Enquiry
PlasmoDB Experimentation
1: 22:02:09 Protein Classification 1
2: 09:11:09 DB Recorder
Datasets
D:1: TB_Datatypes2004, v 1
Wrappers
W:7: Version 2
D:2: TB_Demographics2004, v 1
D:3: Pfa_Annotated_Sanger, v 1
Method
Tools
NCBI Data Processing
1: 22:02:09 Genomic File Tranformation
GenBank Sequence Processing
1: 22:02:09 Build GenBank Sequence

FIGURE 5.12: The environmental structure as presented by the **ExperimentBuilder**.

the individual component dependencies are passed on to the experiment. When a change occurs in a component, the change can affect a dependant component and if any of the dependants exist in an experiment, then we can interpret the impact on the experiment as an aggregated whole.

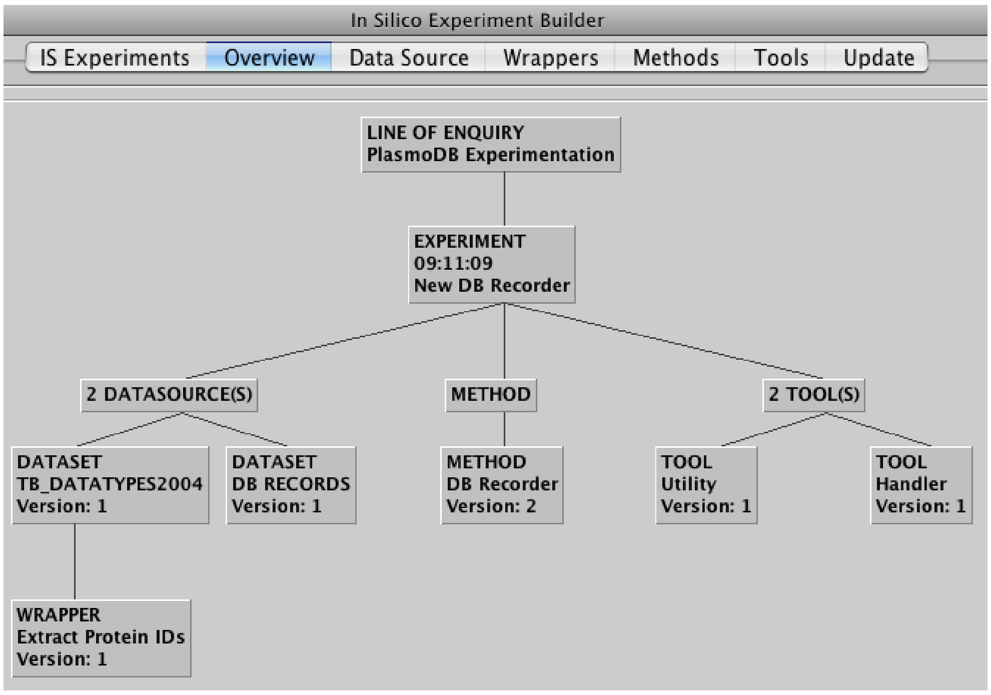

In Silico Experiment Builder
IS Experiments
Overview
Data Source
Wrappers
Methods
Tools
Update
LINE OF ENQUIRY PlasmoDB Experimentation
EXPERIMENT 09:11:09 New DB Recorder
2 DATASOURCE(S)
METHOD
2 TOOL(S)
DATASET TB_DATATYPES2004 Version: 1
DATASET DB RECORDS Version: 1
METHOD DB Recorder Version: 2
TOOL Utility Version: 1
TOOL Handler Version: 1
WRAPPER Extract Protein IDs Version: 1

FIGURE 5.13: The graphical representation of a single experiment provided within the **ExperimentBuilder**.

As experiments are defined and populated with components, the visual overview of the experiment, provided by the **ExperimentBuilder**, is updated to reflect the new state of the experiment. An example of the overview can be seen in figure 5.13.

Chapter 6

Tracking Impact

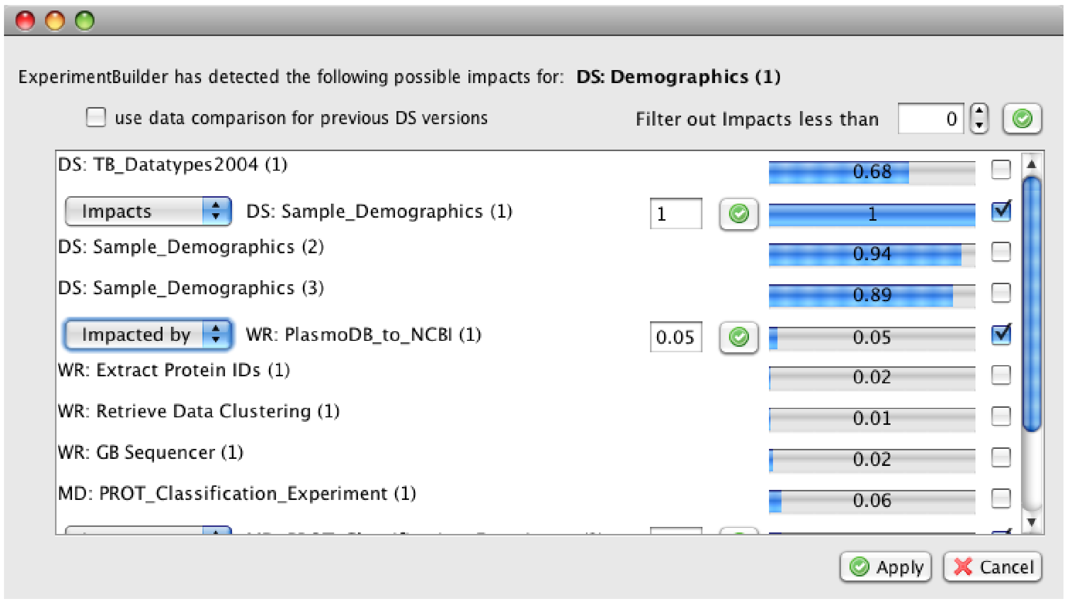
6.1 Component Relationships

We can imagine with little difficulty how one component can relate to another and we understand the need to devise methods for measuring this relationship. We can adapt existing traditional versioning methods to deal with some aspects of simple relation-mapping and we have employed some of these methods in the **ExperimentBuilder**. These methods can be extremely effective in the right domain, but the efficacy of such approaches diminishes when we try to apply them outside of their intended area.

In order to identify all relationships within an experimental environment, we must compare components of all types. We must compare datasets to not only other datasets, but methods and tools. But how can you compare a dataset to a method? We explore the idea of heterogeneous comparison further in the next section. But before starting to think how to make component comparisons, we should consider the elements that we need to include in order to make an appropriate comparison. To define a relationship between one component to another, we look for those elements that, when changed in one component, cause a change in another. The nature of a change can have varied effects on the impact on dependant components and it is this nature that we need to record in order to accurately estimate its dependant network propagation.

There is also a significant distinction to make regarding the relationship between components and the effect this relationship has in estimating the impact between them. A relationship between components is a measure of the similarity between them. This relationship can be used in order to define the likely propagation network of a change originating at a particular component. For example, if we identify a relationship between two components, we could assign a value to this relationship of 0.1 meaning that the two components share 10% of the same or related content. It is important to clarify that this does not mean that change in one component will result in a 10% change in the other. It simply indicates the degree of similitude between the two components.

In order to establish how a change in one component affects a dependant component, we must derive more information about the change itself. A change could occur in the 90% of the component that is unrelated and in that case, there should be no impact passed on to the dependant component. Assigning a value to a relationship can be misleading, suggesting a degree of impact but the impact can only be determined once we have identified and analysed the initial change. The value we provide for a inter-component relationship is the degree of similarity only and we will explain later in this chapter how this measurement is used in impact estimation and, more importantly, in establishing the impact propagation network.


ExperimentBuilder has detected the following possible impacts for: **DS: Demographics (1)**

☐ use data comparison for previous DS versions — Filter out Impacts less than 0

DS: TB_Datatypes2004 (1)			0.68	☐
Impacts	DS: Sample_Demographics (1)	1	1	✓
DS: Sample_Demographics (2)			0.94	☐
DS: Sample_Demographics (3)			0.89	☐
Impacted by	WR: PlasmoDB_to_NCBI (1)	0.05	0.05	✓
WR: Extract Protein IDs (1)			0.02	☐
WR: Retrieve Data Clustering (1)			0.01	☐
WR: GB Sequencer (1)			0.02	☐
MD: PROT_Classification_Experiment (1)			0.06	☐

FIGURE 6.1: Specifying the relationship profile for a new component within the **ExperimentBuilder**.

There are some logistical problems to consider for the establishment of inter-component relationships. Some components will relate to other components coincidentally. Conversely, some components will relate to others despite a lack of any physical commonality. Finding relationships in components depends heavily on the nature of the experimentation and some relationships can only be identified by those who are familiar with the experimental environment. But, we cannot rely on the user to define an entire environment of inter-component relationships. We have only to consider the possible relationships between, for example, ten components to realise that it would take a prohibitively long time to define by hand[1] and we cannot expect users to provide this level

[1]There are 45 possible relationships between 10 components.

of input. We require a system that can estimate, as far as possible, the relationship of two components, based on some internal methodology, and present this estimation to the user who can tweak the estimate to suit their understanding of the true relationship.

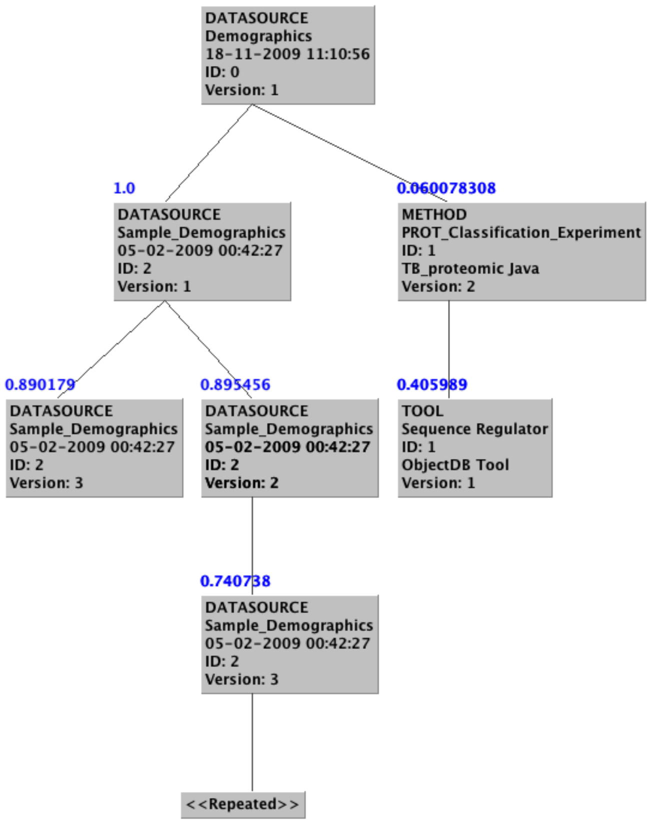


FIGURE 6.2: The component relationships as established within the **Experiment-Builder**.

The key to measuring change and tracking impact starts with the accurate identification of the relationships between components. We must know how a component relates to another so that we can know how a change in one will affect another. Much of the work presented in this chapter centers on the estimation of impact and, for further propagation of the impact, the estimation of knock-on changes that may occur as a result.

6.2 Keyword Profiling

We have described, in the previous section, some of the aspects and issues of defining relationships between experimental components. We assumed a solution for defining relationships in order to discuss some global issues but in this section, we describe our methodology for identifying and defining inter-component relationship and we refer to this technique as *keyword profiling*.

We have already touched on our definition of a specific component relationship. We define the relationship as the degree of similarity between two components. We can use this measurement together with the change to estimate the total impact propagation within the environment. We can not provide a definitive value for impact alone and, therefore, can not define a static impact propagation network. Impact can only be established once we have established a change.

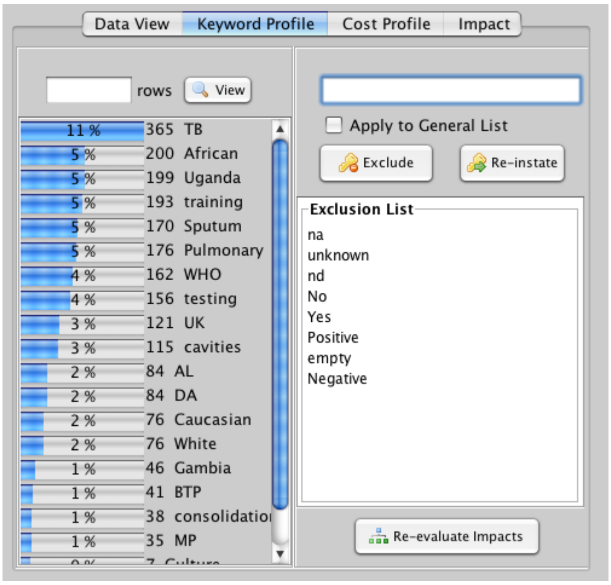
So, how do we define a similarity between components. If we consider two versions of a datasource, we can identify common places in the component that we can measure. For example, we can measure the similarities between field names, column and row numbers and with adequate techniques, we can measure the difference between individual data values. We could expand these techniques to include all datasources but, given our representation of experimental components within the **ExperimentBuilder**, we need to identify and quantify relationships between all components, irrespective of type. There are various aspects of components that need to be considered when identifying relationships between components. Much emphasis should be placed on the content of the component and the metadata of a component can often be an effective way of representing content. There are, however, circumstances where the component content will give little or no evidence to suggest a relationship between the content of another. We must delve further by analysing the semantics of the component; the meaning of the component and the experimental importance as defined by other similar experimental components. We must also consider the possibility of two components that exhibit very little or no similarity but maintain a significant experimental relationship. This scenario is difficult to anticipate and, in most cases, this relationship must be defined by hand.

Keyword Profiling works by extracting a set of key words from a component and builds a list, ordered by semantic importance and frequency. The technique is applied in a slightly different way according to the type of the component but the end product is always the same. A *keyword profile* for one component version can be measured against that of any other component version. *Keyword Profiling* occurs when any new version of a component is added to the **ExperimentBuilder**. There is no user input during this period although the user can choose to exclude certain keywords.

The Keyword Profile Generation Algorithm:

```
begin
  proc generateKWP($(component, exclusionList)$) $\equiv$
    $parts[] = component.split();$
    for $each(part : parts)$ do
        $weight := lookupWeightForPart();$
        $extractKeywords(part, weight, exclusionList);$
    od
  .
  proc extractKeywords($(source, totalWeight, exclusionList)$) $\equiv$
    $regex := [(Punct), (Blank), (Space), (Digit), (Cntrl)];$
    $keywords[] := source.split(regex);$
    $singleWeight := totalWeight/keywords.length;$
    for $each(kw : keywords)$ do
        if $(!kw.inExclusionList())and(kw \geq minSize)$
          $\rightarrow addKeyword(kw, singeWeight);$
        fi od
  .
  proc addKeyword($(kw, weight)$) $\equiv$
    if $(!kw.existsInProfile())$
      $\rightarrow profile.addKeyword(kw, weight);$
     else
          $\rightarrow profile.getKeyword(kw).adjustWeight(weight);$
             $\rightarrow profile.getKeyword(kw).incrementOccurrences();$
          fi
      .
    end
```

The user can define a list of words that are to be excluded from the keyword profiling. There are two levels of user-defined keyword exclusion. There is a general keyword exclusion list, containing all words that are to be disregarded during all keyword profiling. This dictionary is likely to contain everyday linkage words that have no specific semantic meaning or, alternatively, meaningful words that, in the context of the specific experimental environment, are so common that they possess very little meaning and serve only to distort the inter-component relationships. The user can also specify exclusion lists for individual components. Such exclusion lists are used together with the general exclusion list during the keyword profiling of a component, allowing the user some degree of control over the profile in order to improve its efficacy.



11 %	365	TB
5 %	200	African
5 %	199	Uganda
5 %	193	training
5 %	170	Sputum
5 %	176	Pulmonary
4 %	162	WHO
4 %	156	testing
3 %	121	UK
3 %	115	cavities
2 %	84	AL
2 %	84	DA
2 %	76	Caucasian
2 %	76	White
1 %	46	Gambia
1 %	41	BTP
1 %	38	consolidatio
1 %	35	MP

Exclusion List
na
unknown
nd
No
Yes
Positive
empty
Negative

FIGURE 6.3: The representation of a *Keyword Profile* in the **ExperimentBuilder**.

Strings are identified from the component and added to a list of keywords, either as a new keyword in the event of a new string or appended as an occurrence to an existing keyword. Strings are only added as keywords if the string does not appear on any of the relevant keyword exclusion lists. At this point, the keyword lists are normalised so that the total weight for all keywords in a keyword profile add to one. Finally, the list is sorted according to normalised weight and presented to the user as illustrated by figure 6.3.

Keyword profiling works in a manner dependant on the type of component. That is to say, the *keyword profile* for a datasource version is constructed in a different manner to that of a method version by virtue of the inherent differences between the components. Datasources are broken down into three main categories; data content; data semantics (fields, etc.); and component semantics (description, notes, etc.). Methods, tools and wrappers are split differently according to their metadata representation, or in some cases the actual representation. In either case, the different parts of the component are weighted according to their perceived importance dictated by the user. For example, a keyword that appears in the title of a component is *worth* more than a keyword that appears in the *bulk* of the component. If it appears in both areas, there is a cumulative effect for the keyword that grows stronger according to the frequency and positioning.

Currently, this profiling technique ignores numerical values and any string value less than two characters in length. We ignore numerical values as they contain little in the way of semantic information that can be measured between components. Numerical values also contain very little interpretive value unless they are bound with the information they are describing, which in the case of *keyword profiling*, they are not. We can foresee a problem when encountering a dataset containing mostly, or entirely, numerical values as for this case, the *keyword profile* will be semantically weak. In such cases, it may be necessary for the user to forego the *keyword profiling* techniques and specify the relationship by hand. There is also the issue of using keyword profiles to detect change in components[2] when the component contains largely numerical values. This is particularly troublesome for datasets as numerical values are not represented in the *keyword profile* and we have therefore implemented a user-selectable, brute-force comparison method for comparing datasets of this type.

The *keyword profile* allows us to represent the content and nature of a component, regardless of type, and measure it against the profile of any other component, therefore determining the degree of similarity. In the following section, we go on to describe the importance of determining this relationship and we use the *keyword profile* to, not only identify the similarities between components, but characterise and represent detected changes, estimate impact from the change and, finally, propagate the impact throughout the experimental network.

6.3 Looking for Change

As we continue to use the **ExperimentBuilder** to define components, experiments and establish the relationships therein, our environment will grow in order to mimic the real-world experimental environment. At this stage, the environment exists in a static state without change. The components are related to one another but until there is a change, the environment stands still. At the point a change occurs, we can begin to use the tools described in the following sections in order to identify, track and estimate the resulting impact.

Changes are detected in the same way for all components. The **ExperimentBuilder** detects change in a component by measuring the *keyword profile* of the new component version against that of the original component version[3]. Profile comparison for change detection initially works in the same way as the component relationship identification described above. For change detection, we identify the change from one version to the next using profile comparison and then we create a *keyword profile* that encapsulates the change itself. The *keyword profile* of the difference between component versions, or the *delta keyword profile (dkwp)*[4], contains all the keywords, ordered by frequency and semantic seniority, within the change.

[2]This can often normally lead to surprisingly accurate results as we shall see in chapter 7.

[3]Change detection in the **ExperimentBuilder** relies on user submission of the new component version, or the 'suspected' new version.

[4]Please refer to the following section for a description of *keyword profiles* and *delta keyword profiles*.

What follows is a representation of the Algorithm for comparing keyword profiles and generation of the *dkwp.* We combine the keyword profile comparison and the *dkwp* generation here as they require similar algorithms. For the sake of simplicity, they are included here together but in the implementation of the **ExperimentBuilder**, they are not so tightly coupled.

The Algorithm for Comparing two Keyword Profiles:

```
begin
  proc compareKWPs($kwpA, kwpB$) $\equiv$
    $Keyword$[]hits;
    $Keyword$[]misses;
    $KeywordProfile$deltakwp;
    for $each(keyword : kwpA)$ do
        $buildStat(keyword, kwpA)$;
    od
    for $each(keyword : kwpB)$ do
        $buildStat(keyword, kwpB)$;
    od
    $comparisonFactor = (hits : misses)$;
    $return$comparisonFactor;
  .
  proc buildStat($keyword, kwp$) $\equiv$
    if $(kwpB.contains(keyword))$
      $averageWeight = (kwpA.keyword.getWeight + kwpA.keyword.getWeight)/2$;
      $weightDiff = diff(kwpA.keyword.getWeight, kwpA.keyword.getWeight)$;
      $hits.add(keyword, weightComb)$;
      if $(weightDiff! = 0)$
        $deltakwp.add(keyword, weightDiff)$;
    else
        $misses.add(keyword, kwpA.keyword.getWeight)$;
      fi
    fi
  .
```

The encapsulation of the component change, embodied in the *dkwp*, is important so that we can pass the change on to dependant components and infer, with some precision, how that change might cause impacts. These inferences are, again, achieved by comparing *keyword profiles.* The *dkwp* is compared against the *kwp* of the dependant component and the resulting comparison is interpreted by the **ExperimentBuilder** and presented as impact.

6.4 Propagating Impact

We must consider three principal terms of the change management system employed in the **ExperimentBuilder**; **change**, **impact** and **cost**[5]. Components are revised, improved or updated from one moment to the next and the transition between states is referred to as **change**. A change in one component *may* have an effect on another component. If it does, the resulting change in the dependant component is referred to as the **impact**. There may be many **impacts** that occur as a result of a single change and we can track these through a propagation network of dependant components. When we sum the impacts that result from a single change and calculate the effort required in updating the components, we refer to the cumulative effort as **cost**.

Impact is the result of an identified change. Impact can be expressed either as a value for a specific component or as a sum of the total propagated impact as a result of a single change; change leads to impacts, which leads to further impacts and so on. Impact cannot be adequately described through arbitrary values however. We need to attribute a cost to the required implementation occurring as a result of the change. That is the true impact. We discuss our representation of cost and how it relates to change and impact in the next section.

In order to understand how impact flows through our environment, we must revisit our use of keyword profiles. We have already discussed the difficulties in comparing components, in particular those components of different type. We have employed a universal keyword profile mechanism that generates a comparable and standardised representation of each component, allowing us to establish inter-component relationships. So, when we refer to one component relating to another, we are really talking about the degree of similarity between the *keyword profiles* for the two components. This is illustrated in figure 6.4.

Within the **ExperimentBuilder**, component keyword profiles are used exclusively for change detection, impact propagation and cost calculation. Any contextual information about the component that is not contained within the keyword profile is not accounted for during any of these processes.

Keyword profiles enable us to model the context of components regardless of type so they can be measured against one another in order to identify inter-component relationships. A *delta keyword profile* differs by specifying the presence or absence of a keyword as part of a change in one component from one state to the next. We can use the component *kwp*s and δkwps to propagate and estimate and the impact to the total experimental environment.

[5]These are our definitions and not intended to supercede or replace any existing terminology.

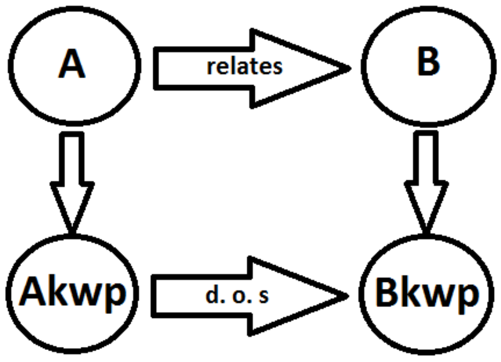


FIGURE 6.4: Inter-component relationships and the resulting degree of similarity between keyword profiles.

The impact can only be adequately expressed if we can estimate the perceived effort required to enact the identified change. In the following section, we describe the way in which we encapsulate both user and environmental efforts and present them through the **ExperimentBuilder**.

Given our dependence on keyword profiles to define inter-component relationships, we should employ a change detection strategy that can take full advantage. Many components exist in a near-constant state of change. During any one of these changes, the component will change state from *A* to *A'* and we can define the transitional change as the difference between them. We are, for the moment, unconcerned with the origin or nature of the change, only its effect on the current state of the component. Inter-component relationships are estimated by analysing the similarities between the component keyword profiles. We can use a similar technique to look for dissimilarities between the component states before and after the change. The result is an ordered list, similar to that of a component profile, but instead contains the differences between one keyword profile and another. We use this technique to embody a component change and we refer to it as the *delta keyword profile* (δkwp).

A *delta keyword profile* (δkwp) contains, as per figure 6.5, the differences between the *kwp* for components *A* and *A'*. Within the δkwp, we record those keywords present in *A* and not in *A'* and vice-versa. But the δkwp cannot contain only differences. We must differentiate between additions and subtractions to the component during the update. This is important so that we can accurately estimate how the change affects dependant components. If the δkwp were to contain only ordered keywords, as in a regular *kwp*, we could not know whether an identified keyword would represent an absence in the new component or a presence. When a keyword is added to the δkwp, we record whether is it an addition or a subtraction as well as the strength and frequency as per an original *keyword profile*.

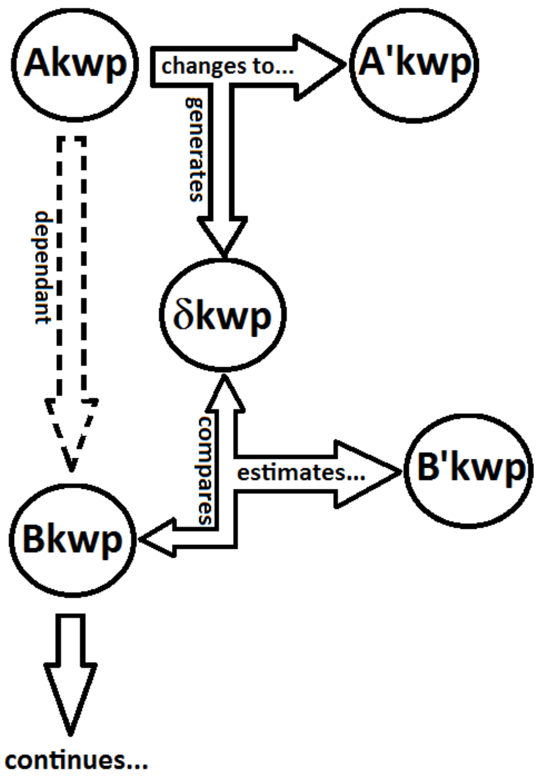


FIGURE 6.5: Impact estimation and propagation and use of the *delta keyword profile* (δkwp).

Represent a component change in the structure of a keyword profile is a particularly useful method for determining the propagation of the impact. Not all dependant components will be affected; there will be components that are related but, due to the specifics of the change, are not affected. We can compare the δkwp against each dependent component *kwp* to see if the change will, in reality, have an effect. If there is a significant comparison between the δkwp and the *kwp* of the dependant component, it indicates that whatever change occurred in component A, it affects parts of the component that have also been identified as occurring in component B[6]. Given this information, we can now state with some confidence that component B is affected. We can go on to determine the effect of the change, the estimated impact and decide whether the impact should propagate further.

If we compare the δkwp against the *kwp* of the dependant component B, we can estimate the *kwp* for the updated version of that component, *B'*, illustrated in figure 6.5. Whether the δkwp keyword is an addition or a subtraction specifies whether the keyword is added or removed from the *kwp* of the newly estimated component, *B'*. Component B may

[6]Referring back to figure 6.4.

change more or less than expected from the inter-component relationship as the degree of impact depends on the nature of the change, represented in the *δkwp*. This method allow the impact to dependant components to be more accurately modelled, basing the propagated impact on the *δkwp* rather than a numerical approximation of the change.

Once we have estimated the *kwp* for *B'*, we can repeat the process for all identified dependants of component *B*. We start, as before, by generating the *δkwp* for the estimated change between *B* and *B'*. This differs slightly than before in the respect that the *B'kwp* is estimated rather than explicitly known. Remember that impact propagation is use to calculate the cost of updating. Based on this estimation, the user will then decide whether it is worth the required effort to update. At this stage, we cannot know *B'* explicitly. Consequently, as we delve further into the propagation network, the estimations of the updated components become less accurate.

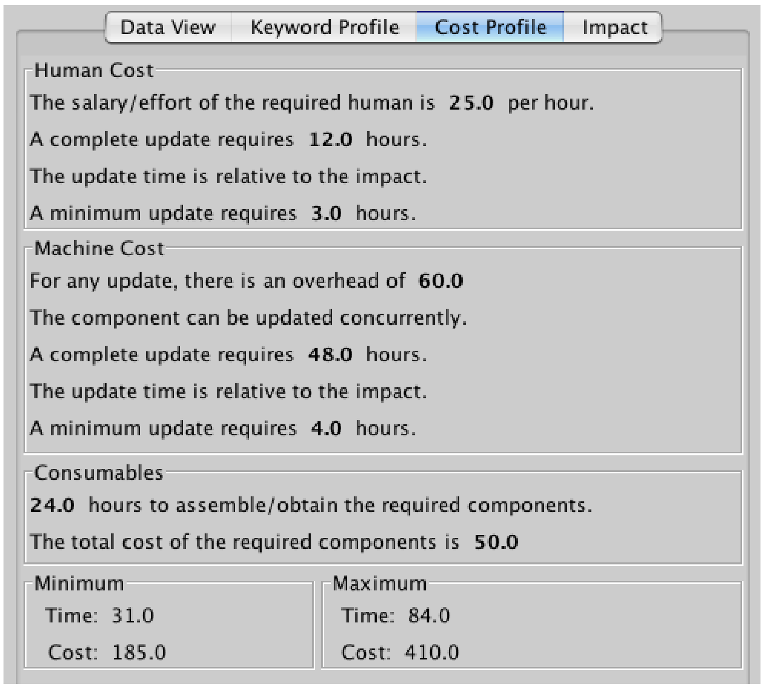
6.5 Conveying Impact with Cost

In the previous sections, we have presented our encapsulation of change and impact so that we can identify, quantify and propagate them within the experimental environment. We have referred to impact within the dependant network, assigning numerical values or modelling with keyword profiles. But what do these values mean? In terms of assessing the impact, they mean very little. We need to quantify impact in a meaningful way so it can be used constructively within an experimental system.

We are measuring impact so we can make informed decisions about whether to update our experimental environment by re-running experiments. Experiments may take longer than others and some may involve many iterations of previous experimentation in order to achieve an updated result. In the process of making the decision to update, we should present the user not with arbitrary values but with a real world estimation of the efforts required.

The *cost profile* for a component captures details about the effort required to update, relating to the degree of change placed on the component. A *cost profile* is created for a component when it is added to the experimental environment in the **Experiment-Builder**. The user is prompted to enter details about the component relating to the degree of effort required to update. The *cost profile* contains various details of the time and cost of both human and machine activities required in updating the component and this are illustrated in figure 6.6.

Cost is split into three categories; human cost, machine cost, and consumable effort. Human cost refers to the amount of time required by the user to update the component.



Human Cost

The salary/effort of the required human is **25.0** per hour.

A complete update requires **12.0** hours.

The update time is relative to the impact.

A minimum update requires **3.0** hours.

Machine Cost

For any update, there is an overhead of **60.0**

The component can be updated concurrently.

A complete update requires **48.0** hours.

The update time is relative to the impact.

A minimum update requires **4.0** hours.

Consumables

24.0 hours to assemble/obtain the required components.

The total cost of the required components is **50.0**

Minimum	Maximum
Time: 31.0	Time: 84.0
Cost: 185.0	Cost: 410.0

FIGURE 6.6: A *cost profile* for a single experimental component.

We make reference to the level of user required to complete the update; can the update be completed by anyone or does it require interaction from the principal scientist? The time of some users is more valuable than others and the *cost profile* attempts to capture this by assigning an hourly unit amount to user interaction. Machine cost is measured in terms of time alone although we include an overhead cost in order to cover any additional costs involved. We can set the human and machine aspects of the update to run either consecutively or concurrently and this has obvious effects on the time required for the update. In terms of machine cost, the concurrency of the component has a significant effect, signifying that the component and, therefore, the experiment, can be executed concurrently with other possible experiments in the update process.

We provide a *consumables* section which includes any logistical time or cost involved in actioning the update. For example, an update may require a new version of a component to be purchased, in which case it would require a purchase cost and a delivery, download or installation time. The final outcome for the *cost profile* is a description of the minimum and maximum effort required for a component update. The cost of the component update can be set as relative to the impact or not. For example, a small update will require less time and cost than a much larger update. This is likely to be the case for bespoke components that may be 'tweaked' as part of the update but for larger commercial products, there may be only one level of update; i.e., a complete update.

The cost of updating the component is a product of the weight of the impact and the relative cost of the update. The **ExperimentBuilder** identifies a change and propagates the impact to the identified dependant components. The estimated impact to each dependant component is weighed against the *cost profile* to produce the estimated cost of the update. The costs of the dependant component updates are combined, taking into account concurrencies where appropriate, and presented to the user as the **total component update cost**.

The user can now decide whether to continue the update depending on their evaluation of the estimated cost. The user can also choose to enact only certain aspects of the update, allowing fine tuning of the update and the avoidance of weighty, expensive, or unimportant elements of the update.

Chapter 7

Validation

This chapter describes the evaluation of the two applications developed for the purpose of the research. The **Hashed Data Model (HDM)** represents a first attempt at monitoring change between datasets and inferring some semantic meaning between versions. The change management framework and the main focus of the research, encapsulated by the **ExperimentBuilder**, is evaluated in section 7.2. A summary of this validation and of the research as a whole can be found in section 8.2.

7.1 The Hashed Data Model

Chapter 4 presents the **HDM** designed to help e-scientists identify, track and manage these types of change. The **HDM** is a tool that abstracts an experimental dataset and metadata to a model that can be used to detect changes between versions of datasets. The characteristics of the **HDM** allow the user to potentially retain a unique version of the dataset for every single experiment.

7.1.1 Case Study Performance

With a granularity factor of one, the HDM will contain an alphanumeric representation of every value of each column. We could theoretically record the state of each row and detect change on an absolute basis. This cannot occur, however, as the alphanumeric hashing will inevitably produce collisions. A sample of the tracking-enabled results can be seen in figure 4.3. At granularities higher than one, it is not currently possible to detect the absolute degree of change within the specified group of data values. For example, consider a HDM with a granularity of five, where each alphanumeric hash value represents a group of five rows of the column. Ignoring the possibility of collisions,

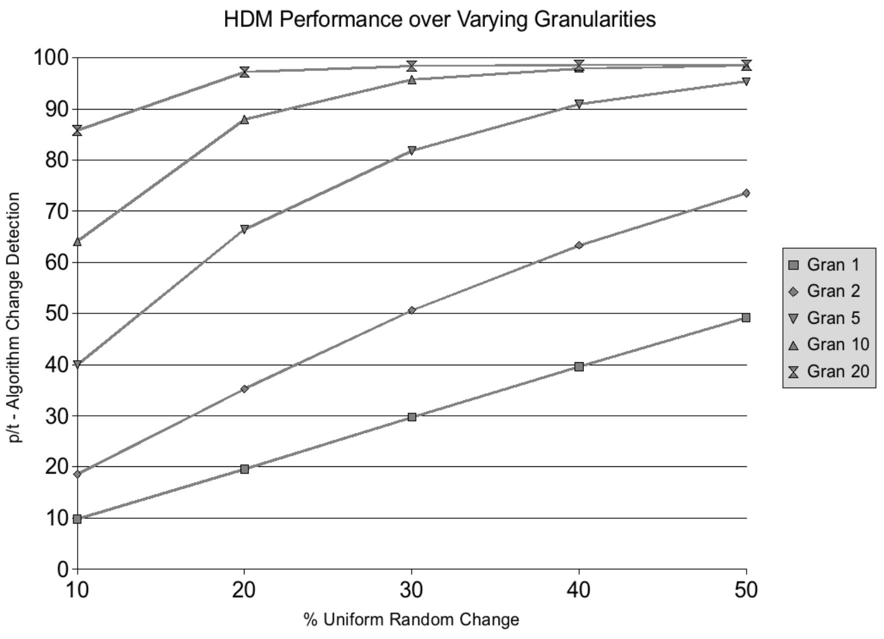


FIGURE 7.1: **HDM** tracking results at varying levels of uniform randomisation.

a change in one of the values within the group will affect the hash value as strongly as a change in two, more or even all of the values. In this case the strength and the impact of an altered alphanumeric value must be estimated. At higher granularities, the estimation method is increasingly important in order to accurately estimate the change.

Figure 7.1 illustrates the performance of the **HDM** at varying degrees of granularity. The test datasets were constructed using an algorithm generating uniformly random changes to the dataset from between 10 and 50 percent. The current **HDM** algorithm estimates change as the number of changed data blocks as compared to the total number of data blocks (p/t). For example, a column with 100 rows at a granularity of one, will have 100 data blocks. At a granularity of 5, it will have only 20 data blocks. As the number of data blocks decreases, the poorer the **HDM** change detection algorithm performs and the more chance a randomly altered dataset will nullify the detection algorithm, giving results ever nearer to 100%.

We can see that at a granularity of one, the **HDM** change detection performs very closely to the actual percentage change. As the granularity increases, the performance of the **HDM** is reduced to a point where only very limited information of the data change can be inferred. It is also worth noting that, regardless of the granularity, at no point is change under-estimated. Any change, no matter how small, is detected.

7.1.2 Limitations of the HDM

The **HDM** prototype can be shown to work within an existing biological domain with real biological data but the **HDM** is only an initial step towards a comprehensive framework for change management. In order to provide real usability for biologists, we must add further functionality. The **DBVS** provides the engine for versioning the scientific data but for a complete versioning framework, we must consider both fine and coarse-grained investigation of the changes that can and frequently do occur.

The **HDM** provides an abstraction of the dataset and therefore, we can never recover the dataset in its original form. This is a problem if we need to investigate the changes on a fine-grained scale. Consider the **HDM** with a granularity factor of one. Even at this level of granularity, we can not infer what the data values were and what they are now. For many cases, this is an acceptable level of granularity but for very fine-grained investigation, we envisage the addition of a versioned database system, possibly the **DBVS**, containing the data sources used for model creation and *in silico* experimentation. The version of the dataset used can be linked to the **HDM** model so that upon discovering changes in the **HDM**, the versioned database can be automatically queried to return the exact nature of the changes. This will provide a level of control and a depth of investigation simply not possible with the **HDM** alone.

7.2 The ExperimentBuilder

The **ExperimentBuilder** is the next step and the result of continued analysis of the case study, from the **Hashed Data Model**. The **HDM** concentrates on changes to datasets only, the **HDM** abstraction being unreversible losing the original form of the data, and therein lies its main limitation. The **ExperimentBuilder** aims to overcome this limitation by providing a complete encapsulation of the experimental environment, providing change management for all the experimental components therein.

Within this section, we revisit the main features of the **ExperimentBuilder**, describing both the respective merits and limitations. The case study described in chapter 3 provided the stimulus for the research and, ultimately, the **ExperimentBuilder** and we shall refer back to it frequently throughout this section.

7.2.1 Evaluation of the ExperimentBuilder

The evaluation of the **ExperimentBuilder** starts with a redress of the principal characteristics of the system, providing a critical explanation of each unique facet of the system that enables change management. We then describe some of the problems associated with providing change management and the degree to which these are satisfied by the **ExperimentBuilder**. We discuss some of the limitations with the presentation of results, namely the real-world costs of updating an experimental environment. Finally, we present the performance of the **ExperimentBuilder**, describing the efficacy of the keyword profiling approach.

7.2.1.1 Complex Experimental Environment

The primary requirements for the **ExperimentBuilder** are the construction, integration and management of experimental components. Apart from the experiments, which can themselves be considered components, there are four types of experimental component, established primarily from a direct analysis of the research case study. The *datasource* component encapsulates a component containing any aspect of experimental data, although a *datasource* should not contain any computational logic. Logic components[1] are split into three distinct groups; a *wrapper* representing to logic that is explicitly attached to a *datasource*, a *method* which represents the central transformation or purpose of the experiment, and experimental *tools* which describe any further steps that are required to complete the designated purpose of the experiment. Any number of *datasources*, *wrappers* or *tools* can be added to the experiment but the user must specify only one *method* per experiment.

Through extensive analysis of the case study, we believe that the complement of components supplied by the **ExperimentBuilder** allows the construction of any kind of required experimentation, including those identified in the case study. The representation of a component within the **ExperimentBuilder** is left largely to the design choices of the user. Component source code can be incorporated directly into the **ExperimentBuilder** and is a logically reasonable representation for the content of some components. The ability to specify component source code does not preclude the possibility that a bespoke representation of the component may be more useful. The **ExperimentBuilder** allows the user to enter any desired representation for a component and this provides a level of freedom to express the nature of the component, unavailable with source code alone.

There are several ways in which users can enter data into the **ExperimentBuilder**. With respect to the biological data, we provide several methods for data entry. The foremost format for entering data is a regular tab-separated data string. Data items are provided in a tab-separated text file with the first unbroken line containing column

[1]A logic component refers to the presence of some computational logic, programming or bundled functionality that adds to the experiment.

names. The heterogeneity of data sources brings several issues into focus, in particular the varying formats that inevitably arise. Accommodating for all, or even many, of the existing data formats is a considerable challenge, outside of the explicit scope of this research. It is intended that, as data formats become necessary for the progression of the experimental environment, they can be added to the application as a patch or a retro-fit, and this has been facilitated through the modular development of the application. Nevertheless, as well as a simple data entry, we provide data entry methods for two specific data types; microarray and genomic[2].

In section 1.4, we introduced the idea of *topological importance* whereby different parts of a dataset hold varying levels of importance. The **ExperimentBuilder** allows the user to define keyword weighting to an extent; the user can specify the semantic value of various areas within the individual dataset so a keyword in one semantic area can have a greater weighting than another in a different area. Also, users can remove keywords from the component *keyword profile* if the weighting unfairly represents the component. Keywords can be excluded from the individual component profile or from the whole experimental environment altogether. At present, the **ExperimentBuilder** does not allow the user to manually add keywords to the *keyword profile.* If the keyword is not present in the component, it can not appear in the *keyword profile.*

As components are added to the **ExperimentBuilder**, they are measured, using *keyword profiling*, against the existing experimental components in order to identify dependencies within the environment. This is an important part in the process of change management. An experimental environment may contain hundreds or even thousands of experimental components, making it infeasible to manually define each inter-component relationship and to do so would be prohibitively time consuming. The **ExperimentBuilder** estimates the relationship of the new component against the existing environment by automatically comparing the *keyword profiles* of each existing experimental component. Based on similarities between *keyword profiles*, we can identify potential dependencies for the new component. The user can specify limits for the identification of relationships; the degree of similarity that warrants an identified relationship. Dependencies can be specified as unidirectional, to impact to be impacted by the environment, or bi-directional. It is generally preferred for the user to specify dependencies as one-way, as too many bidirectional relationships can impede the efficacy of the impact propagation algorithms.

The **ExperimentBuilder** allows the user to manually redefine inter-component relationships in order to rectify any discrepancies in the relationship estimation. We have

[2]These are in the form of in-house scripts logically outside the **ExperimentBuilder** and represent the ongoing development process within the application.

already briefly mentioned some of the limitations of *keyword profiles* and these are explored in more depth in section 7.2.2.2. Suffice to mention here that, under certain circumstances, *keyword profiling* suffers from poor performance and the **ExperimentBuilder** aims to mitigate this by enabling manual redefinition. A particular limitation of the relationship estimation occurs if an important inter-component relationship exists but it is both unknown to the user and remains unidentified by the **ExperimentBuilder** due to limitations in the *keyword profile* comparison algorithm. In such a case, the relationship will not be identified within the experimental environment.

During development, we have not forgotten the original purpose of the **ExperimentBuilder**, which is to provide change management services as an addition to an existing experimental environment. In order to adequately employ our change management tools, we need to deliver them in an efficient and accurate environment. In response to this, the **ExperimentBuilder** allows the user to represent their experiments in a way that mimics, as closely as possible, the real world experimental environment. Within the **ExperimentBuilder**, the user can define a *line of enquiry*, our representation of a particular avenue of investigation. *Lines of enquiry* can be nested and may or may not contain any number of experiments. In addition to being encapsulated within a *line of enquiry*, an *experiment* must contain an experimental hypothesis to provide further identification. The *Line of Enquiry* page of the **ExperimentBuilder** provides the user with a complete overview of the entire *in silico* experimental environment. In one view, the user can navigate between different *lines of enquiry*, browsing experiments and their attached experimental components. Figure 5.12 illustrates the presentation of the experimental environment.

7.2.1.2 A Constant State of Change

In reality, many experimental environments will undergo a near constant state of change. To effectively model an experimental environment, the **ExperimentBuilder** is required to handle both the frequency and the nature of the changes that can occur but how well does the **ExperimentBuilder** handle the frequency and heterogeneity of change? Firstly, the **ExperimentBuilder** can not automatically detect a change that occurs in the real world. We have mentioned before that there are no logical or computational links between the *in silico* experimental environment of the **ExperimentBuilder** and the real world environment. The contents of the **ExperimentBuilder** are contained and controlled directly and only by the user. This is the current state of the **ExperimentBuilder**, maintained by the user. In the future, we can look at directly incorporating data or services from online *Web Services*, at which point, the **ExperimentBuilder** may be able to dynamically react to changes in real time.

The frequency of change, therefore, is monitored directly by the user and depends upon the frequency of user interaction with the **ExperimentBuilder**. This method can lead to inaccuracies in the **ExperimentBuilder** due to mistakes or omissions on the part of the user and we have tried to mitigate this to an extent by providing pop-up reminders for the user, prompting them to check that the experimental environment is up-to-date. Frequency of environmental change may occur at a brisk pace and, due to the required user interaction, the **ExperimentBuilder** may not remain consistently up-to-date. The nature of the environmental change and the characteristics of the likely impact is handled in the **ExperimentBuilder** by directly modelling the change itself. As we described in the previous chapter, changes are represented using *delta keyword profiles*, similar to the internal representation of the experimental components. The techniques used within the **ExperimentBuilder** are described in section 6.4. Using *keyword profiles*, we encapsulate the characteristics of the change directly in a format that can be compared against the remainder of the experimental environment. Disregarding how or why a change has occurred, we concentrate on the properties the change, abstracted to its *delta keyword profile*.

The modelling of change within the **ExperimentBuilder**, in terms of detecting change and implementing the versioning of affected components, is handled differently to the propagation of those changes. Within the experimental environment, for investigative purposes, components are versioned exactly so that each version of a component is completely retrievable. Non-datasoure components versions are recorded using a traditional, open-source **CVS** method[3]. Datasource components are handled differently, using our own **DBVS** methodology, described in section 5.5.2. When a change occurs, the **ExperimentBuilder** generates a *keyword profile* or, more accurately, a *delta-keyword profile* representing the change. The *delta keyword profile* is used to propagate the change to the identified dependant components, calculating the cost of the estimated impacts and presenting it to the user but using keywords to represent components creates problems. Firstly, some keywords are naturally more important than others. We make allowances for this in the **ExperimentBuilder** by assigning different weighting to keywords, depending on their semantic significance in the component. This alleviates rather than solves the problem. We have chosen to restrict user access in how component *keyword profiles* can be altered. Given the existing user-required overhead of defining inter-component relationships, the need for fine-grained manipulation of the individual *keyword profiles* is superfluous. Users can, however, exclude keywords but cannot add

[3]We have used the Google diff-match-patch algorithm available at http://code.google.com/p/google-diff-match-patch/

or alter existing ones. If we do not exclude keywords, we find that the *keyword profile* becomes dominated with keywords that have no semantic meaning but occur very frequently[4].

7.2.1.3 The Real Cost of Updating

A significant part of the research question is concerned with giving the user sufficient information in order to make an informed decision whether or not to update the experimental environment based on an estimated cost provided by the **ExperimentBuilder**. If we are to provide a real world estimation, we need to provide the estimation in real world terms. The **ExperimentBuilder** assigns a *cost profile* for each component, which allows the user to specify the levels of effort required to update the various aspects of the experimental environment. In building the *cost profile*, we encapsulate the effort required to update in fiscal and temporal terms. In order to do this, the **ExperimentBuilder** requires specific input from the user for a description of the work process. The *cost profile* is constructed with a selection of questions to the user regarding the length of time it takes to update the component and the costs involved in doing so. Using the answers to the questions, the **ExperimentBuilder** determines the minimum and maximum levels of effort required for any given update.

If we assume that the effort required to update a component scales according to the degree of change exhibited by it, we can determine the degree of effort required to deal with any given size of change. This determines the real-world impact of an observed change but the assumption can lead to inaccuracies in the calculation of effort as updates do not often scale perfectly. The **ExperimentBuilder**, therefore, allows the user to determine whether the update effort should scale or not. Many components, specifically off-the-shelf or third party components, will have a static update effort; the component will either be updated or not. Other components such as bespoke components or datasources, may better suit a sliding scale of update effort calculation to represent a component under in-house control.

We should remember that the *cost profile* is a subjective method for determining cost[5]. The *cost profile* will, most likely, reflect the primary user of the **ExperimentBuilder**. An experimental scientist working on their own environment might, for example, assign a cost of zero for their own effort in order to simplify the environment[6]. A more complex

[4]We refer to keywords such as "Yes", "No", "NA" etc.

[5]This is providing we have not manufactured a standardised way to enter costs for all types of user.

[6]The purpose of the scientist is to work on their experimental environment, therefore there is no special attention required for an update. They may, therefore, prefer to concentrate on temporal effort only when deciding whether to update or not.

experimental environment may require many different people for a given update. In such a scenario, it may be more suitable to apply a fiscal cost to the effort.

The **ExperimentBuilder** presents the effort to update in terms of both the time required and the estimated fiscal cost of the update. Based on the estimated cost, the user then decides whether to update or not. The cost can incorporate both user effort and computational effort and these are specified individually in the *cost profile*. We assume that the environment already contains the necessary computing equipment, as the experiment has already been carried out at least once, but we allow the user to define the cost and required time for any additional required consumables. *Machine* cost can be a considerable factor when updating as some experiments can take many hours for a single iteration. We also accommodate the possibility that the update of one component may or may not be run concurrently with the update of another and this property is set by the user in the *cost profile* for each component. It is important to record the concurrency of components in order to accurately estimate the cost of updating as the **ExperimentBuilder** will estimate the time for a total experimental update using the concurrency of the components therein.

7.2.2 Limitations of the ExperimentBuilder

In the preceeding section, we have described some of the benefits and limitations of the **ExperimentBuilder** in order to present a balanced appraisal of our solution. Within this section, we discuss the principal limitations of the **ExperimentBuilder** in more detail. If there are processes in place to mitigate these limitations in the **ExperimentBuilder**, we will present them here as well. We aim to present these limitations in a way that prompts further work and improvement and we have described the limitations with this in mind.

7.2.2.1 Design Limitations

The origins of the **ExperimentBuilder** are founded from an extensive analysis of an existing environment of biological experimentation. From that analysis, we have concluded the necessity to model four distinct groups of experimental components[7]; datasource, wrappers, methods and tools. These are described in section 5.4 but how well does this small group of descriptors encapsulate an experiment of unknown nature?

During the analysis, we established the existence of two types of experimental component; those containing data and logic, and those containing only data. We mimic this

[7]We exclude the *Experiment* and *LineOfEnquiry* components from this list and concentrate on the components that make up each *Experiment*

difference within the **ExperimentBuilder**. The datasources are considered the primary components for any given *Experiment* and the change detection and impact tracking methodologies have been, arguably, optimised for use on datasource components. A datasource can not contain any computing logic within the **ExperimentBuilder**. We define a component containing computational logic if the component either generates results from a datasource or transforms the datasource from one state to another. Components containing computational logic are split into three further components in line with our analysis of the case study. The *Wrapper* is a tool that is applied solely to datasources with the objective of preparing the data for use in the *Experiment*. *Methods* are defined as a transformation of one or many datasource(s) from the initial state, or the initial state after any required *wrappers* have been applied, to the next state[8], representing the *result* of the *Experiment*.

The *datasource*, *wrapper* and *method* components are clearly defined within the experimental environment provided by the **ExperimentBuilder**. In addition, we provide the *tool*, which is used to represent any additional piece of computational logic present within the experiment. We have deliberately weakened the structure of the experiment by allowing the integration of any type or number of *tools* into the environment. This is to ensure that experiments that exist in an unusual or non-standard form will still be adequately modelled in the **ExperimentBuilder**. There is a danger that, by defining *tools*, the user could introduce multiple *methods*[9] or could apply *wrappers* without specifying the attached datasource. In other words, they could circumvent the integrity of the experimental structure, using *tools* and there is currently no implementation preventing users from doing this.

By defining the types of experimental components allowed in the **ExperimentBuilder**, we have enforced an experimental structure that, while more intuitive during investigation of the environment, in some cases may restrict the experimental duplication upon which our *in silico* environment relies. To mitigate this problem, we provide the *tool* component so users can specify any number of additional logic components in order to more accurately represent their experimental environment. Misuse of the *Tool* component, however, can lead to over-population of the *experiment* and a break-down of the integrity of the experimental structure. Ultimately, the **ExperimentBuilder** requires a prudent use of experimental components together with a sensible approach on behalf of the user in order to successfully represent the experimental environment and get the most out of the **ExperimentBuilder**.

[8]Note that only one *Method* can be used for any single experiment. If more *methods* are required, we must break the *Experiment* into multiple single-method experiments.

[9]This is strictly against the experimental policy of the **ExperimentBuilder**.

It is imperative that the **ExperimentBuilder** retains the ability to both accurately model the real-world experimental environment and allows the environment to recreated as easily as possible. Much effort has been devoted to facilitating the duplication of the experimental environment due largely to the fact that the **ExperimentBuilder** relies entirely on the user input of experimental details in order to build the *in silico* environment. This brings us to the single, most significant and incontrovertible limitation, namely that the **ExperimentBuilder** contains only experimental metadata[10]. There are no links, logical or computational between the experimental environment in the **ExperimentBuilder** and the real world environment. The *in silico* environment of the **ExperimentBuilder** is created and maintained by the user and changes only in response to user input. This has several key advantages as well as some important limitations. By removing the real-world links from the **ExperimentBuilder**, we are free to define components in whatever way we wish, providing abstractions where appropriate rather than being tied to strict representations of the working component. There are systems that model components exactly, embodying the experimental environment and allow experimentation to be created and controlled within the application[11]. Within such systems, there are strict rules that govern the environmental components, often limiting users to a set of predefined components.

The **ExperimentBuilder**, in contrast, enforces no such restrictions on experimental components, other than that they can be represented textually. This allows the user the freedom to make design decisions that are based on the experimental environment rather than the change management application which should, in our opinion, act as a support framework to an experimental environment rather than an embodiment of the experiments themselves. In most cases, we expect the **ExperimentBuilder** to be integrated with an existing experimental environment, as per our case study in chapter 3. In such situations, it would be unreasonable to expect a user to rewrite or, more inconveniently, recompute the experimental environment to fit our new framework. The **ExperimentBuilder** has been designed to fit around existing experimental environment. The degree of user interaction with the **ExperimentBuilder** depends on the capacity of a component for change as well as the complexity and the tendency of the changes. An environment that changes often will require more user interaction in order to remain up to date. There are ways in which we could link some aspects of the **ExperimentBuilder** to the real world environment without compromising the freedom to create bespoke experimental environments[12].

[10]The **ExperimentBuilder** does in fact contain experimental data in the form of datasources but there is no computational link to the real experimental environment.

[11]We refer, specifically, to workflow re-enactment systems and these are described in section 2.5.2.

[12]Future work can be found in section 8.4.

Pivotal to the processes of change detection and impact propagation are the use of *keyword profiles.* Primarily, we use *keyword profiles* as an abstraction of a single version of a component. We can then use the *keyword profile* rather than the component versions to identify inter-component relationships. We need these relationships in order to accurately determine impact propagation. We also use a variant of the *keyword profile*, the *delta keyword profile* as an abstraction of the changes between one component version to the next. But how accurately are component versions and their changes represented by *keyword profiles*? The keyword manager accumulates identified keywords[13] and assembles them in order of semantic importance[14]. In section 7.2.3, we describe the performance of *keyword profiles* against that of a brute-force comparison technique. Ultimately, the performance of the *keyword profile* depends, in most cases, on the clarity of definition exhibited by the component together with the level of user diligence in curating the *keyword profiles*[15].

7.2.2.2 Implementation Limitations

The **ExperimentBuilder** compares the *keyword profiles* of components in order to identify a relationship. The strength of this relationship is determined by the levels of similarity found between the two *keyword profiles.* But how accurate is this determination? The *keyword profile* for one component version may be very accurate as may the other but this does not mean that we will necessarily capture an accurate relationship between the two. The reason being that a keyword in one *keyword profile* may not necessarily carry the same semantic meaning as a keyword in the other. Regardless of this, the profile manager will still register a degree of similarity. *Keyword mismatch* leads to phantom similarities and an overestimation of the inter-component relationship. There are many situations where keyword identification is not ideal for representing a component version. Semantically important keywords may be under-represented within the *keyword profile* which will lead to inaccurate relationship identification for that component version. The **ExperimentBuilder** allows the shifting of semantic weight throughout the *keyword profile* in order to mitigate the chance of under-representation.

In response to the inaccuracies that can occur during inter-component relationship identification, the **ExperimentBuilder** allows the user to tweak inter-component relationships after the automatic estimation process. Users can then redefine relationships

[13]A keyword is a non-numerical value with a length of at least three.

[14]Semantic importance is calculated as a result of topological importance and frequency. See section 6.2 for more details.

[15]The user can exclude undesirable or irrelevant keywords from the *keyword profile* in order to improve its efficacy. Note that the user can not make any changes to a *delta keyword profile.*

according to what they know about their own experimental environment. This is necessary for identifying relationships that cannot be determined using the built-in keyword analysis methods.

During our analysis of the existing experimental environment, we dedicated a significant effort to identifying the sources of possible change, described in section 3.3.1. This portion of the analysis contributed, in part, to the identification of the experimental components present in the system but also described the types of changes that were occurring. Components can change in many different ways and for many reasons. Representing the heterogeneity of the changes presents a significant challenge. The **ExperimentBuilder** represents a change as a *delta keyword profile*, which specifies the addition or removal of keywords from one component version to the next. As such, changes are represented by the impact on the effected component, rather than the cause or nature of the change.

Impact propagation from an identified change relies on two pre-existing estimations made by the **ExperimentBuilder**; the change, represented by a *delta keyword profile*, and the network of dependant related components from the origin of the change. When a new component version is added, the **ExperimentBuilder** conducts a keyword analysis, measuring the resultant *keyword profile* against the existing profiles, in order to establish any inter-component relationships and insert the new component version into the correct part of the impact propagation network. The success of impact propagation depends on the success and accuracy of its component parts. The same is true when considering the estimation of the update cost, supplied by the **ExperimentBuilder**, which can only be as accurate as the preceeding processes. The **ExperimentBuilder** uses *cost profiles* to describe the costs and efforts required to rebuild a particular component. This description is made, albeit admittedly somewhat clumsily, in terms of fiscal and temporal cost in order to express the real-world effort in updating. By expressing effort in these terms, the internal workings of the estimation can be hidden. The **ExperimentBuilder** allows the user to inspect all aspects of the estimation as well as the raw value versions of the presented estimations.

When a component requires an update, the computed cost of every dependant component is weighed and presented to the user so an informed decision can be made. The cost calculation is iterative so the cost to update a dependant component contains the total cost of its dependant components as well. When the experimental environment is complex with many dependant components, the cost of updating a single component, no matter how trivial, can be significant.

7.2.2.3 Technical Limitations

Our research question drives the application of a change management framework for an experimental environment. Before integrating change management, it was necessary to develop a framework capable of modelling an existing experimental environment, requiring an accurate model for the experimental environment together with an adequate user interface to drive the prototype, resulting in a substantial piece of work. Considerable effort was directed at building the experimental framework of the **ExperimentBuilder**, much of it requiring construction before work on the change management framework could begin.

As such, some elements of the **ExperimentBuilder**, in particularly the user interface, feel undernourished as a result of not being *core* material. The model persistence methodologies are a good example of this. It was clear from the start that we would require a data management system to handle the experimental data and components including the management of environmental persistence. The **ExperimentBuilder** environment will likely contain multi-versioned components and we, therefore, require a version-enabled data management system. After some initial investigation, we decided to devise a bespoke data management solution to handle the multi-versioned experimental data. Persisting the experimental data, with all versions intact and retrievable, presented a significant challenge.

7.2.3 Performance of the Keyword Profile Approach

Much of our research and framework for managing experimental change depends on our use of *keyword profiles*. We use *keyword profiles* to represent experimental component versions in order to identify inter-component relationships and establish the impact propagation network. We use *delta keyword profiles*, a derivative of the component version *keyword profile*, to represent a change from one component version to the next. Due to their inherent similarity, we can then compare dependant components against the change to determine the impact. The impact is simulated through our impact propagation network and, together with the registered *cost profiles* for each dependant component version, we estimate the levels of required effort to update due to the change.

We use *keyword profiles* as an abstraction of a component version but what impact does this abstraction have on performance? In other words, how good a representation of a component version is the *keyword profile*? We have already described some of the limitations of *keyword profiles*, as well as some of the benefits, in the previous sections. But we have, so far, made no mention of the performance of the *keyword profile* when compared against the components themselves, without an abstracted representation[16]. In other words, to what degree is a *keyword profile* an accurate representation of a component version?

In order to measure the performance of a *keyword profile*, and by extension the *delta keyword profile*, we have constructed an experimental dataset, upon which we can enact some determined change. Based on this change, we use the **ExperimentBuilder** to determine the degree of change from one version to the next using the integrated *keyword profile* system. We have discussed the limitations of the *keyword profile* for detecting change in value-based keywords. Indeed, we have decided to remove values altogether from the profiling mechanism due to the difficulty in assessing their semantic value. Because of this design choice, the **ExperimentBuilder** will perform poorly on value-intensive components, preferring components with few or no value data items. We have chosen two data components, upon which to base our performance metric. Ideally, as for the **HDM** performance, we would select a component populated with random data. In this case, however, a random data component would nullify the functionality of the *keyword profile*, essentially generating an evenly distributed profile containing no semantic meaning. Although it weakens the performance analysis, we must provide a component with semantic meaning in order to fully test the *keyword profiling* system.

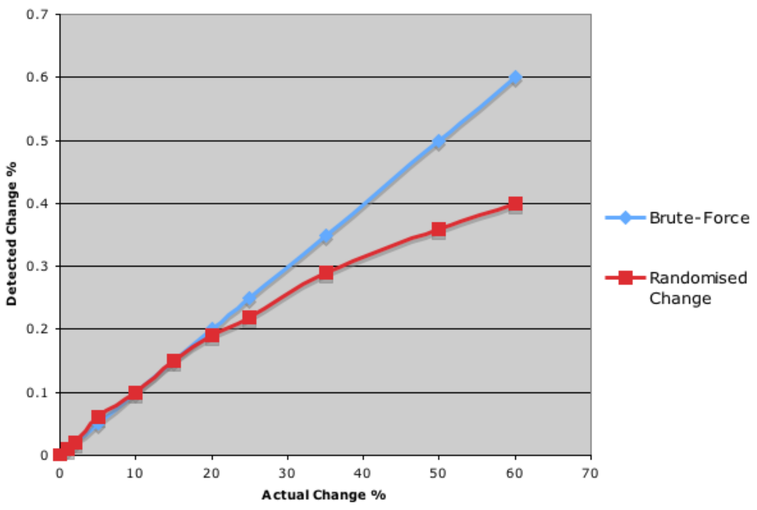
We are using the *demographics* datasource, described in section 3.2.1 with varying levels[17] of randomised change appearing within the dataset. We will use the **ExperimentBuilder** to detect the levels of similarity between the original and simulated datasets, the same methodology that looks for component version change, using *keyword profiles* to abstract the component versions. We will also employ the brute-force dataset comparison technique[18], available in the **ExperimentBuilder** to provide a point of comparison for the *keyword profile* approach.

In order to test the performance of the integrated *keyword profiling* system, we are comparing the accuracy in detecting change at varying levels. We compare these results against those of a brute-force approach, which literally tests each data item measuring change. The difficulty arises, not in affecting the change, but deciding how to simulate the change. The dataset can be changed by varying degrees and these changes are affected randomly throughout. The difficulty is determining to what the randomly selected data items should be changed in order to adequately simulate a real world change. If we change the data item to a random value, we are effectively removing a

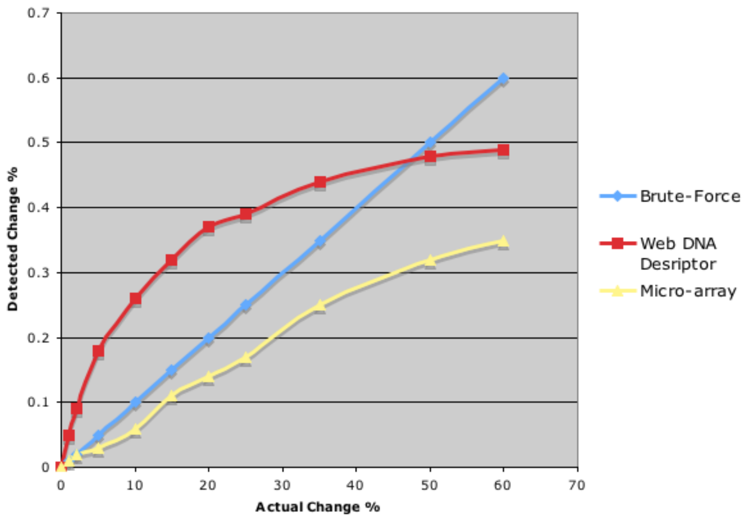
[16]We have already made the case for using a component abstraction, such as a *keyword profile*, primarily for the enabling of heterogeneous component comparison.

[17]We apply levels of 1, 2, 5, 10, 15, 20, 25, 50 and 60 percent of change to simulate the new version.

[18]The brute-force comparison technique compares every single data item in both versions, returning a relative difference value to represent the degree of change.



(a) Randomly generated change



(b) Randomly spliced data change.

FIGURE 7.2: Performance results for the **ExperimentBuilder**.

portion of the sample dataset and, due to the normalisation of the *keyword profile*, the change will not be fully represented.

In order to successfully affect a random data change comparison, we first change the selected data items to a single variable[19] and the results of this comparison can be seen in figure 7.2(a). But a dataset will not always change to the same value, which may account for the observed accuracy fall-off at higher levels of randomised change. So, we can not change the selected data items to random values and it is not ideal to use a static value either, although this does yield much more feasible results.

[19]In our performance comparison, we randomly select data items and change them to the value "DataChange".

Ideally, we need a legitimate series of dataset versions that have undergone normal, real-world change. Unfortunately, it is very difficult to get hold of this kind of information so we need to simulate the change. The random and static simulations mentioned above do have merit, although the nature of the simulated change is unrealistic. The problem is the data which we are introducing to the dataset in order to build the subsequent version. Under real-world conditions, the introduced change will contain some semantic value and will exhibit cohesive properties with the existing data, but how can this be simulated? Figure 7.2(b) illustrates the performance of the **ExperimentBuilder** on datasets that have been spliced, to varying degrees, with randomly selected data items from other experimental datasets. We have selected two scientific, but unrelated, datasets with which to splice the demographics data; a **DNA** descriptor file and a portion of microarray data. The results can be seen in figure 7.2(b).

We can see from the illustrated results that the **ExperimentBuilder** performs to some degree, correctly estimating each level of change in respect to others. There are, however, a number of limitations of the file-splicing approach. Firstly, although we have achieved semantic cohesion within the changed data items, there is no link to the existing data[20]. Secondly, we must acknowledge that, even though the demographics dataset and the splicing datasets are unrelated, using *keyword comparison* does report a small degree of similarity, at around 5% for each of the splicing files. This affects the accuracy for change detection by the **ExperimentBuilder** as around 5% of the data items being substituted will report a false positive against the *keyword profile* for the original dataset.

The performance analysis of the **ExperimentBuilder** somewhat demonstrates the efficacy of the *keyword profile*. We can see that *keyword profiles* do estimate change to some degree and, importantly, to react positively to percentage changes in the simulated changed datasets. This is demonstrated most ably by the random, static value insertion technique illustrate in figure 7.2(a). Ideally, we require real-world dataset versions but these are difficult to come by and it is even more difficult to accurately assess the degree of change between versions; that is the purpose of the **ExperimentBuilder**.

[20]As there would be in a real-world data change.

Chapter 8

Conclusion

The following chapter provides a conclusion to our work and this thesis. We begin by detailing a summary of our contribution in section 8.1, followed by a critical appraisal in section 8.2. Following this, we provide an appraisal of our contribution in comparison with the relative merits of other competing technologies in section 8.3.1. Finally, we look at possible future avenues that our contribution might take in section 8.4.

8.1 Summary of Work

During the previous chapters, we have often stressed the need for a new change management framework. This is primarily due to the lack of any existing framework that specifically targets change and the impact of change. Largely in unchartered territory with little previous research from which to derive system specifications, considerable efforts were required to build a precise set of requirements for the framework. The body of these efforts appear in the form of a comprehensive analysis of an existing experimental environment using a series of computational machine learning experiments. Further details of this case study can be found in chapter 3.

The focus of the case study is aimed directly at elements of the experimental environment that can change. Various experimental components can change but they do so in very different ways and these differences also impact the manner in which the change propagates through the system. The case study analysis of chapter 3 not only details the sample system but specifically all the identified points of possible change therein. The report goes on to describe how each of the changes can cause impact throughout the system. At that point, we were able to conclude the type of changes that could occur within a sample environment and, to a certain degree, observe how those changes affected the rest of the system.

Having identified a set of regularly occurring changes, it appeared that many occurred within the experimental datasets so it seemed sensible to start focussing on datasets and, in particular, the associated metadata. The Hashed Data Model (**HDM**) helps e-scientists identify, track and manage these types of change. The **HDM** is a tool that abstracts an experimental dataset together with any associated metadata to build a model that can be used to detect changes between versions of datasets. The **HDM** breaks down the dataset and then reconstructs it as an abstract model using hashes to represent specific portions of the dataset. If we break each dataset down in exactly the same way, we can then compare one to another but the main goal of this approach is to discover whether a dataset has changed from one moment to the next. The hashed value reflects a change no matter how small or insignificant. The characteristics of the **HDM** allow the user to potentially retain a unique version of the dataset for every single experiment. Of course the benefit of the hashed value, in that it can represent a change no matter how fine, is also its most significant restriction. Using hashes, we can tell if there has been a change but we know nothing about the size or strength of the change. We have incorporated some techniques into the **HDM** in order to rectify this limitation and, to some extent, these have been successful but there must be some abstraction of the dataset and this results in some loss of precision.

In order to successfully implement an experimental tracking framework, it is necessary to model the experimental environment within which the changes occur and the tracking is to take place. The **ExperimentBuilder** is the product of this requirement and embodies the experimental processes, allowing the user to track and observe both change and impact within their experiments.

There were three main stages in the construction of the **ExperimentBuilder**; construction of the experimental environment, tracking experimental impact and persisting the entire multi-version environment. The **ExperimentBuilder** provides users with the ability to define *Lines of Enquiries* (**LOE**s), which encapsulate the idea of a particular line of thought or hypothesis. **LOE**s can be nested, much like a file directory, to allow the user to build up their entire experimental environment within a single application. Users must also define the experimental components that they intend to use and the **ExperimentBuilder** provides tools to construct these. Once these components have been defined, experiments can be constructed, assigned to **LOE**s and populated with components.

The **ExperimentBuilder** estimates impact in the experimental environment by measuring relationships between components. It is easy to hypothesise how we could identify relationships between components of the same type, i.e. dataset vs. dataset. We can define algorithms to look for similar field names, field types, size, or metadata or simply a direct data comparison and then imply a relationship based on the degree of similarity. But what about components of differing types? How do you compare a dataset to a method? The **ExperimentBuilder** uses a heuristic algorithm that breaks down each component, regardless of type into a single, measurable *keyword profile* (*kwp*) that can be used as a representation of the component and compared against the *kwp* of any other component. The **ExperimentBuilder** automatically builds *kwp*s as components are added and then estimates the relationships of the new component to those within the rest of the environment. It also allows the user to fine-tune the impact relationships, should they wish.

At some point, a change will occur somewhere in the environment. When this happens, the **ExperimentBuilder** measures the strength and nature of the change and propagates it towards all components that have an identified dependant relationship but change cannot simply be passed on. A change may occur in one component that should not be passed on to a dependant component. Consider a scenario where a change in the source component[1] does not affect a dependant[2], regardless of the strength of the overall relationship. The **ExperimentBuilder** overcomes this problem by abstracting each change into a model similar to a component *kwp*. The change *kwp* can be compared against a component *kwp* to determine the effect of the change. We can then propagate the change based on its own characteristics rather than that of the source component. Using methods similar to these ones, the **ExperimentBuilder** can accurately propagate impact through multiple levels of related components. For exact descriptions of the tracking methods, please refer to section 6.4.

The final stage of the construction of the **ExperimentBuilder** is the persistence of the environment requiring a relatively complex solution and is far from trivial. We can easily employ strategies for statically persisting datasets, methods and tools but when we consider that these components are changing and we need to persist all available versions, environment persistence becomes complex. We require a solution that can record every version without becoming inflated and unmanageably large. We have developed several strategies for persisting a multiple version environment and these are described in more depth in section 5.6.

8.2 Critical Appraisal

The purpose of the research was to address the problems of change within an experimental environment. The practical outcome of the research is an application that

[1]We refer to source components as those where the change or impact originates. It may not necessarily be the original source of the change.

[2]A component that is defined as receiving impact from a source component is defined as a dependant.

encapsulates an experimental change management framework. Throughout the introductory chapters, we presented a concise history of genomics, focussing primarily on the **Human Genome Project (HGP)** as an example of a large-scale data-intensive project. We discussed the problems inherent in such projects and described some methods that have been conceived to deal with them. Inevitably, some problems receive more attention than others and we highlighted some areas that received relatively little. One significant area is the evolution of environmental data and tools over time. These can often change in an unpredictable and unannounced manner causing significant problems for dependant projects. The motivation of the work (section 1.1) builds the research question focussing on the **HGP** as an example. We could have described numerous other projects with similar requirements that prompted similar research questions and many of these are also described in chapter 2. Rather than discuss multiple applicable projects, we focus on the **HGP** in order to simplify the background to our research question.

Due to a lack of existing research on the subject of experimental change management and impact propagation, it was necessary to find a clear and appropriate case study that encapsulates frequent and unannounced change and embodies the associated problems. The case study uses data from a private collaborative database consisting of clinical and proteomic data from 350 patients. The data is then mined using various machine learning techniques in order to derive patterns and relationships within the data. Additionally, proteins can be cross-referenced from an external public database. The case study contains many features that make it suitable for our research. Firstly, it employs both private, collaborative and public datasources. Different datasources behave differently with respect to change depending on their characteristics and it was important to find a case study that used, at least, some varying datasources. For any given research, it is necessary to have sufficient examples in order to demonstrate the validity of the work. Due to the lack of pre-existing research, there was an inherent lack of identified case studies appropriate for investigating change management. We have built a comprehensive analysis of an appropriate system, presented in chapter 3.

Throughout the course of the case study analysis, we focussed on measuring the states of change that occurred within the system. We conducted an overview of the entire system, providing a detailed analysis of all parts of the experimental process along with all aspects of data integration. We followed up with a document detailing all the parts of the system that could initiate a change. Every conceivable change in the experimental environment was recorded and was then followed up by a detailed analysis of how each change could propagate through the system and the resulting impacts. The result of the case study provides a complete change profile for an existing experimental environment specifying every possible change with the identified impact. We use this analysis to build a system specification, taking into account all the identified requirements.

The Hashed Data Model (**HDM**) prototype has be shown to work within an existing biological domain with real biological data, including the data from the case study. However, the **HDM** is only an initial step towards a comprehensive framework for change management. In order to provide real usability for biologists, more must be added to the **HDM**. The limitations of the **HDM** are uncovered as we investigate the inner workings. The **HDM** builds an abstraction of the dataset and therefore, we can never recover the dataset in its original form. This is a problem if the changes are to be investigated on a fine scale. Consider the **HDM** with a granularity factor of one[3]. Even at this level of granularity, we can not infer what the data values were and what they are now. For many cases, this is an acceptable level of granularity but for explicit data investigation, we propose the addition of a versioned database containing the data sources used for model creation and *in silico* experimentation.

The analysis of the limitations of the **HDM** highlights several requirements for a next step. Among other requirements, it is clear that we require a fully persistent experimental environment that encapsulates all aspects of any experiment. One of the major achievements of the **ExperimentBuilder** is the ability to accurately and fluidly represent a portfolio of experiments and faithfully record the interactions between. The **ExperimentBuilder** allows the user to create experimental components in order to build experiments. As these components are built, they are analysed by the **ExperimentBuilder** and compared against any pre-existing environmental components to identify possible inter-component relationships. We can then estimate and define relationships not only within experiments but between them throughout the entire environment.

One of the principal characteristics of the **ExperimentBuilder** is the accumulation of experimental metadata to form a representation of the environment. This approach has several advantages, discussed in chapters 5 and 6. There are, however, significant limitations to this method. The most notable, perhaps, is the lack of workflow re-enactment. We store the component metadata rather than the components themselves and, because of this, there is no physical link between our experimental environment and the external experimentation. We can not, therefore, initiate or re-run experimentation from the **ExperimentBuilder**. Moreover, the **ExperimentBuilder** is not only reliant on user participation in order to function effectively but requires strict data authenticity to allow valid change management. There are currently no formal mechanisms for enforcing correct user input within the **ExperimentBuilder**.

[3]A granularity of one represents a **HDM** where every single data item is modelled with its own individual hashed value.

We have constructed an original change tracking algorithm, implemented within the **ExperimentBuilder**, allowing the estimation of environmental impact and, more importantly, enabling the estimation of multiple levels of propagated impact. Clearly, we would like to know the effects of changes and how they ripple through the system. The **ExperimentBuilder** enables us to do that. There are limitations to the change management framework. All aspects of the change management use *keyword profiles* and herein lies some of the main limitations. The use of keywords can miss some key points of the data; values are disregarded under the strict use of keyword profiling and this can ignore some significant component differences. The same keyword can carry a different semantic meaning, depending on its context and placement and, currently, the *keyword profiling* techniques do not account for non-related identical keywords.

A significant achievement for the **ExperimentBuilder** is the successful persistence of the experimental environment. Given the complex nature of the environment together with the requirement for multi-versioned components, this proved to be far from trivial. We used a **MySQL** database to house the environmental data. A **MySQL** implementation has several benefits, most notably the ability to house data either locally or remotely with the management of multiple remote connections. As **Experiment-Builder** connections are made to the database in the form of either a *persist* or a *restore*, we can monitor multiple users, avoiding such problems as *deadlock*. There is currently no specific functionality included in the **ExperimentBuilder** to manage concurrency.

8.3 The Contribution Revisited

This section describes our main research contributions to be considered. In chapter 1, we have introduced the important problem of biological data and experiment versioning which, hitherto has been largely ignored. We have presented a clear definition of related problems that lead ultimately to our research question.

In chapter 2, we have presented a comprehensive and thorough examination of all the relevant background, including an extensive review of existing related technologies together with an appraisal of each as it relates to our research. At that point, we can conclude the need for a new change management framework. The analysis of an existing experimental environment in chapter 3 reinforces our case for a change management framework. This case study is an example of the requirement for change management and as validation for our change management framework.

There are two main stages of implementation within our research. The **HDM** is a framework that provides a light-weight abstraction of an experimental dataset enabling change tracking at varying granularities as well as semantic differentiation of data. The **HDM** has limitations, that have already been discussed, and the **ExperimentBuilder** represents a progression to a complete representation of an entire experimental environment, encapsulating all required aspects of experimentation into a single, manageable framework, allowing investigation and integration of multiple experimental components. The **ExperimentBuilder** provides the experimental framework for our change management methodologies to track changes in our environment and determine the real-world cost of updating any affected components.

The outcome of our change management framework is a cost of updating, presented to the user in terms of time and effort based on the estimation of impact resulting from a single or series of changes.

8.3.1 An Appraisal of the Contribution Relative to Other Products

Over the past chapters, we have made mention of other technologies which may, in part, compete with our offering, encapsulated by the **ExperimentBuilder**. At the conception of the research, a review of the relevant literature was carried out with the conclusion that there existed no single tool or framework that dealt specifically with the evolution of data and/or processes that combine to form experiments. At the time of writing, there still remains no single framework or tool for achieving that goal, save for our implementation. There are, however, several technologies with which we can compare various parts of implementation and approach to discuss the relative merits and demerits of each.

It would be inappropriate to talk about versioning without mentioning traditional versioning methodologies, in particular **CVS** and other related technologies. For more detailed information on these technologies, please refer to section 2.2. Traditional versioning methods have been very popular over the years and are increasingly being integrated into modern operating systems[4]. Versioning is also being employed not just by developers or computer *super-users* but increasingly by a mainstream audience with the release of popular tools such as **Dropbox**[5], a light-weight snapshotting tool for multi-user, any-location access.

The limitation of traditional versioning systems, however, is that the target of the versioning is invariably a text file[6]. Traditional versioning systems are aimed at versioning text or text-based resources and, because of this, makes them inappropriate for dealing

[4]We refer to *Time Machine* available as standard in Apple's Mac OS X.
[5]More information can be found at http://www.dropbox.com
[6]This excludes snapshotting tools that generally do not consider the internal elements of a file anyway.

with the large-scale management of evolving experimentation. We require the representation of many versions with multiple inter-component dependencies and traditional methods do not possess the expressivity to handle this. This is not surprising; the problem domain is outside that which traditional methods were designed to work. We do, however, use traditional methods where appropriate. When retaining multiple versions of components (e.g. methods), we use **CVS** to generate the backwards *diff* from the current version. This saves from having to retain the complete textual copy of each version. We use other mechanisms for retaining the inter-component dependencies.

Shui et al. [83] present an XML-based version management system for tracking complex biological experiments in order to describe changes in the XML-based description. This provides a useful format for representing biological experiments and is one of the first mentions of biological processes being as important as the data that is used to generate the results.

But we need to go much further than this. It is not sufficient to say that something has changed. The drawback with most tools concerned with change is that they have been designed to be *reactive*. By this, we mean that at some point during or after experimentation, presumably after a failed experiment[7], we can look to see if something has changed and then do something. This strategy has been adopted by most, if not all, data providers. The larger primary data providers, such as **Swiss-Prot** and **Ensembl** allow us access to the archives to compare versions of data but these are only really useful after an experiment has failed. There is little to promote inspecting previous versions before running the experiment in order to establish whether an experiment will pass or fail; it would take too long. But experiments can require a lot of time and/or effort to run. It would be preferable to know whether an experiment might be affected by a change without running it.

We aim to provide a system that is *proactive*, a revolution of the run-it-and-see methods, that establishes a framework to estimate how an experiment may be affected, given a change to one of its components. This is a very new idea and, as such, there is no specific methodology with which to make a direct comparison.

We can look for similarities with the **Taverna** e-science workbench, the main user interface for ^{my}Grid. **Taverna** is a large-scale project that provides language and software tools to enable the design and implementation of workflows. Both **Taverna** and our own **ExperimentBuilder** allow the design of workflows, the difference being that **ExperimentBuilder** cannot implement the workflow. This does, however, give the **ExperimentBuilder** a significant advantage in that it is not limited by the components that can be specified in a **Taverna**-compliant workflow. In the **ExperimentBuilder**,

[7]We refer to an experiment as failed if that experiment is not returning the expected results.

the user can design the workflow in any desired way that should conform only to their own experimental environment.

There is very little literature as to how change detection is handled within **Taverna** and, indeed the whole of ^{my}Grid. If the services or infrastructure within ^{my}Grid decay, so does the workflow. There are tools that can help identify the decay of the workflow, such as the aforementioned **Workflow Monitor**, but debugging failing workflows remains a crucial but neglected topic[41]. Of course, it is not a requirement of the **ExperimentBuilder** to implement a workflow just as the management of evolving components is not a principal requirement of **Taverna** so it is not possible to compare the two technologies directly. It is perhaps useful to think of the **ExperimentBuilder** as sitting aside **Taverna** with regards change management. Using the **ExperimentBuilder**, we implement a workflow not for the purposes of enactment but to identify dependencies within our experimental environment which may lead to impacts to a component as a result of a change somewhere in the environment[8].

8.4 Avenues for Future Work

There are numerous avenues for expanding and improving the **Experiment Builder**. We can improve the way the **ExperimentBuilder** interacts with the *in silico* experimentation and revisit the possibility of integrating parts of the experimental process directly into the **ExperimentBuilder** possibly allowing workflow enactment. Individual improvements aside, the goal of the **ExperimentBuilder** is to improve the experimental process through the addition of change management. We have shown the change management framework to successfully deal with the issues raised by our analysis of the case study. We have demonstrated that the **ExperimentBuilder** works but we have not fully analysed whether it is worth employing the **ExperimentBuilder**. In order to fully demonstrate the efficacy of the **ExperimentBuilder**, we need to integrate it into a large-scale experimental project in order to analyse whether the addition of the **ExperimentBuilder** is justified; i.e., is the level of required user interaction worth the benefits provided by the **ExperimentBuilder**? This is an important aspect to consider, especially as the success of the **ExperimentBuilder** depends on the usability of the system, particularly the willingness of the user to employ the system.

While there has been substantial progress in the areas of biological data management and related areas over recent years, there is still relatively limited work carried out on biological change management. The purpose of the the **ExperimentBuilder** to improve the

[8]For details on the precise operation of the **ExperimentBuilder**, please refer to chapters 5 and 6.

experimental process by enabling change management and there are many opportunities for improving existing experimental applications. There are several future directions for related work, including four specific high-level directions that seem promising; saving of workload due to accurate impact estimation; validation and/or certification for safety-critical experimentation; workbench and research validation; fraudulent research detection.

In chapter 1, we describe a scenario whereby a scientist, carrying out normal experimentation, might subsequently find that the source data for the experiments has changed. At this point, there are some difficult questions to consider, especially if the line of affected experimentation is current and/or important. The scientist needs to know, with some degree of certainty, how the data source has changed and, more importantly, how these changes may affect their experimental results. Our change management framework, encapsulated by the **ExperimentBuilder** helps such a scientist not only identify these changes as they occur, but convey an accurate estimation of the impact on their results. This allows us to know when we may need to update our results but more importantly, allows us to know when the data has changed insignificantly so as to not affect our results. Results obtained from complex experimentation may take many hours and computational cycles to reproduce, clearly undesirable if such an update is not required. We can, theoretically, save a great deal of time, therefore increasing productivity and simultaneously improving the reliability of results.

But we can provide much more with a change management framework. Consider an experimental system containing very important experiments. Note that any scientist might consider their experiments *very important* but for the moment, we are interesting in only those experiments yielding results of a critical nature; disease diagnosis and prognosis, drug delivery systems, or part of a clinical trial for an important drug. The success of such experimentation depends heavily on achieving valid results in a timely manner and can, therefore, be thought of as *safety-critical* experimentation. In this situation, a change management framework can be implemented as a validation process for *safety-critical* results by ensuring that the results have been generated by the most current state of experimental process.

Currently, results are generated through a specific experimental process and then submitted, presumably in an up-to-date state. Without a change management framework, however, the description of the state is meaningless. As we have already discussed, the data and tools that form experiments can change in a unannounced and unpredictable manner. Considering this state of constant change, results may be out-of-date at any point after submission. We can attach a date, or some other brief metadata, to the results which may give us an indication of the reliability of the data but a change management framework allows us to record so much more. Results can be presented together with metadata describing the experimental process, much like the implementation of the **ExperimentBuilder**, and this allows us to measure the validity of the results at any point in the future. We may need to update the results the very next day or they may be reliable for ten years, depending on the changeability of the experimental components. For *safety-critical* applications, we will know the exact moment that results become unreliable and, as importantly, we will know for how long they remain valid.

We can easily identify *safety-critical* applications that require a strict validation process but when we consider more mainstream research avenues, the same advantages of a change management framework can be applied. With a few conceptual tweaks, we can conceive a validation framework for an entire line of research. The **ExperimentBuilder** monitors experimental components in order to determine the impact on results. But the story doesn't end there. Some of the results go on to form the basis of other experiments and the experimental process continues. Some of the data is used to form direct analyses or graphs and these may appear in a publication or report of some kind[9]. If we can define graphs, reports and publications as components within a framework such as **ExperimentBuilder**, we can then define them as dependents of experiments and, ultimately, data sources. Using a change management framework in this way, whenever data is presented in any form, it comes complete with an associated validation framework. It could therefore become impossible to present data, either in a publication or simply informally, if the presentation was based on data that was out-of-date.

We could go further than simply validating the presentation of data. Unreliable data can occur for many reasons; some beyond the control of the scientist but some due to poor scientific procedure. A change management application providing a validation framework can help identify the state of data reliability. But sometimes, unreliable data can be purposely presented. More and more cases involving scientific fraud have come to light over the last two decades, largely due to an increase in biostatisticians who have begun to work more closely with physicians and scientists in many branches of medical research[74]. There are many ways to commit scientific misconduct and it is generally very difficult and costly to identify and investigate potential cases[93], but there is evidence to suggest research fraud to be more prevalent than originally thought[10].

Scientific fraud is a very serious issue and can, especially in the case of medical research fraud, endanger human life. In 1997, two cancer specialists, Friedhelm Hermann and

[9]It may sound obscure describing data in this way but this is precisely how data flows from its initial source to the point where it is presented as evidence to support theory.

[10]A review in the British Medical Journal (1999) found that half of the US biomedical researchers accused of scientific fraud and subjected to formal investigations in recent years were found guilty of misconduct

Marion Brach were found to have forged several publications and were subsequently dismissed from their respective posts. Over the next 18 months, it transpired that nearly 100 published papers over the previous ten years contained apparent data manipulations and a further 120 publications could not be validated and possibly contained wrong data. The data manipulation became clear in several images where old data had been either wrongly used or manufactured. It is impossible to measure the impact, ironically given our research, of such deception, especially given the longevity of the misconduct. How much research is based on these fraudulent publications and how can we now guarantee the quality of such work and how could a change management framework help?

If we were to set up the **ExperimentBuilder** to flag changes to source data for any published work, we could tell immediately when any published data were to become invalidated. This would make it impossible to publish work that was based on source data that was either out-of-date or had been manipulated throughout the course of experimentation[11]. Of course, employing the **ExperimentBuilder** to function in this way would be the choice of the individual scientist and we make no exception for those individuals who choose to avoid a genuine approach to research. At the very least, the **ExperimentBuilder** could eradicate accidental data unreliability and, if implemented properly, it could help remove the possibility of any out-of-date data appearing in published, peer-review form for any reason.

[11]The **ExperimentBuilder** would identify these manipulations as changes, albeit regardless of intent.

Bibliography

[1] N. L. Anderson and N. G. Anderson. Proteome and proteomics: new technologies, new concepts, and new words. *Electrophoresis*, 19(11):1853–61, 1998. Journal Article Review Germany.

[2] G. D. Bader, I. Donaldson, C. Wolting, and et al. BIND–The Biomolecular Interaction Network Database. *Nucleic Acids Res*, 29(1):242–5, 2001. Journal Article Research Support, Non-U.S. Gov't England.

[3] A. Bahl, B. Brunk, R. L. Coppel, and et al. PlasmoDB: the Plasmodium genome resource. an integrated database providing tools for accessing, analyzing and mapping expression and sequence data (both finished and unfinished). *Nucleic Acids Res*, 30(1):87–90, 2002. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. Research Support, U.S. Gov't, P.H.S. England.

[4] B. R. Barkstrom. Data product configuration management and versioning in large-scale production of satellite scientific data. *Lecture notes in computer science*, 2649: 118–133, 2003.

[5] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. GenBank. *Nucleic Acids Res*, 37(Database issue):D26–31, 2009. Journal Article England.

[6] N. Bhatnagar, B. A. Juliano, and R. S. Renner. Data annotation models and annotation query language. *World Academy of Science, Engineering and Technology*, 33, 2007.

[7] A. Birkland and G. Yona. BIOZON: a system for unification, management and analysis of heterogeneous biological data. *BMC Bioinformatics*, 7:70, 2006. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. England.

[8] W. P. Blackstock and M. P. Weir. Proteomics: quantitative and physical mapping of cellular proteins. *Trends Biotechnol*, 17(3):121–7, 1999. Journal Article Review England.

[9] J. A. Blake and M. A. Harris. The Gene Ontology (GO) project: structured vocabularies for molecular biology and their application to genome and expression analysis. *Curr Protoc Bioinformatics*, Chapter 7:Unit 7 2, 2008.

[10] B. Boeckmann, A. Bairoch, R. Apweiler, and et al.. The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003. *Nucleic Acids Res*, 31(1): 365–70, 2003. Journal Article England.

[11] H. Boutselakis, D. Dimitropoulos, J. Fillon, and et al. E-MSD: the European Bioinformatics Institute Macromolecular Structure Database. *Nucleic Acids Res*, 31(1):458–62, 2003. Journal Article Research Support, Non-U.S. Gov't England.

[12] A. Brazma, P. Hingamp, Quackenbush, and et al. Minimum information about a microarray experiment (MIAME)-toward standards for microarray data. *Nat Genet*, 29(4):365–71, 2001. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, P.H.S. United States.

[13] P. Buneman, A. Deutsch, and W. C. Tan. A deterministic model for semistructured data. *Proceedings of the Workshop On Query Processing for Semistructured Data and Non-standard Data Formats*, pages 14–19, 1999.

[14] P. Buneman, S. Khanna, and W. C. Tan. Data provenance: Some basic issues. *Fst Tcs 2000: Foundations of Software Technology and Theoretical Computer Science*, 1974:87–93, 2000.

[15] P. Buneman, S. Khanna, and W. C. Tan. Computing provenance and annotations for views. *Data provenance/derivation workshop*, 2002.

[16] E. Camon, M. Magrane, D. Barrell, and et al. The Gene Ontology Annotation (GOA) project: implementation of GO in SWISS-PROT, TrEMBL, and InterPro. *Genome Res*, 13(4):662–72, 2003.

[17] J. Cardoso and A. Sheth. Introduction to semantic web services and web process composition. *Semantic Web Services and Web Process Composition*, 2005.

[18] J. Cashmore. The value of cleansing data. *Mater Manag Health Care*, 14(10):52, 2005. Journal Article United States.

[19] K. H. Cheung, K. Y. Yip, A. Smith, R. Deknikker, A. Masiar, and M. Gerstein. YeastHub: a semantic web use case for integrating data in the life sciences domain. *Bioinformatics*, 21 Suppl 1:i85–96, 2005.

[20] S. Y. Chien, V. J. Tsotras, and C. Zaniolo. XML document versioning. *SIGMOD Rec.*, 30(3):46–53, 2001.

[21] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. DBNotes: a postit system for relational databases based on provenance. *SIGMOD 05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 992–944, 2005.

[22] G. Cochrane, K. Bates, R. Apweiler, and et al. Evidence standards in experimental and inferential INSDC third party annotation data. *Omics*, 10(2):105–13, 2006. Journal Article Research Support, N.I.H., Intramural Research Support, Non-U.S. Gov't United States a journal of integrative biology.

[23] P. Dadam, V. Lum, and H. D. Wener. Integration of time versions into a relational database system. *10th International Conference on Very Large Databases*, pages 509–522, 1984.

[24] S. B. Davidson, C. Overton, and P. Buneman. Challenges in integrating biological data sources. *J Comput Biol*, 2(4):557–72, 1995. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. United states a journal of computational molecular cell biology.

[25] S. Decker, S. Melnik, F. Van Harmelen, and et al. The semantic web: The roles of XML and RDF. *IEEE Internet Computing*, 4:63–74, 2000.

[26] D. di Bernardo, M. J. Thompson, T. S. Gardner, and et al. Chemogenomic profiling on a genome-wide scale using reverse-engineered gene networks. *Nat Biotechnol*, 23 (3):377–83, 2005. Evaluation Studies Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. Research Support, U.S. Gov't, P.H.S. United States.

[27] K. R. Dittrich, D. Tombros, and A. Geppert. Databases in software engineering: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*, pages 293–302, 2000.

[28] R. D. Dowell, R. M. Jokerst, A. Day, S. R. Eddy, and L. Stein. The distributed annotation system. *BMC Bioinformatics*, 2:7, 2001.

[29] M. Y. Eltabakh, M. Ouzzani, and G. A. Aref. BDBMS? a database management system for biological data, Jan 2007 2007.

[30] A. Finkelstein, J. Hetherington, L. Li, O. Margoninski, P. Saffrey, R. Seymour, and A. Warner. Computational challenges of systems biology. *Computer*, pages 26–33, 2004.

[31] M. D. Flouris. Clotho: Transparent data versioning at the block I/O level. *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*, 2004.

[32] I. Foster, J. Vckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, 2002.

[33] J. Frew and R. Bose. Earth system science workbench: A data management infrastructure for earth science products. *Proceedings of the 13th International Conference on Scientific and Statistical Database Management (SSDBM 01)*, pages 180–189, 2001.

[34] J. Frew and R. Bose. Lineage issues for scientific data and information. *Data provenance/derivation workshop*, 2002.

[35] J. Futrelle, S. S. Chen, and K. C. Chang. NBDL: A CIS framework for NSDL. *Joint Conference on Digital Libraries*, 0:124–125, 2001.

[36] T. S. Gardner, D. di Bernardo, D. Lorenz, and J. J. Collins. Inferring genetic networks and identifying compound mode of action via expression profiling. *Science*, 301(5629):102–5, 2003. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. United States.

[37] M. Gertz, K.-U. Sattler, F. Gorin, M. Hogarth, and J. Stone. Annotating scientific images: a concept-based approach. *Scientific and Statistical Database Management,*, 14th International Conference, 2002.

[38] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, 1988.

[39] D. Gilbert. Shopping in the genome market with EnsMart. *Brief Bioinform*, 4(3): 292–6, 2003.

[40] C. Goble. Position statement: Musings on provenance, workflow and (semantic web) annotations for bioinformatics. *Data provenance/derivation workshop*, 2002.

[41] C. Goble and D. De Roure. The impact of workflow tools on data-centric research. *Data Intensive Computing: The Fourth Paradigm of Scientific Discovery*, 2009.

[42] M. Golfarelli, J. Lechtenborger, S. Rizzi, and G. Vossen. Schema versioning in data warehouses. *Conceptual Modeling for Advanced Application Domains*, 3289: 415–428, 2004.

[43] J. Grabda. *Marine Fish Parasitology.* Weinheim Germany: VCH Verlagsgesellschaft., 1991.

[44] F. Grandi and F. Mandreoli. A formal model for temporal schema versioning in object-oriented databases. *A Timecenter Technical Report TR-68*, 2002.

[45] D. Grune, B. Berliner, and J. Polk. Concurrent versioning system.

[46] J. Hendler. Communication: Enhanced science and the semantic web. *Science 299*, pages 520–521, 2003.

[47] V. Henson and J. Garzik. BitKeeper for kernel developers. *Ottawa Linux Symposium, Ottawa, Ontario Canada*, 2002.

[48] H. G. Herbert, N. H. Gehani, W. H. Piel, J. T. L. Wang, and C. H. Wu. BIO-AJAX: an extensible framework for biological data cleaning. *SPECIAL ISSUE: Data engineering for life sciences*, 33(2):51–57, 2004.

[49] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, page 19, 1994.

[50] E. Hodis, J. Prilusky, E. Martz, I. Silman, J. Moult, and J. L. Sussman. Proteopedia - a scientific 'wiki' bridging the rift between three-dimensional structure and function of biomacromolecules. *Genome Biol*, 9(8):R121, 2008. Journal Article Research Support, Non-U.S. Gov't England.

[51] T. J. Hubbard, B. L. Aken, S. Ayling, and et al. Ensembl 2009. *Nucleic Acids Res*, 37(Database issue):D690–7, 2009.

[52] M. Hucka, A. Finney, H. M. Sauro, and et al. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–31, 2003. SBML Forum Evaluation Studies Journal Article Research Support, Non-U.S. Gov't England.

[53] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 239–249, 1999.

[54] M. Kanehisa, S. Goto, M. Hattori, K. F. Aoki-Kinoshita, M. Itoh, S. Kawashima, T. Katayama, M. Araki, and M. Hirakawa. From genomics to chemical genomics: new developments in KEGG. *Nucleic Acids Res*, 34(Database issue):D354–7, 2006. Journal Article Research Support, Non-U.S. Gov't England.

[55] M. Kanehisa, M. Araki, S. Goto, and et al. KEGG for linking genomes to life and the environment. *Nucleic Acids Res*, 36(Database issue):D480–4, 2008. Journal Article Research Support, Non-U.S. Gov't England.

[56] P. D. Karp, M. Riley, M. Saier, I. T. Paulsen, J. Collado-Vides, S. M. Paley, A. Pellegrini-Toole, C. Bonavides, and S. Gama-Castro. The EcoCyc database. *Nucleic Acids Res*, 30(1):56–8, 2002.

[57] A. Kasprzyk, D. Keefe, D. Smedley, D. London, W. Spooner, C. Melsopp, M. Hammond, P. Rocca-Serra, T. Cox, and E. Birney. EnsMart: a generic system for fast and flexible access to biological data. *Genome Res*, 14(1):160–9, 2004. Comparative Study Journal Article Research Support, Non-U.S. Gov't United States.

[58] T. S. Keshava Prasad, R. Goel, K. Kandasamy, and et al. Human protein reference database–2009 update. *Nucleic Acids Res*, 37(Database issue):D767–72, 2009. Journal Article England.

[59] S. Khaddaj, A. Adamu, and M. Morad. Object versioning and information management. *Information and Software Technology*, 46(7), 2004.

[60] M. Klein and D. Fensel. Ontology versioning on the semantic web. *Proceedings of the First International Semantic Web Working Symposium*, pages 75–91, 2001.

[61] J. L. Y. Koh, S. P. T. Krishnan, S. H. Seah, P. T. Tan, A. M. Khan, M. L. Lee, and V. Brusic. BioWare: A framework for bioinformatics data retrieval, annotation and publishing. search and discovery in bioinformatics. *ACM SIGIR Workshop on Search and Discovery in Bioinformatics (SIGIRBIO)*, 2004.

[62] D. Korn and E. Krell. The 3-D file system. *USENIX Summer 1989 Technical Conference Proceedings*, June 1989:147–156, 1989.

[63] J. Kovse and T. Harder. V-Grid - a versioning services framework for the grid. *Proc. 3rd Int. Workshop Web Datenbanken, Berliner XML Tage, Berlin*, pages 140–154, 2003.

[64] Z. Lacroix. Biological data integration: wrapping data and tools. *IEEE Trans Inf Technol Biomed*, 6(2):123–8, 2002. Journal Article United States a publication of the IEEE Engineering in Medicine and Biology Society.

[65] T. Miyagawa, N. Nishida, J. Ohashi, and et al. Appropriate data cleaning methods for genome-wide association study. *J Hum Genet*, 53(10):886–93, 2008. Journal Article Research Support, Non-U.S. Gov't Japan.

[66] J. P. Narain, E. Pontali, and S. Tripathy. Epidemiology and control strategies. *Symposium on HIV & TB. Indian Journal of Tuberculosis*, 2000.

[67] N. F. Noy, S. Kunnatur, M. Klein, and M. A. Musen. Tracking changes during ontology evolution. *The Semantic Web ISWC 2004*, 3298:259–273, 2004.

[68] P. I. Olason. Integrating protein annotation resources through the distributed annotation system. *Nucleic Acids Research*, 33:468–70, 2005.

[69] Z.M. Ozsoyoglu, G. Ozsoyoglu, and J. Nadeau. Genomic pathways database and biological data management. *Animal genetics*, 37 Suppl 1:41–7, 2006.

[70] C. Pancerella, J. Myers, T. C. Allison, and K. Amin. Metadata in the collaboratory for multiscale chemical science. *Proceedings of the Dublin Core Conference (DC-03)*, 2003.

[71] D. Pearson. Data requirements for the Grid. *Scoping Study Report - Draft Status*, 2002.

[72] Z. N. J. Peterson and R C. Burns. Ext3cow: The design, implementation, and analysis of metadata for a time-shifting file system. *Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University*, 2003.

[73] S. Ram and G. Shankaranarayanan. Research issues in database schema evolution: the road not taken., 2003.

[74] J. Ranstam, M. Buyse, S. L. George, and et al. Fraud in medical research: an international survey of biostatisticians. ISCB subcommittee on fraud. *Control Clin Trials*, 21(5):415–27, 2000. Journal Article United states.

[75] Peter H. Raven. *A Biological Survey For The Nation.* National Academy Press, 1993.

[76] A. L. Rector, J. E. Rogers, P. E. Zanstra, and E. Van Der Haring. Open-GALEN: open source medical terminology and tools. *Proc AMIA Symp*, 2003.

[77] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[78] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. *SIGOPS Oper. Syst. Rev.*, 33(5):110–123, 1999.

[79] N. Schiller. Internet reviews: National biological information infrastructure. *College & Research Libraries News*, 68, 2007.

[80] C. Schonbach, P. Kowalski-Saunders, and V. Brusic. Data warehousing in molecular biology. *Brief Bioinform*, 1(2):190–8, 2000. Journal Article Research Support, Non-U.S. Gov't England.

[81] S. M. Searle, J. Gilbert, V. Iyer, and M. Clamp. The otter annotation system. *Genome Res*, 14(5):963–70, 2004. Journal Article Research Support, Non-U.S. Gov't United States.

[82] S. P. Shah, Y. Huang, T. Xu, M. M. Yuen, J. Ling, and B. F. Ouellette. Atlas - a data warehouse for integrative bioinformatics. *BMC Bioinformatics*, 6:34, 2005. Journal Article Research Support, Non-U.S. Gov't England.

[83] W.M. Shui, N. Lam, and R.K Wong. A novel laboratory version management system for tracking complex biological experiments. *Bioinformatics and Bioengineering, 2003. Proceedings. Third IEEE Symposium*, pages 133–140, 2003.

[84] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Abstract metadata efficiency in versioning file systems. *Conference on File and Storage Technologies (San Francisco, CA)*, 2003.

[85] R. D. Stevens, A. J. Robinson, and C. A. Goble. myGrid: personalised bioinformatics on the information grid. *Bioinformatics*, 19 Suppl 1:i302–4, 2003. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. England.

[86] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation*, pages 165–180, 2000.

[87] M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. *On the Move to Meaningful Internet Systems*, 2888:603–620, 2003.

[88] W. F. Tichy. RCS-A system for version control. *Softw. Pract. Exper.*, 15(7):637–654, 1985.

[89] O. G. Troyanskaya, K. Dolinski, A. B. Owen, R. B. Altman, and D. Botstein. A bayesian framework for combining heterogeneous data sources for gene function prediction (in Saccharomyces cerevisiae). *Proc Natl Acad Sci U S A*, 100(14):8348–53, 2003.

[90] J. Van den Broeck, S. A. Cunningham, R. Eeckels, and K. Herbst. Data cleaning: detecting, diagnosing, and editing data abnormalities. *PLoS Med*, 2(10):e267, 2005. Wellcome Trust/United Kingdom Journal Article Research Support, Non-U.S. Gov't United States.

[91] S. Volik, B.J. Raphael, G. Huang, and et al. Decoding the fine-scale structure of a breast cancer genome and transcriptome. *Genomic Research*, 16(3):11, 2006.

[92] D. Wheeler. Using GenBank. *Methods Mol Biol*, 406:23–59, 2007. Journal Article United States.

[93] C. White. Suspected research fraud: difficulties of getting at the truth. *Bmj*, 331 (7511):281–8, 2005. Biography Historical Article Journal Article England.

[94] J. Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*. 2008.

[95] M. R. Wilkins, C. Pasquali, R. D. Appel, K. Ou, O. Golaz, J. C. Sanchez, J. X. Yan, A. A. Gooley, G. Hughes, I. Humphery-Smith, K. L. Williams, and D. F. Hochstrasser. From proteins to proteomes: large scale protein identification by two-dimensional electrophoresis and amino acid analysis. *Biotechnology (N Y)*, 14 (1):61–5, 1996. Journal Article Research Support, Non-U.S. Gov't United states.

[96] C. H. Wu, L. S. Yeh, H. Huang, L. Arminski, J. Castro-Alvear, Y. Chen, Z. Hu, P. Kourtesis, R. S. Ledley, B. E. Suzek, C. R. Vinayaka, J. Zhang, and W. C. Barker. The Protein Information Resource. *Nucleic Acids Res*, 31(1):345–7, 2003. Journal Article England.

[97] I. Xenarios, E. Fernandez, L. Salwinski, X. J. Duan, M. J. Thompson, E. M. Marcotte, and D. Eisenberg. DIP: The database of interacting proteins: 2001 update. *Nucleic Acids Res*, 29(1):239–41, 2001. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. Research Support, U.S. Gov't, P.H.S. England.

[98] S. Yang, S. S. Bhowmick, and S. K. Madria. Bio2x: a rule-based approach for semi-automatic transformation of semi-structured biological data to xml. *Data Knowl. Eng.*, 52(2):249–271, 2005.

[99] J. Zhao, C. Wroe, C. Goble, R. Stevens, D. Quan, and M. Greenwood. Using semantic web technologies for representing e-science provenance. *Semantic Web - Iswc 2004*, 3298:92–106, 2004.