MODELLING AND COMPUTING THE QUALITY OF INFORMATION IN E-SCIENCE

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2008

By
Paolo Missier
School of Computer Science

Contents

List of Tables

List of Figures

To
my mother and my father, who went too soon
Matteo and Chiara, who wonder what happened to all the playtime with their dad
Marina, whose big heart and shoulders bear it all with great patience

and Emanuele

And to those who still, with an open mind,
make an effort to tell the worthy from the worthless

Abstract

Modern experimental science, or *e-science*, increasingly relies upon the use of information integration and analysis techniques to achieve its results. A central requirement in e-science is that the necessary data and service resources be contributed by many parties within a scientific community, trascending the boundaries of individual labs. This scenario bears the promise of reducing the overall cost of science by encouraging the reuse of scientific information on a large scale. At the same time, however, there is a risk that data of poor quality, resulting for example from inaccurate experiments, may propagate out of control and contaminate other experiments. To compound the problem, the fast-paced evolution of the experimental techniques is making it difficult to investigate and standardise methods of quality control. Quality assurance for e-science information is therefore an important and largely open problem.

In this thesis we argue that user scientists should play a central role in ensuring that the third-party information they wish to use is of acceptable quality. This is difficult, however, because users, who are not part of the information production process, are often left to estimate quality of data using empirical rules that are based on limited and indirect evidence of correctness. This results in implicit quality control rules being applied in a bespoke and intuitive fashion, if at all. Our research hypothesis is that these quality rules for data acceptability are a form of latent knowledge, for which we have coined the term *quality knowledge*, where objective measures overlap with the scientists' subjective propensity to the risk of using errouneous data. In the thesis we investigate ways to make such quality knowledge explicit, and to exploit it in order to make e-science experiments *quality aware* in a principled way. Our main result is a model and architecture for *Quality Views*, i.e., quality processes that embody the user scientists' personal criteria for data acceptability. We show that, with appropriate support from software tools, Quality Views can become reusable quality components that are easily integrated into e-science experiments.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the "Copyright") and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of School of Computer Science (or the Vice-President).

Acknowledgements

I am grateful to my supervisor, Dr. Suzanne Embury for her guidance, to my advisor Prof. Carole Goble for her suggestions and for providing a broad perspective for my work, and to Prof. Andy Brass for providing challenging quality problems to us.

I would also like to thank Prof. Alun Preece and his Aberdeen staff, in particular Dr. Binling Jin, as well as Dr. David Stead at the School of Medical Science, University of Aberdeen, for disclosing the secrets of proteomics to us.

Thanks also to Dr. Conny Hedeler, whose competence and discipline in sifting through countless papers together has proved vital to charting an area of science that was alien to me; and to Dr. Mark Greenwood for the stimulating discussions.

Credit is due to several students who contributed to parts of the Qurator workbench implementation. These include Paul Waring, Richard Stapenhurst, and Jianheng Kiu.

In addition to being a great friend, Dr. Daniele Turi has contributed his priceless knowledge and expertise to the definition of the formal syntax and semantics of the Taverna workflow language.

I am especially grateful to the entire CS crowd at Manchester for providing a positive and nurturing environment. When good people understand the point of sharing (and of serendipity!), good ideas tend to emerge from almost any conversation —even when coffee is not involved.

Chapter 1

The information quality problem in e-science

The Metaphysics of Quality would show how things become enormously more coherent –fabulously more coherent–when you start with an assumption that Quality is the primary empirical reality of the world.... [†]

1.1 Quality knowledge

The term *e-science* has come to refer, in recent years and especially in the life sciences, to the practice of performing scientific experiments in which data processing and analysis, in addition to the more traditional lab operations, play a substantial role [GGS+03]. This new type of *in silico* experiment relies on a rich base of data and services that are contributed on a large scale by many labs, either as community-driven, autonomous efforts, or as commercial ventures. To the extent that sufficient IT support for data and services is provided, this new form of science holds the promise to significantly reduce the time and effort required to produce meaningful research output. In the past few years, progress in this direction has been documented in active research areas in the life sciences, including genomics and post-genomics (the study of gene expression and gene products) as well as, more recently, in systems biology [IGH01].

[†]R. Pirsig, Lila: An Inquiry into Morals, 1991

The promise of e-science rests on the central assumption that the information produced by one lab can be reused by others, possibly in new ways and after a process of integration and adaptation (as witnessed by a number of cooperative integration projects, including Tambis, Ispider, GIMS, and more [SBB+00, BEF+05, CPH+03]). When this is the case, the lifetime of scientific information can extend over many generations of experiments, and its value can propagate through a scientific community.

With the value, however, also comes the risk that errors in the data, if undetected, may propagate as well, causing damage in uncontrolled ways. This is a well-known concern, although estimating the risk of scientific damage is proving complicated. Only a handful of studies are available, for example, to assess the risk of automated functional predictions on genes and proteins [WKA04, BDM02, Bre99, DV01], or to estimate the cost of using the results of a faulty microarray experiment [BEPG+05].

To mitigate this problem, obvious lines of attack that come to mind include, on the data provider side, the early detection and correction of errors in the data, and on the consumer side, the validation of data prior to its use. While some of this is indeed being done, dealing with quality appears to be a difficult problem. One reason is that scientific data is often new and difficult to understand, itself being the subject and the result of experimental, cutting-edge research; and furthermore, the experimental techniques used by scientists evolve rapidly, following the advances of the technology that it is based upon. This makes it difficult to establish common criteria for quality control, and to associate standard quality control procedures to the data production process, as would instead be expected in a stable industrial context. A few initiatives, mentioned later, are only now being undertaken to establish guidelines for the submission and documentation of experimental results, notably in some areas of post-genomics.

A second reason, discussed later in more detail, is that different consumers of scientific data may tolerate different levels of quality, depending on the application for which the data is used.

We have recently surveyed the state of the art in quality control for post-genomics [HM07], as discussed briefly in the next chapter. The main point that emerges from the analysis is that the idea of defining and enforcing quality controls on data and processes is becoming pervasive but is often *latent and implicit*. This is especially true on the information consumer side, where the problem is

to establish criteria for validating third party data. One may indeed encounter, within a scientific workflow, steps that involve some form of data validation; but the way they are encoded does not follow a recognizable paradigm, making it hard to explain it to others and to reuse as part of similar experiments: there appears to be no principled way to make the validation criteria explicit and to expose them as "first-class citizens" as part of data processing.

The main idea pursued in this thesis is that "quality" is a form of latent knowledge, which is grounded in the scientists' expertise and reflects their implicit assessment, often biased and partly subjective, of the risk of using erroneous data. We make the hypothesis that it is possible to make such *quality knowledge* explicit and to encode it in such a way that it becomes a recognizable part of data processing. Also, we see the task of eliciting knowledge about information quality as an experimental process in its own right, and one that parallels the scientific processes that produce and consume the information. Even more importantly, it should be possible to separate the objective from the subjective components of quality knowledge, for example, objective indicators of likely errors from the subjective importance attributed to them.

If this hypothesis is correct, we then can hope to make quality knowledge at least in part reusable across processes that deal with similar types of data and similar scientific problems; we see reuse as the key to reducing the "cost of quality", at least at the scale of a sub-field of e-science. Ultimately, we hope to show that, by taking a principled approach to quality modelling, we can reduce the cost of making existing scientific data processes *quality-aware*.

What makes this research hypothesis difficult to prove is not only the latent nature of quality knowledge, but also its diversity: in order to achieve any degree of reusability, we have to provide suitable abstractions that capture the common features of many different quality criteria, and yet are still useful in practice. This is where we hope to offer user scientists a quality management environment that helps them *elicit* quality knowledge and put it to use as part of the experimental process.

Before we move on to state the specific objectives and structure of the research work described in this thesis, let us make the definition of "quality" more precise, using a simple, everyday example.

1.2 Quality of data and quality-based decisions

We use the term *quality of data*, in a broad sense, to indicate properties of the data that describe various types of error condition. Consider the familiar example, used for instance in [SMB05], of a collection of database entries that describe films. These entries may be subject to various types of error. The name of a director may be misspelled; the association between a director and a film may be incorrect; some attributes of a film may be missing; the biography of a director may be out of date; and some unforgettable film may be missing altogether.

We can use a collection of distinct data properties to describe each of these errors, or their combinations. In the data quality community some level of consensus has been reached to give names to the most common properties, collectively referred to as *quality dimensions*. Thus, a misspelling is a *syntactic inaccuracy*, a wrong association is a case of *semantic inaccuracy*, missing attributes denote *field incompleteness*, an outdated biography is not *current*, and when films are missing, the database is *incomplete* with respect to some universe of known films.

We can therefore identify the quality of a specific film entry in the database as a point in the multi-dimensional space defined by some subset of these quality properties (a more accurate definition would have to consider data at different levels of aggregations, for example when films are missing, we associate incompleteness to the entire database rather than individual films; but this approximation will suffice for the sake of our argument). The point is that, to the extent that we can devise error detection procedures to compute values for each of these properties, quality forms a well-defined space of metadata. Additionally, if we assume that suitable ordering relations are defined on each of the quality dimensions, we can define quality improvement activities as any procedures that move the quality vector in some positive direction, making error correction quantifiable.

Over the years, the data management community has been developing a variety of error detection and correction techniques, which apply with varying assumptions to different database configurations; some of these are well-established and are surveyed in Chapter 2 (a related, important line of research concerns how to make data processing tolerant to errors, when these can be detected but not corrected). As a result, a simple framework based on quality dimensions has been deemed sufficient as a reference for addressing data quality problems.

This is, however, a reflection of the data providers' perspective on the data; if we consider the data consumers' definition of quality, we find that this foundation

is useful but not sufficient: one important concern that data consumers have is whether or not the data is of *sufficient* quality to be used in applications. In other words, it is the interpretation of quality properties in the context of data usage that is of interest. Let us illustrate this with another familiar example. Imagine that, while travelling on a motorway, drivers see electronic panels warning of a queue a few miles ahead. The signs are there to help drivers decide whether to stay on the motorway or take an early exit. The question is, can drivers actually rely on the information they see and use it to inform their (subjective, perhaps) decision process? What if the signs are hours old, or inaccurate in other ways? A bad judgment in this case may result in wasteful detours on secondary roads. The presence of suitable meta-information, for example "this message was updated five minutes ago based on live CCTV footage of an accident" would certainly help, but is not normally provided – too much information can be confusing, is probably the motorway agency's argument, and it cost additional effort to produce. Either way, the information consumer is faced with the problem of *estimating* the quality of the data in the absence of a decisive proof of correctness. Drivers familiar with the area, for example, may rely on their prior experience, or otherwise decide to trust the agency on the grounds of its reputation. We can carry this little example further, to point out that drivers may not always need information of perfect quality: if leaving the motorway is a safe option anyway, for example, then knowing that there is actually a queue ahead may be less relevant. The last point is an important one: the need for quality – and consequently, the need to produce good quality estimates, is dependent upon the context of use of the data.

Returning to our e-science domain, we argue that this paradigmatic example is representative of a typical scenario where scientists must resort to empirical judgment on whether they can reliably make use of third party information, when meta-information regarding its quality, as defined earlier, is either incomplete or not available at all. In making this judgement, users must take account of the risk of using faulty data, and conversely, of discarding important data. To make the scenario more concrete, in the next section we present two examples involving quality assessment in e-science.

1.3 Examples of quality knowledge for life sciences data

Our first example, set in the biology field of qualitative proteomics (the study of the set of proteins that are expressed under particular conditions within organisms), describes a case of quality knowledge that has been elicited through independent experimentation, and can now be used to make a *protein identification process* quality-aware. We have used this example in several papers, including [MPE+05], [PJM+06] and [PJP+06], although we have not contributed to the scientific research in this area. The second example concerns the functional annotation of protein entries in a well-known database, Uniprot. In this case, it is the quality of the annotations that is in question.

These two examples illustrate two common and complementary scenarios. In the first, quality assessment is applied to data that is computed in the course of an *in silico* experiment; while in the second, the data is stored persistently in a database. Ideally, it should be possible to use similar techniques to add quality controls to both. We are going to use the former of the two examples as a use case throughout the rest of the thesis.

1.3.1 Quality assessment in qualitative proteomics

The term *protein identification* refers to the problem of understanding the regulation and function of proteins that are present in a cell sample. In a typical proteomic experiment, several different samples are analysed using one of several available techniques, for example 2-dimensional gel electrophoresis (2DE). This results in a distribution of protein spots on a gel. Many hundreds of proteins can be separated from a single sample in this way. In *qualitative proteomics*, the gel represents the initial experimental artifact used to identify the specific proteins that are present in the original sample.

Specifically, one technique that is used for this task is called peptide mass fingerprinting (PMF). In this technique, the protein within the gel spot is first digested with an enzyme that cleaves the protein sequence at certain predictable sites. The resulting protein fragments, called peptides, are extracted and their masses are measured in a mass spectrometer. This yields a list of peptide masses, or a "fingerprint", for each spot. The fingerprint is then compared against theoretical peptide mass lists, derived by simulating the process of digestion on

sequences extracted from a protein database (e.g. NCBInr[1]). Since, for various reasons, it is unlikely that an exact match will be found, the protein identification search engines (e.g. Mascot[2]) that perform this task typically return a list of potential protein matches, ranked in order of search score. Different search engines calculate these scores in different ways, so their results are not directly comparable. It may therefore be difficult for the experimenter and subsequent users of the data to decide whether a particular protein identification is acceptable or not.

Two main types of data quality problem arise in this type of experiment:

- Protein identification is intrinsically subject to uncertainty, due to limitations in the technology used, experimental contamination, an incomplete reference database, or an inaccurate matching algorithm. The results may contain false positives, and it is often the case that the correct identification is not ranked as the top match.

- Experiments performed at different times, by labs with different skill levels and experience, and using different technologies, reference protein databases and matching algorithms, are difficult to compare.

Therefore, it would be useful for scientists seeking to interpret the results of proteomic experiments, to be able to define a quality test that applies to a list of protein matches, to identify the likely false positives; and to apply the test repeatedly, on many experimental datasets. Such functionality would be particularly useful to scientists wishing to compare protein identification results generated by other labs with those produced within their own.

There are three readily accessible indicators that can be used to rank the identified proteins, called *hits*, and which are independent of the particular search engine used:

- *Hit ratio*: the number of peptide masses matched, divided by the number of peptide masses submitted to the search:

$$\texttt{Hit Ratio} = \frac{\texttt{Matched Masses}}{\texttt{Submitted Masses}}$$

  This is a measure of confidence in the hit: while, ideally, the protein identified should contain most of the peaks in the spectrum, the presence of other components and other types of noise may reduce the hit ratio.

[1] ftp://ftp.ncbi.nlm.nih.gov/blast/db/blastdb.html.
[2] http://www.matrixscience.com/

- *Mass coverage*: the number of amino acids contained within the set of matched peptides, expressed as a fraction of the total number of amino acids making up the sequence of the identified protein, and multiplied by the total mass (in kDa) of the protein. This indicator will have a high percent value for a more complete match, and a lower value for a partial match.

- *Excess of limit-digested peptides* (`ELDP`). This accounts for the possibility that some cleavages are missed during protein digestion, resulting in an incorrect peptide sequence. Ideally, a complete (limit) digest will have been achieved during PMF, in which case the number of missed cleavage sites would be zero. However, in practice a small number of missed cleavages are to be expected, and the algorithms try to take this into account. ELDP represents the predicted missed cleavages, and is calculated by subtracting the number of matched peptides containing a missed cleavage site, `MCP`, from the number of peptides with no missed cleavages, `NMCP`: $\texttt{ELDP} = \texttt{NMCP} - \texttt{NMCP}$.

It has been shown in [SPB06] that a simple linear combination of these three indicators:

$$s = \alpha\ \texttt{Hit Ratio} + \beta\ \texttt{ELDP} + \gamma\ \texttt{MC}$$

is a good score for proteins in the hit list, providing an effective, and inexpensive, criterion for indentifying false positives.

From the point of view of the automated computation of the score, a crucial issue is the availability of the indicators, for each protein in the hit list. These values are routinely produced during the course of the so-called *wet lab* part of the experiment, performed in the traditional biology lab. Whether this information is available when the follow-on *in silico* portion of the experiment is executed, however, depends on the data format used to record the experiment itself. One example is the Pedro data model [TSWR03], which describes one of the data formats currently in use for storing descriptions of proteomics experiments, using an XML syntax[3]. This model currently does not account for all the required indicators. Thus, in practice the experimenter will have to choose among the score models that are actually computable within a specific data processing environment. In the following chapters, we are going to assume that enough information is provided to compute `Hit Ratio` and `MC`, but not `ELDP`, imposing the use of a less

[3]http://pedro.man.ac.uk/files/PEDRoSchema.xsd

discriminant score model, $s_1 = \alpha$ HR $+ \beta$ MC, that only involves these two quantities. Indeed, a number of protein identification tools satisfy this assumption. Among these are Imprint, an in-house, freely available software tool for PMF developed at the University of Manchester, and MASCOT [PPCC99], a commercial software product. The output obtained from these tools includes some of these indicators used in practice to compute protein hit lists from 2DE experiments.

Once the score metric is available for each element in the protein hit list data set, scientists may set acceptability thresholds according to additional, personal criteria, based for example on a trade-off between completeness of the result, and the likelihood of including false positives. It is easy to see how, once its effectivess has been demonstrated experimentally, a metric of this sort can be weaved into a software tool that allows user-scientists to experiment with various setting of the acceptance threshold, and see their effect on their *in silico* experiment.

1.3.2 Quality of protein functional annotations

In the example just presented, the quality indicators are computed from the same process that produces the data, e.g. the Imprint tool. As a consequence, their values only become available at process execution time, and therefore quality assessment can only be performed as part of the scientist's *in silico* experiment. A complementary scenario, just as common, occurs when quality assessment is performed on persistent data, either during a user query, or on the entire database, independently of any user experiment.

Consider for example the large Uniprot database (www.uniprot.org), containing rich descriptions of proteins. The database is curated by human experts, who systematically annotate the entries with a description of the expected protein functions. The annotations are stored in the GOA database (www.ebi.ac.uk/GOA/), and consist of terms from the Gene Ontology (www.geneontology.org/). These annotations often reflect *predictions* of protein function, which are based either on experiments reported in journal publications, or on the algorithmic analysis of similarity to other proteins whose function is known with certainty. It has been argued [LSBG03] that the reliability of the annotations varies depending, among other factors, on the source of information used by the curator.

One may therefore formulate a quality hypothesis for Uniprot annotations where the underlying indicators include the sources used for annotations. In fact,

the GOA database conveniently provides a set of *evidence codes*, issued by the curators to justify their annotations. These include for example "Inferred from Direct Assay" to indicate that the annotation is based on knowledge of an actual experiment; "Inferred from Electronic Annotation", indicating that the prediction is based on protein sequence similarity obtained by an algorithm; and "Traceable Author Statement", denoting an annotation based on a review paper where the experiment can be traced, i.e., to a repository of experiment descriptions (like Pedro [KMGa04]).

An attempt to establish a correlation between evidence codes and correctness of GOA entries can be found in [LSBG03], although no conclusive study is available, to our knowledge, that provides an authoritative quality metric based on the evidence codes criteria.[4] The very problem of establishing quality in this case, however, show that this example fits our model of quality as the result of an experimental process: the "quality hypothesis" requires a formalization of relevance of the evidence codes, for example in the form of a score model.

1.4 Quality knowledge management as a research problem

In this section we state our research objectives, outline our technical approach, and anticipate our results. As a starting point we make several observations regarding the nature of the information quality problem in e-science; we derive them from the preceding examples as well as from our recent survey on quality issues in post-genomics [HM07] (described in more detail in the next chapter).

Firstly, we note that concrete definitions of quality criteria are often given in the form of decision procedures regarding data correctness. The procedures often encode empirical rules that are used to decide whether there is enough support for the hypothesis that the data is correct. This reflects a distinctly user-centric perspective, as we have anticipated in our earlier example in Section 1.2.

Secondly, the complexity of the data and of the application domain makes it difficult to obtain conclusive evidence regarding the presence of errors: the information available to the scientist is often insufficient to determine data correctness

[4] One may choose to consider additional evidence, such as the frequency of annotation updates compared to the update frequency of the underlying protein data. But the relevance of such evidence has not been proven.

with certainty. Therefore the scientist's rules and procedures must encode a predictive model, which is often determined on the basis of prior experience dealing with similar types of data. Ultimately, the rules encode the scientist's intuitive quality knowledge, with some inevitable approximation. The users' empirical acceptability criteria express a combination of tolerance to data errors, i.e., situations where data that is somewhat incorrect can still be used, and propensity to risk, i.e., when it is not certain whether the data is correct or not.

Thirdly, the effort required to define the decision procedures, that is, to elicit the quality knowledge, is often substantial, with the result that scientists may instead resort to simple rules of thumb, if any at all.

And finally, the rules tend to be tailormade for specific data types, and are difficult to generalize. A model of quality that is shown to work well on protein identification data conducted using PMF technology, for example, fails to perform correctly when the same data is obtained using other technologies, e.g. "MS/MS" [LZRA03, NKKA03, NA04].

1.4.1 Aims and objectives of the research

With the term *quality knowledge* we denote a combination of the elements just mentioned, namely predictive estimation of correctness, tolerance to errors, and propensity to risk. Based on this definition, we set out to find ways to exploit such knowledge by making it available to applications. Thus, our main research objectives are, first of all, to find a characterisation of quality knowledge that is suitable to capture these elements; and secondly to show that, using software engineering and data management techniques, we can support the practical need for user scientists to add quality controls to their applications. In doing this, we hope to show that quality knowledge is, to some extent, reusable across data processing applications that deal with similar types of data. If this is the case, then we can turn customized quality controls into commodities that can be added to applications when needed, with little effort.

One novel aspect of this research is that it is focused on a consumer-oriented, personalized computation model of information quality, which takes into account the elements mentioned above. This is in contrast to the distinctly provider-oriented and data-centric models that have been at the center of traditional DQ studies.

The management of quality knowledge, from its elicitation to its encoding in

such a way that user scientists can make effective use of it in their applications, is a non-trivial research problem. We have already mentioned some of the reasons for its complexity: the diversity of the quality criteria, which are rooted in the types of data and their applications; the experimental nature of the data and of the scientific processes that produce and consume it, which leads to a lack of established and accepted quality standards; and the cost of quality control. We also note that available research results in the traditional area of data quality are of only limited help: while numerous, sometimes very specific DQ techniques have been developed to perform data reconciliation in relational (and, more recently, semi-structured) data, few of the available toolkits can be of direct help to scientists. We will elaborate further on this point in the next chapter, where we explore the data quality landscape.

We observe, however, that some current trends in e-science give us suggestions on how to successfully address this problem. We refer in particular to the increasing proportion of *metadata* that experimenters associate with their experimental results. Metadata comes in various forms and serves a number of purposes. One prominent example is the detailed documentation of experimental procedures, a sort of "electronic lab book". One of the main reasons for collecting detailed process information, known generically as *provenance* [SPG05], is to provide the community with a detailed trace that describes how experimental results were obtained, often following their publication, as well as to enable third parties to repeat the experiment. In other contexts, provenance is used as a persistent process trace that can be used to explain the outcome of the process [MGM+07]. When the process is formally encoded, typically as a workflow, and its execution automated, then collecting provenance metadata becomes a matter of routine; indeed, some e-science computing infrastructures are now provenance-aware [BC07].

This increasing wealth of metadata has the potential to become a precious source of evidence to be used when formulating quality hypotheses. In this respect, it is encouraging to see that the importance of provenance is gaining recognition within the e-science community, and infrastructure is being built to support its collection and exploitation. In fact, process provenance is not the only source of rich metadata. In important areas of the life sciences, initiatives to standardize on a rich documentation of the experiments (possibly human-defined) are met with interest. Organizations such as the Microarray Gene Expression Data Society (`www.mged.org`) in the field of gene expression studies (transcriptomics)

and The Human Proteome Organisation (`www.hupo.org`) are notable examples.

1.4.2 The information quality lifecycle

As a first step towards our goals, we propose to model the user process of quality knowledge elicitation and exploitation along the principles of experimental science: user scientists make initial hypotheses, in this case regarding the predictive power of certain indicators of data correctness; then they test their hypotheses on the data, analyse the results, and revisit their hypotheses if needed, possibly collecting additional supporting evidence, in a cycle of incremental refinement. We refer to this experimental process as the *Information Quality lifecycle*. In this section we propose a first, high-level description of this process, illustrated in Figure 1.1, and use it to make out research objectives more precise. In the next chapter we are going to spell out specific tasks that compose the lifecycle, so that we can use it as a reference throughout the thesis to keep track of our progress.

The illustration in Figure 1.1 shows the lifecycle as consisting of two loops: the inner loop (on the right in the figure) concerns the phase of quality knowledge elicitation, while the larger outer loop (on the left) represents the phase of quality exploitation by user scientists.

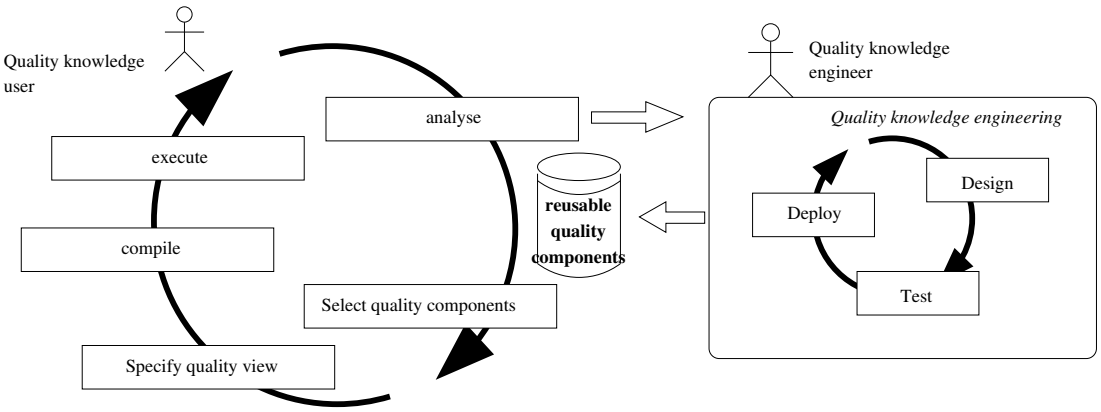


Figure 1.1: The Information Quality lifecycle

In the elicitation phase, the scientist's problem is to define a predictive model for quality that is applicable to certain specific data types and is based on some quality evidence, typically some form of metadata (provenance, for instance, if appropriate). We use the term *quality knowledge engineer* to refer to role of the scientist who is engaged in this modelling activity. The inner loop accounts for such iterative process of model design. Once it is tested, the quality model

is made available to other scientists, referred to as *quality knowledge users*, as a software component. This requires "advertising" the component to the community. For this, we will define a new model of IQ concepts, i.e., an ontology that accommodates symbolic definitions of user-defined quality functions (i.e., functions that take an input dataset and return a quality value for each of its elements). Quality knowledge engineers are expected to update the ontology with their new functions.

In the exploitation phase, the goal of the user scientists is to determine whether any of the quality functions defined in the IQ model can be used as a quality control component as part of their own applications. This goal translates into a sequence of specific tasks, namely (i) the discovery of available quality functions, (ii) their composition into a quality process, (iii) the testing of the quality process on the user's own data, and (iv) the analysis of the results, to see whether the selected quality functions and their composition are appropriate for their data. As anticipated, the experimental nature of this process suggests that these tasks should be part of a loop of quality knowledge exploitation.

To support these tasks, we have proposed a form of process specification that we call a *Quality View* (QV) [MEG+06]. A QV specifies an abstract and implementation-independent composition of quality functions. We have chosen the term Quality View to suggest an analogy with database views, which provide different "virtual" perspectives on the same data, depending on the user's application needs.

Quality Views are at the core of our proposed model for supporting the outer loop tasks just mentioned. Specifically, we are going to design ontology-based tools for the discovery of quality functions, show that Quality Views can be automatically compiled into executable *quality processes* (in the form of software services) and that these services can be integrated with the users' data processing applications, for example a scientific workflow. The result of this integration is an enhanced, "quality-aware" data processing application that, when executed, includes the new quality controls as specified by the user. To support the final result analysis step, we also propose the notion of *quality provenance*, a way to record the execution of a quality processes that makes it possible to determine their impact on the data.

Finally, at the end of each iteration the users must determine whether their quality process should be refined. This may involve either the selection of new

quality functions, i.e., a new iteration on the outer loop; or the refinement of the quality functions themselves, i.e., a feedback into the inner loop.

Our overall objective is to support the users in various phases of this lifecycle. We focus for the most part on the outer loop portion. Thus, we assume that suitable "nuggets" of quality knowledge have been discovered by scientists. The problem of supporting the elicitation phase is discussed briefly in the last chapter, as a matter for future work.

Within this scope, our contributions include two main elements: a conceptual model for IQ, which reflects the principles outlined in this chapter and facilitates the reusability of quality functions; and a suite of software tools to help users make their data-centered applications quality-aware. Together, these elements form the *Qurator information quality workbench* [MEG+07][5]. In the next section we precisely characterise our contributions in terms of the workbench elements. Towards the end of the thesis (please see Section 6.4), after all these elements have been introduced, we will establish a clear relationship between the workbench and the tasks in the IQ lifecycle, in order to clarify the scope of our contributions.

1.4.3 Technical approach and research contributions

Figure 1.2 gives an overall picture of the workbench components. We discuss our contribution on each of its elements.

An ontology for information quality concepts. As anticipated, our first contribution is a conceptual model for information quality, shown at the bottom in the figure. The model consists of two parts: a characterization of quality knowledge in terms of data classification and data ranking (on the right in the figure), that provides a formal foundation for quality functions; and (on the left) a semantic model, i.e., an ontology, that provides a logical structure for IQ concepts that facilitates the sharing and reuse of those functions. A noteworthy aspect of the model is the separation between an immutable *Upper ontology*, consisting of abstract quality concepts and their relationships, e.g. "quality evidence"; and a growing, user-contributed

[5]The name "Qurator" comes from the EPSRC project that funded this research from 2005 to 2007 (GR/S67593 & GR/S67609), under the title *Describing the Quality of Curated e-Science Information Resources*. Project partners included the School of Computer Science at the University of Manchester and the School of Computing at University of Aberdeen. The work described in this thesis only includes research carried out at Manchester.

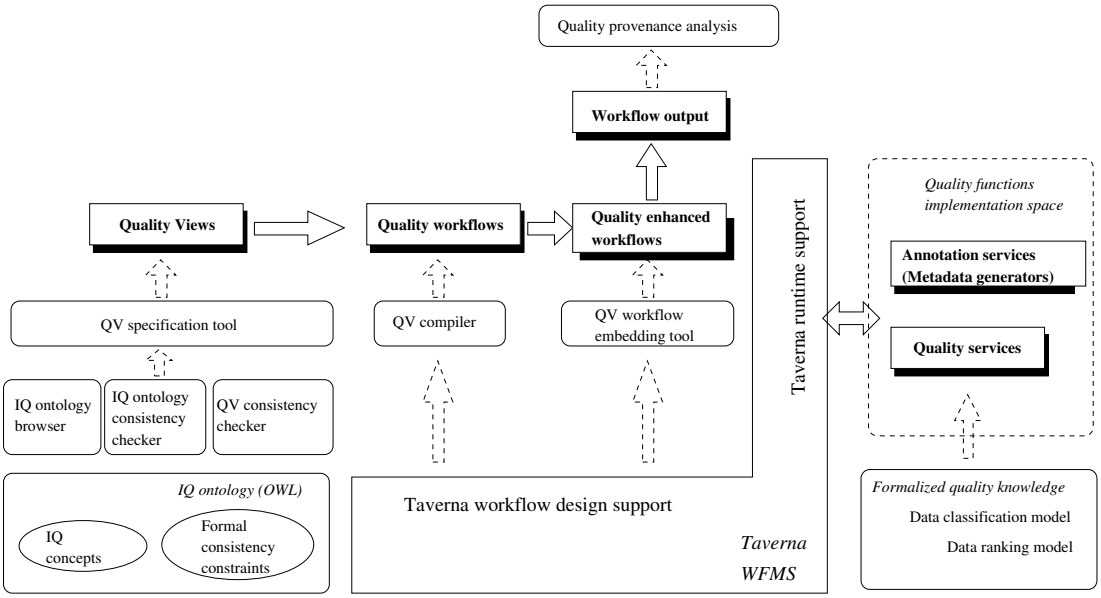


Figure 1.2: Components of the Qurator information quality workbench

lower ontology that extends the upper ontology with domain-specific concepts, for instance "Hit ratio used as quality evidence in the context of certain proteomics data". We use the ontology to represent the output of the quality knowledge elicitation loop, in abstract terms. We have used the OWL Semantic Web language to define the IQ ontology. This has made it possible to provide an axiomatic definition of IQ concepts, using the Description Logics operators available in OWL. The implementation counterpart of this growing collection of ontology concepts is a corresponding collection of user-defined Web Services that realize the quality functions (on the right in the figure).

A process model for quality function composition (Quality Views). Quality Views (QV) are abstract specifications of quality processes and represent the core artifacts of Qurator (top left). Regarding Quality Views we make the following specific contributions:

- A formalization of the Quality View language and semantics;
- A formal definition of *Quality View consistency*, based on the semantic definition of its composition functions (i.e., as concepts in the ontology). As we will see, a consistent QV is one that satisfies certain constraints defined on the ontology using logical axioms;

- An interactive software environment, denoted "QV specification tool" in the figure, that allows user scientists to specify consistent Quality Views. The consistency guarantees are made possible by exploiting automated reasoning on the ontology;
- A Quality View compiler that produces an executable service. Here we assume that quality functions are themselves implemented as services (specifically, as Web services), and therefore they can be used as elementary building blocks to compose Quality Views. Thus, the compiler translates the abstract QV definition into a composite process, specifically a workflow. We provide a formal description of the compiler that is based on the QV semantics, mentioned above, and show that, under certain assumptions, the semantics of the resulting workflow is consistent with the formal semantics of the abstract QV.
- A model for embedding the quality workflows that result from the compilation into existing scientific workflows. With this model, Quality Views can be deployed as part of the scientists' *in silico* experiments.

A model for generating quality functions code. Although we have stated earlier that quality knowledge elicitation is out of the scope of our work, we find that once the logic of a quality function has been defined, we can support the user task of providing a service-based implementation for the function, by generating some of its code. This contribution is based on the observation that our characterization of quality knowledge makes quality functions similar to each other in structure and, to some extent, in their implementation logic. We exploit this similarity to derive a model for automating the generation of implementation stubs for them, and discuss the potential and limitations of the approach.

A model for adding Quality Views to XML query processing. Observing that Quality Views can be used not only in the context of workflow processing, but also of query processing, we define (i) an architecture for the integration of Quality Views into XML data processing, i.e., using XQuery, and (ii) a simple extension to the XQuery syntax to make it easy for users to request the invocation of their Quality Views as part of XQuery processing. As a result, users can add quality-based conditions to their query, to achieve a quality-based filtering of the result.

A model of quality provenance. As mentioned earlier, the analysis of Quality View results is the last task in the outer loop of the IQ lifecycle. Having defined quality as a decision process, we observe that users should also be able to understand why a certain decision, e.g. to accept or reject a data element, has been made. To address this problem we define a model of quality provenance that describes the execution of a quality process in detail. At the end of a workflow execution, users may query the model to determine how the quality process has affected the data, and, most importantly, on the basis of which quality evidence. We have implemented this functionality, denoted "Quality provenance analysis" at the top of Figure 1.2.

To summarize, each box in Figure 1.2 (with the exception of the Taverna workflow management system) represents an original contribution. We have also implemented the example quality functions described throughout the thesis, namely for the proteomics running example, as Web Services (thus, they are part of the implementation space shown in the figure). The overall result is a workbench for information quality management that realizes the main objectives set forth in this chapter.

1.4.4 Thesis organization

The rest of the thesis is organized as follows. We discuss the proposed conceptual model for information quality in Chapter 2, after presenting a critical survey of the state of the art in data quality management both in the field of post-genomics (to which proteomics belongs), and from the point of view of general data engineering. We present the IQ ontology in detail in Chapter 3, along with the axiomatic definition of the IQ concepts, and the ontology-based QV composer. The Quality Views model, its syntax and formal semantics are presented in Chapter 4, while Chapter 5 is devoted to the translation of Quality Views into workflows. In Chapter 6 we discuss the issues of code generation for quality functions, of quality-aware XML query processing, and of quality provenance. This leads to a complete presentation of the Qurator workbench as an architecture for the support of the IQ lifecycle. We conclude in Chapter 7 with a discussion on current limitations of this work, and we propose a research agenda for its continuation.

Chapter 2

Modelling quality knowledge

What is good, Phœdrus, and what is not good...
need we ask anyone to tell us these things? †

In this chapter we analyse in more detail the nature of information quality problems in e-science, and propose a conceptual model for IQ that will form the basis for the rest of the research presented in this thesis.

The chapter is organized into three interrelated parts. In the first part (Section 2.1) we propose an analysis of quality control issues in the field of post-genomics (the study of gene expression and gene products), currently a very active area of research. We proceed by generalization from the example of the preceding chapter on quality control for proteomics experiments. As we have observed in our recent survey on this topic [HM07], experiments in this area of science tend to follow a common, predictable pipeline structure, consisting of a traditional *wet lab* portion, followed by a newer *dry lab* portion with a strong data management and analysis component. Noting that different quality issues and control techniques can be associated to each of the stages in the pipeline, we propose a simple framework for information quality analysis that reflects this structure.

Here, we anticipate the main observation that emerges from the analysis: while the importance of developing quality control techniques is clear to scientists, we do not often see these techniques used explicitly in *in silico* experiments. This is especially true for the data-intensive portion of the experiments, where new

†R. Pirsig, The Zen and the Art of Motorcycle Maintenance, 1974

issues of data quality overlap with the more traditional issues of process control in the lab. Instead, we see a number of bespoke procedures, often implicit or hidden within the lab practice, that are ultimately designed to determine whether the outcome of an experiment is acceptable as a valid scientific result. In the introductory chapter we have argued that defining such procedures is a complex problem, and we have introduced the term *quality knowledge* to emphasize the importance of making them explicit and, hopefully, reusable.

In the second part of the chapter (Section 2.2) we survey the landscape of current Data Quality (DQ) practice and research, as a natural source of relevant technology for quality control. Here we find mainly techniques to address problems of data reconciliation and consistency, and duplicate elimination; these are typically applied to traditional data settings, for example prior to performing data integration and warehousing. These represent only a small portion of the e-scientists' problems, however: the broader issue of the correctness of scientific information remains open. One reason for this is the complexity of the data and of the application domain, which makes it difficult to obtain conclusive evidence regarding the presence of errors: information correctness is often at best estimated, rather than established with certainty. Thus, quality assessment calls for the use of techniques for correctness estimation. We also observe that the acceptability criteria employed by scientists who are faced with uncertainty regarding correctness of information, are often empirical. Such criteria express a combination of tolerance to data errors, i.e., when data that is somewhat incorrect can still be used; and propensity to risk, i.e., when it is not certain whether the data is correct or not. Based on these observations, we define quality knowledge as a combination of these elements: predictive estimation of correctness, tolerance to errors, and propensity to risk.

In the third part of this chapter (Section 2.3) we address the problem of how best to capture these elements, and we propose an original computational model for information quality. More specifically, we propose a user-centred model that is based on a combination of two elements: an assignment of data to classes, and a decision (for instance, accept / reject) that is associated to each class. The classification is objective but at the same time predictive, while the decisions capture a subjective perspective and reflect the user's propensity to risk. We conclude in particular that the main tools available to the scientists are not so much traditional algorithms for data quality, like those discussed in our earlier survey, but

rather those for knowledge discovery and management. Having defined quality knowledge in terms of data classification problems, we then refine our reference IQ lifecycle of Figure 1.1, to reflect this model.

Finally, we argue that a key to making computational quality cost-effective is to facilitate the reuse of the quality knowledge across a large number of applications, and within a scientific community. With the goal of reusability in mind, our model is designed to capture the common features of the users' decision process, abstracting from the specific type of quality knowledge. Thus, it represents a foundation on which we later build our Qurator workbench.

2.1 Quality control in post-genomics

In post-genomic biology, high-throughput analysis techniques allow a large number of genes and gene products to be studied simultaneously. These techniques are embedded in experimental pipelines that produce high volumes of data at various stages. Ultimately, the biological interpretation derived from the data analysis yields publishable results. The quality of these results, however, is routinely affected by the number and complexity of biological and technical variations within the experiments, both of which are difficult to control.

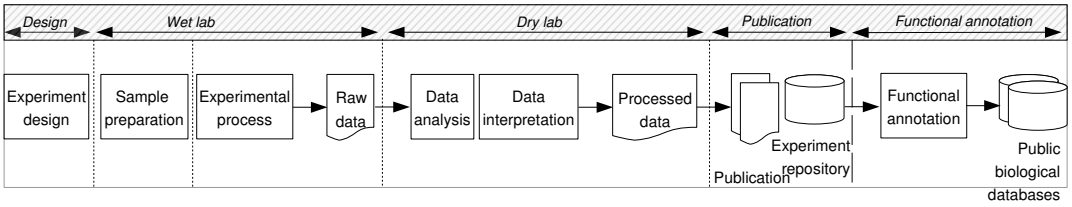


Figure 2.1: Abstract view of a typical data processing pipeline in biology

In abstract, an experiment consists of a recognizable sequence of generic steps (see Figure 2.1), which begins with the design of an experiment for some scientific hypothesis, once the constraints imposed by the technology and the equipment are accounted for. The biologist then performs a *wet-lab* experiment, which usually results in some form of raw data output. Biological variability is usually accounted for either by performing the same experiment on multiple samples, or by repeating it on the same sample. The *dry-lab* portion of the experiment involves data processing and analysis. Scientists are increasingly making use of workflow technology to automate some of the dry-lab tasks.

In the introduction we described the problem of identifying certain proteins of interest within a sample, one of the core problems in *qualitative proteomics*. If we cast this type of experiment as an instance of the generic pipeline just described, as we have proposed in [HM07], we can then associate different types of quality problems with each of its stages. In fact, the general quality framework can be used for other types of post-genomic science as well, notably transcriptomics. Figure 2.2, with the proteomics experiment shown at the bottom, illustrates this approach.

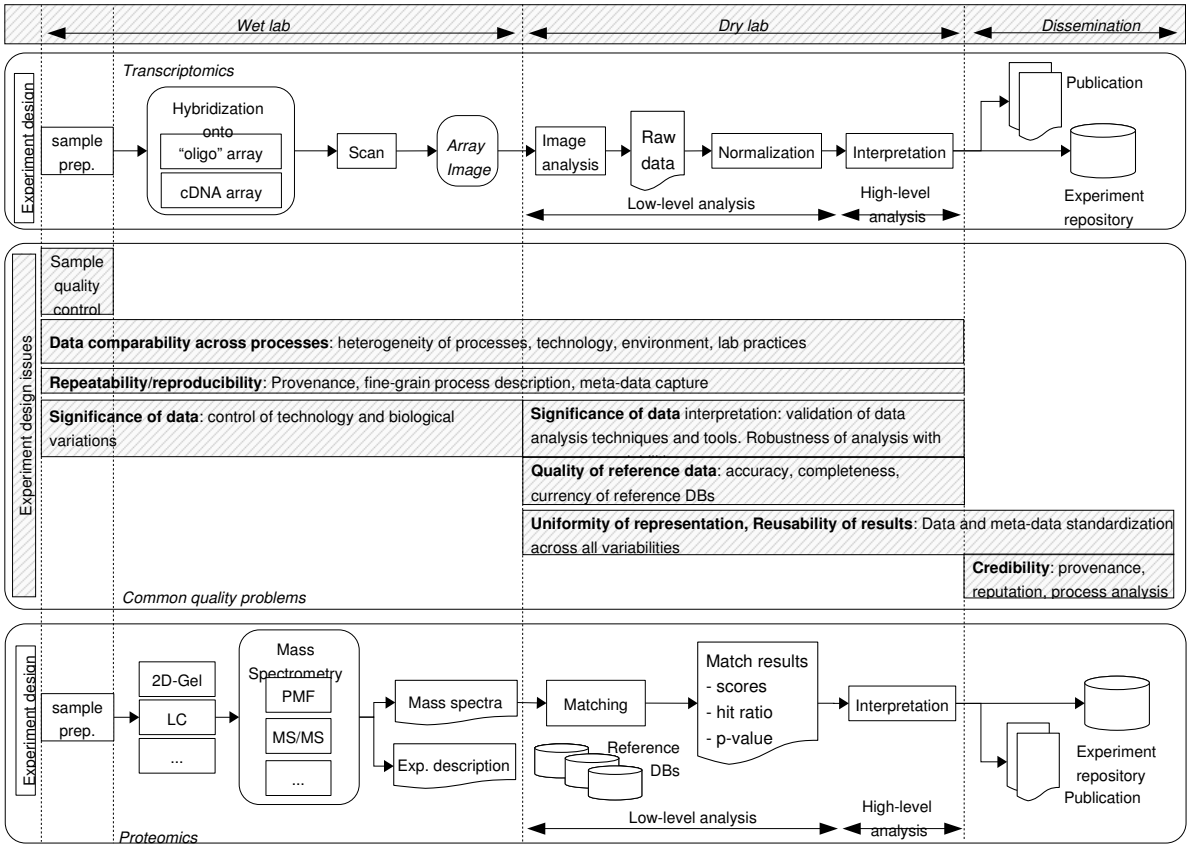


Figure 2.2: An experimental pipeline for qualitative proteomics and transcriptomics, and associated quality issues

The "swimlanes" in the middle of the figure denote the main types of quality issues and the phases of the experimental pipeline to which they apply. In particular, by "data significance" we mean that the scientist can trust the data to be the genuine result of an experiment that is not flawed; thus, we take it as a general definition of accuracy of experimental data. In the companion Table 2.1 (reproduced from Table 4.3 in [HM07]), we give more details regarding the

specific problems and techniques in this area, namely: improving the sensitivity of protein identification at the level of technology; improving on the score models, searching and matching algorithms; and improving the underlying statistical models for avoiding false positives.

A general conclusion that we draw from this survey is that a number of quality problems are addressed by promoting a rich description of experimental metadata, that is, of the experiment itself. This is true for important other areas mentioned in the table, such as data comparability, uniformity of representation and reusability of the results. As suggested in the introduction, the availability of this metadata is a key to addressing the issues of data significance just mentioned, by means of inductive techniques. In the next section we compare this perspective with that of data quality research, as perceived by the data engineering community.

2.2 Current Data Quality research

The work mentioned in this section provides a background for our own work and, in some cases, ideas for future research. The main point that emerges from this survey is that the user-centred, metadata-based approach to information quality proposed in this thesis brings an element of novelty to the data quality landscape, and represents a complement to traditional approaches and techniques.

Indeed, when we turn to DQ research and practice for solutions that may be applicable to various stages of an experimental pipeline, we find that techniques in this area are only fit for the purpose to the extent that data management in e-science can be viewed as a special case of data management in relational, and possibly semi-structured, databases. Thus, the prevalent role of these techniques, we will conclude, is to enable data integration and analysis across multiple databases. This leaves the problem of defining domain-specific quality criteria open, and certainly does not suggest that quality management issues can be addressed as a knowledge management problem, as we propose.

Efforts in data quality research have been driven, historically, by the core data management issues faced by data-intensive business applications, namely data warehousing and analysis. In this respect, data providers view errors in the data essentially as complicating factors in solving traditional data engineering problems, i.e., data integration and fusion, and as noise that interferes with

Table 2.1: Quality issues in protein identification experiments

Common quality issues	*Artifact*	*Specific issues*	*Examples, techniques and references*
Quality of sample	Biological assay	biological variability, contamination control	sample contamination with, e.g., human proteins from the experimenter may obscure the results of downstream analysis
Process repeatability, results reproducibility	(General)	adequate process description	data modelling for capturing experiment design and execution results [FB02] the PEDRO data model [TPG+03] see also uniformity, below
	Raw data (mass spectra)	technical and biological variability	analysis of reproducibility [SZB+04] quantitative assessment of variability [CZS+04]
Data comparability	(General)	reproducibility across platforms, technologies, and laboratories	review: [HWSG02]
Significance of data	(General)	variability control	review on statistical and computational issues across all phases of data analysis ([LE05]) review on analysis of sensitivity [Smi02]
	Raw data (mass spectra)	sensitivity of spectra generation methods, dynamic range for relative protein abundance technical and biological variability limitations of technology for generating spectra	review on strategies to improve accuracy and sensitivity of PI, quantification of relative changes in protein abundance [RA05] studies on scoring models, database search algorithms, assessment of spectra quality prior to performing a search, analysis of variables that affect performance of DB search [SCY04] (review), [BGMY04] review on limitations of 2DE technology for low-abundance proteins [FGMA02]
Significance of data interpretation	Match results	significance and accuracy of match results, limitations of technology for accurate identification	definition [CMM03] and validation of scoring functions review on limitations of technology: [NA04] statistical models [NKKA03] studies on matching algorithms [SCY04] (review), [ZAS02]
Quality of reference data	(General)	redundancy of reference DB (same protein appears under different names and accession numbers in databases) accuracy, completeness, specificity, currency of reference databases	criteria for the selection of appropriate reference DB [TPG+03]: using a species-specific reference database will result in more real protein identifications than using a general reference database containing a large number of organisms; using the latter may result in a large number of false positives
Uniformity of representation, re-usability of results	Output data, publication	heterogeneity of presentation	need for representation standards [RS05] the PEDRO proteomics data model [TPG+03] guidelines for publication [CAB+04] standards for meta-data [HWSG02]

analytical results, for example census statistics. As a consequence, much of the research within the data management community has focused on error detection and data cleaning techniques that apply when data is stored in large databases using relational or, more recently, semi-structured data models.

A traditional example is the *record linkage* problem for data deduplication; this is a central problem in all data integration efforts, certainly also relevant to the life sciences data domain, where integration problems abound (a series of workshops on Data Integration in the Life Sciences (DILS) have been devoted to this topic alone [Rah04,LR05,LNE06,BT07]). Additional important problems include handling specific cases of completeness and consistency issues in databases. The reason why these approaches are very specific, and thus of limited use, is that they rely on the presence of suitable constraints on the schema. "Consistency", for example, is defined in the context of relational databases as the compliance of the data with certain functional dependencies, a rather narrow scope[1].

An important recent development, which is likely to affect our future research, is the recognition among the data engineering community that data errors often cannot be corrected, and therefore techniques are needed for dealing with unavoidable uncertainty in the data. Uncertain databases are emerging as a prolific research area, that we consider important for the management of quality knowledge: to the extent that predictive quality models are subject to approximations and uncertainty, the ability to manage probabilistic models within a database seems very relevant. We mention current results in this area in Section 2.2.5, and further discuss their implications for our future research in the last chapter of this thesis.

Although databases for the life sciences do exhibit some of the classical "dirty data" problems, and therefore to some extent they can benefit from these results, the main point we make here is that establishing correctness of rich life science data is contingent upon a good understanding of the domain, and requires access to suitable metadata to be used as predictors of quality. Neither of these two aspects, as we will see, is typical of current DQ literature and practice.

[1]For an in-depth account of current DQ issues, and of open research problems, one may consult [BS06], the most recent book to date on the topic.

2.2.1 Record linkage and data cleaning toolkits

The term *record linkage*, or equivalently, *record matching*, is used in the context of data heterogeneity resolution, where it is often the case that the same real-world entity, for example a person's name or address, is represented in slightly different ways, either because of a lexical misspelling or because of acceptable variations and synonyms. Record linkage refers to the problem of reconciling database records by "linking" those that represent the same entity despite these differences. It is worth emphasizing that the problem only refers to lexical and syntactic differences, without making attempts at considering data semantics.

The problem has a long history; it was originally addressed by Newcombe [NKAJ59] using Bayesian statistics, motivated by applications in population genetics [New67], and later formalized by Fellegi and Sunter [FS69]. Here we give a simple intuition of the Fellegi-Sunter model, as an example of a probabilistic classification of data that we are going to refer to later in the chapter. A full account of this and other models of record linkage can be found in a recent survey on data deduplication [EIV07].

In this problem we are given two tables A and B, and pairs $\langle \alpha, \beta \rangle$ ($\alpha \in A, \beta \in B$) are assigned to one of three classes `M`, `U`, and `R` (for "matched", "unmatched", and "reject", respectively). The third of these classes contains pairs for which the match status cannot be determined with sufficient statistical confidence. Subsequent research [VM04] extends the model to a general multi-way classification of the record pairs. To simplify the description, we only consider the simpler case of a 2-way classification `M`, `U`, and refer the reader to the original paper for more details. We are given pairs $\langle \alpha, \beta \rangle$ of records with the same structure, each containing n fields. The main idea is to perform a pairwise comparison of the n fields for each such pair, using a collection of n similarity functions that return either a binary value (i.e., agree/not agree) or a value in a known interval. This yields a comparison vector $\vec{x}$ of length n for each pair. We expect that pairs in `M` will exhibit high similarity on most of the n fields, while pairs in `U` will only show some isolated random high similarities. In general, we associate a probability distribution to each component of $\vec{x}$, and by extension, under the assumption of conditional independence among the components, a distribution to the whole $\vec{x}$.

To partition the space of all pairs into the two classes `M` and `U`, we use a decision

rule that relies on the comparison vectors:

$$\langle \alpha, \beta \rangle \in \mathtt{M} \text{ if } p(\mathtt{M} \mid \vec{x}) \geq p(\mathtt{U} \mid \vec{x}) \text{ , and } \langle \alpha, \beta \rangle \in \mathtt{U} \text{ otherwise}$$

In Bayesian statistics, $\vec{x}$ plays the role of an experimental observation, and $p(\mathtt{M} \mid \vec{x})$ and $p(\mathtt{U} \mid \vec{x})$ are posterior distributions that express the probabilities of M and U given observation $\vec{x}$. Thus, the rule says that a pair is a match if the probability of M is larger than the probability of U, given $\vec{x}$. Using Bayes' theorem, we can express this rule in terms of the prior distributions for U and M and of the *likelihood ratio* $l(\vec{x}) = \frac{p(x|M)}{p(x|U)}$, as follows:

$$\langle \alpha, \beta \rangle \in \mathtt{M} \text{ if } l(\vec{x}) \geq \frac{p(\mathtt{U})}{p(\mathtt{M})} \text{ , and } \langle \alpha, \beta \rangle \in \mathtt{U} \text{ otherwise}$$

With this formulation, the matching problem is reduced to the problem of estimating the likelihood ratio and the prior probabilities $p(\mathtt{M})$, $p(\mathtt{U})$. Under the conditional independence assumption, Fellegi and Sunter show that these probabilities can be estimated without the need for training data; in addition, the model can be extended to incorporate the third "reject" class alluded to above. Winkler [Win93] later extended the model by relaxing the assumption of conditional independence, showing that the well-known EM algorithm can be used to estimate $p(x \mid M)$ and $p(x \mid U)$. See also [Win02, Win06] for more recent surveys on these techniques.

One of the known weak spots of record linkage algorithms is that they depend on the choice of suitable similarity functions to compare record pairs: different functions exhibit different accuracy depending on the characteristics of the datasets. Thus, unsurprisingly, research has recently been focusing on the semi-automated selection of suitable similarity functions, for instance in [CCGK07] and in [BBS05, BKM06], where Bilenko *et al.* apply machine learning techniques to discover the functions.

Machine learning has also been proposed in commercial products that specialize in record matching, for example Choicemaker (`www.choicemaker.com`), where accurate matching is obtained by training a three-way classification model similar to that of Fellegi and Sunter, i.e., with match, non-match, and uncertain regions [BBWG03].

A number of toolkits, available either as research or commercial products, have

been built around record matching algorithms. These toolkits provide a variety of utilities that are necessary to prepare the data prior to performing record matching, i.e., to normalize the content of the record fields (see for example [BG05] for a survey of research-oriented tools), typically requiring human expertise for their manual configuration. Research products in this space include for instance Potter's wheel [RH01], Ajax [GFS+01], Tailor [EEV02] and the cleaning toolkit developed at Telcordia Technologies [CCG+00].

Finally, it is interesting to note that data quality services, centered on record matching, are now being added to commercial business intelligence solutions, a sign of the increasing awareness, at the enterprise level, of the importance of ensuring baseline data quality levels prior to performing sophisticated data analysis[2].

These toolkits can certainly play a role in e-science quality assessment, i.e., when data concerning traditional types (names, addresses, etc.) require integration. However, matching data structures that describe complex biological objects is likely to require ad hoc solutions, or a substantial effort for the configuration of the matching algorithm (i.e., for the selection and tuning of similarity functions, for example).

2.2.2 Completeness

Record matching, of course, is not the only type of data quality problem of interest. In the introduction we have briefly alluded to various quality dimensions, which collectively form a "data quality space". We analyse the completeness dimensions in this section, and the consistency dimension in the next. The lesson to learn here is that very specific definitions of these properties are required before we can see interesting research results for error detection and correction.

Completeness refers either to the presence of expected values in a record field (and is then also known as *density*), or to the coverage of an entire dataset relative to some other source (this simple distinction is articulated in detail in [SB04] for relational data). In the past, researchers have been interested mainly in the former type, for practical reasons; for example, it is common for census data to be incomplete, due to the voluntary nature of the information and the way it is

[2]Some of the solutions in this space include for example Microsoft SQL server Business Intelligence and SAS (`www.sas.com`), with its acquisition of DataFlux quality solutions (`www.dataflux.com`).

collected, and yet it is important to be able to perform meaningful analysis on census data. One approach to the problem has been to insert likely values into the fields, using statistical techniques of *data imputation* [Nau75, Red96]. This may not always be possible, due to the low confidence in the values, or even permissible, as is often the case for official government data in the public sector [MLV+03]. In these cases, resolving completeness issues has more to do with improving the data acquisition and editing process, than with any algorithmic solution.

The second type of completeness is perhaps more specifically relevant to life sciences data, where in many cases we see *secondary* databases that integrate information from a number of primary sources, adding value to them in various ways. For example, in the context of the Qurator study we have designed a secondary database for SNP data, i.e., data about simple genetic mutations, that draws its contents from various primary sources while offering added-value quality services to users of that data [MEH+07]. Here the problem is that, if the new database is missing some SNPs that biologists consider important, or even worse, has unpredictable gaps, the potential added value will not be delivered, because the services will simply not be trusted.

In this case we have a simple definition of completeness of the secondary source *relative* to one or more of the primary sources: for each specific query, the database completeness associated to the query is the amount of data that is returned by the query, expressed as a fraction of the amount of the data that would be obtained by issuing the equivalent query to the primary sources. Although simple to express, this measure is not easy for data providers who are responsible for the secondary database to compute, and even when available, is hardly ever disclosed to users at query time. Naumann *et al.* describe a possible approach to managing completeness [NFL04] in the context of information integration. They introduce the notion of usefulness of a source to answer a query, and show (i) how to measure completeness of a single and of integrated sources, and (ii) how to use measures of completeness for source selection and query planning.

We have recently addressed this problem in our own work [ME05] and have proposed a simple algorithm to enable providers to estimate database completeness. The main original idea is to keep track of the incremental updates to the secondary database in an intensional way, i.e., as a set of update queries, and to then compare such query history with the conditions found in user queries as they

are submitted; this enables a provider to estimate the coverage on a per-query basis.

Note that we have presented coverage as a provider's problem, in that it requires knowledge of the provider's data architecture. In some cases, however, users may try to assess coverage without the help of the provider, i.e., by performing queries to multiple data sources (i.e., to both a secondary and a primary database), when those primary sources are available to them. In the case of the SNP database mentioned earlier, for example, this is an option because all sources are public. This approach, however, requires a significant deployment effort by the users in terms of data architecture, which may not be practical.

2.2.3 Consistency

Consistency refers, in general, to the compliance of data with certain constraints. A simple case is *syntactic* consistency, where grammar rules are used to define the acceptable formats of some data type, for example addresses or phone numbers. In this case, error detection can be done using a parser, and in some cases compliance can be restored by means of some domain-specific rules. Database schema constraints, for example referential integrity in relational databases, are examples of *semantic* consistency. While the DBMS is sometimes able to detect these violations, this is not always the case. Some of reasons, listed for example in [ABC+03], include the integration of autonomous data sources, where constraints may hold on individual sources but not on the integrated database, and legacy DBMSs that do not perform integrity checking. In this case, record matching is the main tool used in practice to restore consistency, i.e., to determine the most likely primary key record for a foreign key that violates an integrity constraint, as a particular case of record matching in two arbitrary datasets. Note that, when we extend this notion of consistency to multiple independent databases, and to more general consistency rules, record matching may no longer be adequate, or sufficient. Consider for instance the case of two protein records in two databases, where the proteins have the same name but different descriptions. Reconciling the two records typically requires the use of domain knowledge in combination with rules to determine whether the two descriptions are compatible with each other, or whether it is more likely that one of the protein IDs is wrong.

As we can see, these notions of consistency are fairly restrictive. We can

sometimes write slightly more general, domain-specific rules, to express for example the fact that a certain area code must be consistent with the name of the city: having a US address record with area code 07950[3] and city equal to San Francisco, for example, is a semantic inconsistency. In a recent effort to address the problem of detecting this type of errors in a principled way, Bohannon *et al.* [BFG+07] propose a definition of consistency that relies upon an extended version of data functional dependencies (FD). Their main observation is that an FD such as "area code → city", defined at the schema level, is insufficient to detect an inconsistency like the one above, because the pair ⟨07950, San Francisco⟩ is not a violation unless the table contains another record of the form ⟨07950, X⟩ with X different from San Francisco. However, by adding a new, instance-level constraint to the FD, i.e., "07950 → Morristown", we capture a more specific semantic rule. Based on this observation, the authors develop a theory of *Conditional Functional Dependencies* (CFD) and show that, in some cases, they can be used to automatically restore consistency to the database [CFG+07]. They also note that, in general, it is possible to propose several repairs to the database, i.e., alterations that restore consistency, and therefore that some criteria are required to select repairs. This leads to a formal notion of *accuracy*, as a measure of adequacy of a particular repair. In the model proposed in [CFG+07], for example, the repairs are chosen to minimize the number of changes to the database.

In summary, current research on inconsistency detection and correction, although formally interesting, appears to be confined to the rather narrow scope of functional constraints on relational data. By contrast, expressing the consistency of biological data requires a broader approach and more complex rules, for example to establish the plausibility of data produced by an experiment, given the description of the experiment and its intermediate results. In this case, plausibility follows from consistency: an experimental pipeline where noise is introduced at some stage, for example, may disrupt consistency by corrupting the results produced downstream from that stage.

2.2.4 Quality-based source selection

Record matching is often used as a preliminary phase for the integration of multiple information services with overlaps among their datasets. After a reconciliation step, it becomes possible for a distributed query processor to choose a

[3]The area code for Morristown, NJ.

specific data source based on quality considerations, in addition to data availability. Several researchers have recently explored this idea. Martinez and Hammer [MH05] propose to associate a predefined set of quality measures to biological data sources, and to expose them as part of an integrated data model. Naumann *et al.* [NUJ99,Nau02b,Nau02a] go one step further, by proposing a quality-driven query processing architecture where multiple quality dimensions are taken into account while processing a query over heterogeneous information systems. The approach is based on the assumption that data providers can compute quality values for their data, at various levels of granularity, and can expose them to the query planner as part of the data model. A similar approach is found in [MAA04], where the use of utility functions is advocated for the purpose of resolving inconsistencies across multiple databases. We recognize a similar line of reasoning in the DaQuincis data broker for cooperative information systems [SVM+04]; here the authors make the assumption that the quality of semi-structured data, and specifically of XML documents, can be fully described by associating a "quality element" to each data element. From here, the authors proceed to describe a data broker that makes use of the quality metadata to perform source selection at query time. Finally, in our own work [ABBMed] we have addressed the related problem of decomposing a query over multiple data sources, when a complete and accurate quality characterization is available and the cost of obtaining the data from each of the sources is known. We have proposed a brokering algorithm that computes a cost-optimal complete answer to the query, by assembling partial answers from the sources under quality constraints.

As we have noted in this brief survey, this type of research (including our own) relies on the assumption that the necessary quality metadata is available: however it is not clear how it is generated, when, or by whom. The new approach proposed in this thesis provides a partial answer to these questions: quality metadata reflects the users' specifications, which are given in the form of quality functions. In this respect, our user-centred approach to quality is a complement to the work just described, providing an important missing element for its actual applicability (indeed, to the best of our knowledge, none of the approaches just mentioned is currently used in practice).

2.2.5 Living with incomplete and uncertain data

Researchers have been aware of the consequences of incompleteness and inconsistency in databases for a long time, initially dealing simultaneously with issues of completeness, i.e., null values, and of constraint violations [Mor90]. Recent work by Arenas *et al.* [ABC99, ABC$^+$03] explores the extent to which one can rely on inconsistent databases to obtain consistent results to queries. Here the idea is that when constraints are expressed using a first order language, a repair is always available, and one can formally define the notion of a consistent answer on a repaired database, and provide methods for computing such answer.

In addition to exploring the feasibility of automatically repairing an inconsistent database, more recent research stems from the assumption that various types of inconsistency in the data are unavoidable (a recent Dagstuhl seminar, for example, has been devoted to issues of tolerance to inconsistencies [BHS05b]). A closely related problem is that of uncertain data: here attributes may be set-valued, and only some combinations of these values, over a number of attributes, represent a consistent database state.

One way to address uncertainty is to use *probabilistic databases*. As Dalvi and Suciu put it in [DS07]: "We know well how to manage data that is deterministic. Databases were invented to support applications like banking, payroll, accounting, inventory, all of which require a precise semantics of the data. In the future we need to learn how to manage data that is imprecise, or uncertain, and that contains an explicit representation of the uncertainty.". Their paper presents a general paradigm for data with uncertainties. It follows previous work from the same authors [DS04], which describes a new type of query semantics that is based on a probabilistic model of the data. Probabilities are internal measures of imprecision of the data; while users formulate queries using a standard query language, the system computes the answers, and for each answer it computes an output probability score representing its confidence in the answer. These scores can then be used to rank the answers. Among the types of applications that call for a probabilistic framework, the authors mention the growing importance of web-scale information retrieval systems, as well as, unsurprisingly, approximate record matching in the context of information integration.

Another example of current research in this area is the Trio system [Wid05, BSHW06], developed at Stanford and recently released to the community. Trio is based on ULDB, an extension to a relational database designed to manage

uncertain data along with its *lineage*. When data is uncertain, the database must represent multiple possible instances, each corresponding to one of the possible database states. Lineage describes the derivation through which each of these possible instances of the data is obtained[4]. The idea pursued in ULDB is that lineage can be used for understanding and resolving uncertainty. Intuitively, when users issue queries to an uncertain database, the results will be subject to uncertainty too. The authors show that lineage is an intuitive mechanism for helping determine which output is correct in the presence of uncertainty, and develop a data architecture that supports practical uncertainty management.

Antova *et al.* also address the problem of efficiently representing uncertain data, i.e., multiple possible states of the database [AKO07a, AKO07c]. Multiple worlds originate when each tuple field may have more than one value, i.e., its actual value is uncertain but restricted to a finite set. Choosing one value for each uncertain tuple leads to an exponential number of worlds. The idea pursued in this work is based on the notion of *world-set decompositions* (WSD), a novel and compact way to represent the sets of all possible combinations, that makes it possible to compute queries efficiently on them. In practice, WSDs are decompositions of a single world-set relation into several new relations, such that their cross product is again the world-set relation. The authors have implemented a DBMS based on this idea, called *MayBMS* [AKO07b].

In conclusion, we consider both probabilistic models and uncertainty management to be potentially relevant to our own work. Although they do not directly help in the task of quality assessment, their relevance is in providing a representation of uncertain quality metadata. Having chosen to define quality using predictive models on the data seems to lead naturally to a notion of quality that is intrinsically subject to uncertainty and, in some cases, may be described by a known probability distribution. We can therefore expect that managing the corpus of quality metadata, i.e., metadata that describes the quality of data, may require a probabilistic data model and corresponding management architecture.

[4]The term *data provenance* could presumably be used as an alternative to lineage, however the authors never make an explicit distinction between lineage and provenance.

2.3 Information Quality as data classification

As we have seen in the preceding survey, data management experts and statisticians view errors in the data essentially as complicating factors in solving traditional data engineering problems, primarily data integration and fusion. Their efforts are driven mostly by the core data issues faced by data-intensive business applications, i.e., large scale data warehousing and analysis; and the technical solutions tend to be rooted in database technology and independent of the data domain. In other words, the research is driven more by the structure of the data than by its content, the assumption being that, for the purpose of error detection and correction, one can only exploit the data semantics that can be found in the database itself.

This is in contrast to the e-science context, where rich data semantics, i.e., metadata, is often available either directly from the data processing environment, or in the form of independent annotations to the data —it is this surrounding context that makes data into information. We are specifically interested in two main forms of such metadata, namely the detailed record of an experiment, often collected in order to ensure repeatability and verifiability by third parties; and quality control metadata, computed by data analysis tools as a measure of the reliability of their output. A typical example of the latter (cited again in the next chapter) is that of BLAST, a popular sequence alignment algorithm, where a measure of confidence is associated to each alignment.

The documentary nature of such metadata makes it a valuable asset for quality assessment purposes. This seems to fit well with the scientists' approach to quality assessment, where the goal is ultimately to determine the "fitness of data for use" in the scientific context. Viewed in this perspective, the quality assessment task translates into a decision process, where the reliability of scientific data, especially that produced by external, sometimes uncertified sources, must often be assessed using indirect forms of empirical evidence, and is therefore subject to uncertainty.

Our model for information quality, presented in the rest of this chapter, is based on an evidence-based decision process. The evidence consists of a collection of properties that describe some attributes of the data that the scientists can observe and measure. The scientists' task is to select a combination of these attributes, and to develop a model (normally using statistical techniques) to estimate the likelihood of errors in the data, given the attribute values. As an example, consider the problem of assessing whether a collection of microarray

datasets produced by a certain research group can be safely used by another, and assume that the available evidence consists of some provenance information, such as the historical error rates exhibited by the group, as well as the number and impact factor of their publications. The user's problem is to determine whether these two characteristics can be used to predict the absence of errors in the datasets, and if so, to measure the accuracy of the prediction (i.e., how often the prediction is correct when observed on datasets of known quality). We formulate this in abstract terms as the problem of defining a decision model, known as a *classifier*, that is able to assign a data element, in this case a microarray dataset, to one of two classes, "accept" or "reject", based on a set of underlying data attributes.

In the next section we begin by introducing a simple, two-way, i.e., accept/reject classification model, and then extend it to multiple classes.

2.3.1 Simple quality classes

Data classification problems, not at all trivial, have been studied extensively in the context of statistics and data mining (referred to in the following as "Knowledge Discovery", or KD). In particular, machine learning algorithms have been developed to automatically or semi-automatically generate classification models in an inductive fashion, starting from training datasets, i.e., data whose faulty condition is known. Various types of classification models are available, both from the literature and as practical implementations [Mit97, WF05]. Some of these, such as the naive Bayesian classifiers, provide users with a measure of confidence in the classification; for each class and data element, they compute the probability that the element belongs to the class. In some circumstances, this information can be used to assess the risk associated with poor quality, namely when the cost of using faulty data can be quantified (for example, some authors have estimated the cost of using the results of a faulty microarray experiment [BEPG$^+$05]). Other classification models, known as *deterministic*, assign the data to the most likely class, but they do not provide an explicit measure of probability of class membership; decision trees are a well-known example of this type of classifier. Note that the outcome of general classifiers is not limited to two classes; the theory applies to any predefined, finite set of classes.

The idea of using predictive classifiers, regardless of their specific features, is at the cornerstone of our IQ framework. By using a classification model, we

translate the scientist's problem of providing a precise, yet personal definition of "quality of information", into a conventional knowledge discovery problem. Specifically, the quality knowledge engineer is faced with the following tasks: (i) selecting data attributes to be used as evidence; (ii) defining a classification model based on those attributes, and (iii) assessing the performance of the model. A classifier's performance is a measure of its ability to perform correct classifications on known test datasets, and is used as an estimate of the expected number of errors incurred in the classification of a previously unknown dataset. Broadly speaking, this measure involves observing the number of false positives and false negatives produced by the classifier on the training set data.

These tasks embody the "inner loop" steps of the IQ lifecycle; they are all part of the standard KD framework used in data mining, and are well supported by current technology: by formulating quality definitions as a type of data classification, we can leverage the large body of theoretical and practical work that is available from the areas of statistics, knowledge discovery, and machine learning. Figure 2.3 illustrates our operational definition of IQ as a process based on a simple classification model, and introduces our notation for the IQ framework elements.

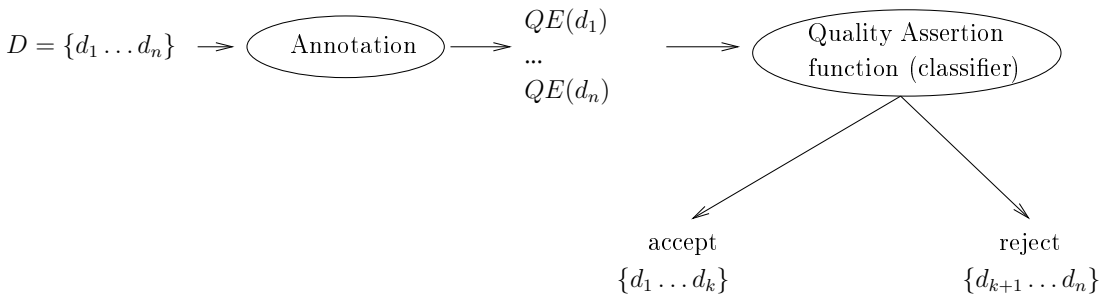


Figure 2.3: Basic acceptability as a binary classification model

Given a dataset D that is to be assessed for quality, the first step is to compute the values for the evidence attributes used in classification, using one or more *annotation functions*. The term "annotation" emphasizes that the role of these functions to associate "quality metadata" with the input dataset. In the microarray dataset example, these functions are responsible for retrieving the historical error rates on microarray experiments for a given research group, as well as the list and impact factors of the group's publications. It is important to note that we view annotation functions as black boxes; while a variety of processes can be employed to obtain this information, their exact nature is not exposed to

the framework. We use the term *Quality Evidence* to denote the result of quality annotation, to emphasize the role of these attributes. Note also that, in general, while each evidence element represents "raw" quality metadata, when taken in isolation the elements are not sufficient to provide a significant indication of quality. A value for Hit Ratio, for example, only becomes a useful quality predictor when combined with other types of evidence as part of a score model.

The second step in the IQ process is to apply the classification model to the quality evidence. We have coined the term *Quality Assertion* for these classifiers, again in an effort to emphasize their role in the process, i.e., to deliver formal, and yet user-defined assertions on quality properties for the data. The effect of classification is a binary partition of D, as shown in the figure.

One may observe that, in some cases, a continuous score, rather than a discete classification, is a more suitable form of quality knowledge. Of course this is not a limitation, since a score can be mapped to a classification simply by selecting a threshold on the score interval (and, by extension, by selecting multiple thresholds to obtain a multi-way classification). The issue of threshold selection, however, raises an important point regarding the distinction between subjective and objective elements of a quality model: let us discuss this in the context of our proteomics example.

First of all, note that the structure of the example fits the the IQ framework: the predictor variables Hit Ratio, ELDP, number of peptides, etc. form a corpus of quality evidence, and the annotation functions compute the evidence values using the output of the protein identification algorithm. The Quality Assertion in this case is represented by the score model. As we know from Section 1.3, Stead *et al.* in their paper [SPB06] propose a new type of objective score model and experimentally demonstrate its performance; they do not, however, give any explicit indications regarding threshold setting, presumably because this pertains to the subjective area of personal risk assessment: a low threshold would accept most of the dataset but increase the risk of introducing false positives, while a high threshold may be too selective. This does not mean, however, that the threshold selection is arbitrary: in fact, the very same technique used to demonstrate the performance of the score on tests data can be used as a guideline. Let us look at this technique in more detail.

In [SPB06] the performance of the score model is described using a standard statistical tool known as a ROC curve [Faw06]. Given a ranked test dataset, the

k-th point on the ROC curve is obtained by plotting the rate of false positives against the ratio of true positives, relative to the top k data elements. A point (x, y) with x close to 0 and y close to 1 for a given k shows good performance, indicating that many true positives are mixed with few false positives among the top k elements. Thus, the curve effectively represents all possible trade-offs between benefits (true positives) and costs (false positives) for all thresholds k. It is important to note that the curves are computed using known test datasets, therefore the actual performance may differ on the dataset under study. Nevertheless, this tool can be used to make informed decision of a suitable threshold for data acceptability. As suggested earlier, the threshold setting reflects (an estimate of) a scientist's propensity to accept a relatively high cost for a corresponding benefit (a large accepted dataset). Note that the threshold value could, in principle, be optimized given some cost function, for instance one that associates different weights to false positive and false negative errors. However our IQ framework does not, at the moment, include features to support such an optimization step. The important point here is that with this framework we do not eliminate subjectivity, but rather we find a specific *locus* for it in the process.

2.3.2 Multi-way quality classification

A binary classifier, which can only predict "accept" or "reject", is of limited use. It does not allow us, for instance, to express decision criteria like those involved in the Fellegi-Sunter model for record linkage (Section 2.2.1), which includes a third class, namely the uncertainty region[5]. As another natural example of three-way classification, let us consider the discretization of the protein hit score model again. This time, however, we automatically set two thresholds using the sample mean $\overline{x}$ and standard deviation s of the scores distribution as a guideline, for instance by setting the classes to be $[m, \overline{x} - s]$, $(\overline{x} - s, \overline{x} + s)$, $[\overline{x} + s, M]$, where m and M are the lower and upper bounds of the scores, respectively. We take the resulting three classes to mean "low", "close-to-average", and "high" quality.

To accommodate these additional quality scenarios, we need to allow for multi-way classifications. We use the term *quality classes* to denote any predefined, finite set of class labels used to characterize quality properties of the data, and we say that data elements that are in the same class are *quality-equivalent*. Note

[5]Record linkage can be viewed as a classification problem where the input dataset consists of all pairs of records from an input collection that contains duplicates.

however that, with this generalization, the meaning of a quality class is no longer clear. With our binary classes "accept" and "reject", the intended meaning of the class was implicit, and could be left to the intuition, while with three classes or more this is no longer the case. In record matching, for example, we must explicitly state that class `R` means "no linkage status can be established for record pairs in this class". Note also that we have so far ignored a natural notion of ordering among the classes, i.e., the fact that some classes should be preferred to others. For instance, when the classes are obtained by discretizing a continuous interval, as in the example above, the order on the score interval maps to a total order on the corresponding set of classes: "high" is better than "low". We address both problems in the rest of this section.

Firstly, we introduce a total order relation among quality classes, defined by the quality knowledge engineer, as a simple and standard way to express preferences among the data. In addition to being a natural consequence of using score models to express quality, note that this is also consistent with commonly accepted models of consumer behaviour used in microeconomics [Var96]. Here the decision processes of consumers are modelled by means of a set S, used to represent the available choices (e.g. of goods to buy), and a *partial* order $\leq$ on S, called a *preference relation* on S. This relation is used to express the relative appeal of two elements of S to the consumer. In this model, two elements a and b such that $a \leq b$ and $b \leq a$ are said to be *indifferent* to each other. Since the indifference relation is an equivalence, the partial order $\leq$ induces a partition on S, so that for any equivalence class in the partition, the consumer has no preference between any two elements of the class.

We can make a parallel with our quality setting, as follows. Rather than having an explicit partial order among a predefined set S of goods, we provide an intensional definition of the quality classes, that can be applied to any input dataset S – this is our classification procedure. Clearly, this procedure along with a total order on the classes induces a partial order on S: two elements of S that are in different classes are ordered according to the order of the classes, while two elements in the same quality class are not ordered. Conversely, given a fixed set S and a partial order $\leq$ on S, it is natural to define a set of classes and a classification procedure that is consistent with the order.

Regarding the second problem, i.e., the intrinsic meaning of multiple quality classes, we note that one can easily annotate a class with an informal description

of its intended meaning. In fact, we could also attempt to associate some type of formal descriptions to the classes. Regardless of the description, however, the important point is that the same quality classes may be *used in different ways by different users*. One user may, for example, ignore class `R` in the record matching model, while another user may associate a particular process to it, i.e., to perform clerical review of the uncertain record matches. In our user-centric quality model, we are primarily interested in capturing how a quality class is going to be used in the context of a user process.

In this spirit, we propose that the meaning assigned to a quality class by the quality knowledge engineer be limited to the class name, possibly an informal description, and an order relation. At the same time, we propose to capture the users' perspective on the same quality classes, by mapping quality classes to a new space of *actions*. Actions represent user processes that can be carried out on the data using the facilities available from the processing environment. For example, suppose that a user defines a workflow process that, among other tasks, performs record linkage using the Quality Assertion function made available by some knowledge engineer. This user is aware of the three-way classification, and decides to interpret it as follows: filter out the `U` pairs, allow the `M` pairs to proceed to the next workflow task, and send the `R` pairs to a dedicated task that will perform some special processing. We capture this user intention using a set $A = \{\texttt{filter}, \texttt{allow}, \texttt{analyse}\}$, representing an abstraction of the three user actions, and by establishing a mapping from the quality classes to the actions: $M(C, A) = \{\texttt{M} \rightarrow \texttt{allow}, \texttt{U} \rightarrow \texttt{filter}, \texttt{R} \rightarrow \texttt{analyse}\}$. This scenario is depicted in Figure 2.4.

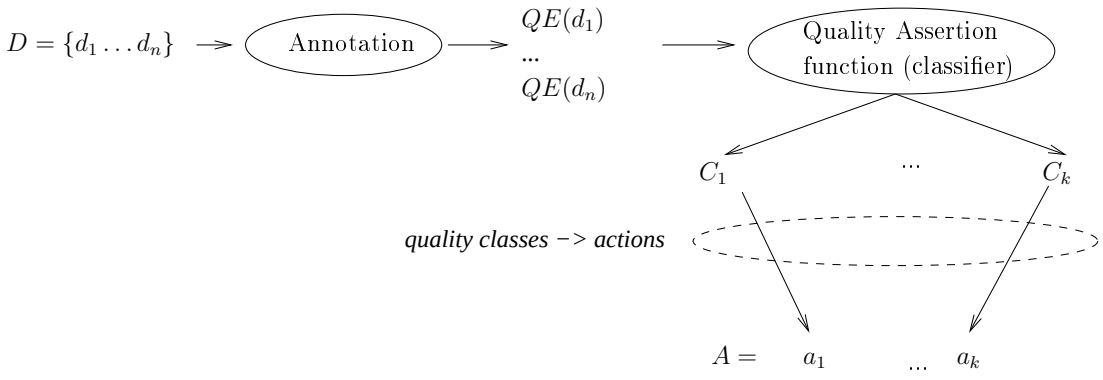


Figure 2.4: Multi-way classification with explicit quality actions

Our goal here is to separate the definition of quality classes, defined independently of data usage and embodied by classification models (with the ordering), from the space of quality-aware data processing, where quality classifications are used. The assignment of quality classes to actions is the bridge between the two spaces. This difference between definition and use is central to our model: while we can often give an interpretation of quality classification that is independent of any application (a record pair is, after all, a match or a non-match regardless of the use that we make of it), it is the mapping to actions that reflects the user perception of the importance of such quality classification on data processing.

To appreciate this point, consider a second, optimistic user who decides to retain `R` record pairs (which may be false positives); this user's preference would be modelled simply by defining $M(C, A) = \{\texttt{M} \rightarrow \texttt{allow}, \texttt{U} \rightarrow \texttt{filter}, \texttt{R} \rightarrow \texttt{allow}\}$.

It is also important to note that the set of available actions is constrained by the operating environment, in addition to the user preferences. We can easily see this by placing the same record linkage classification in the context of a visual data presentation environment, where the universe of actions now includes hiding or highlighting data elements, or colouring them in different ways. The user's interpretation is now given as a mapping of our three classes to these new actions.

One may observe that in this framework actions are defined purely as a set of labels, much like quality classes. The difference, however, is that the action labels have an operational meaning in the context of the user data processing environment. For example, `analyse` can be made to correspond to a workflow processor with an input consisting of the data found in one quality class (namely `R`, according to our example mapping). We will return to this important point when we discuss the translation of user quality preferences to executable quality processes, in Chapter 5.

2.3.3 Multiple classifications and condition-action mappings

So far, the framework we have proposed does not prescribe any particular mechanism or language to specify the mapping between classes and actions. In

fact, we have made the implicit assumption that each quality class is associated to exactly one action. Two simple observations suggest that this assumption may be too restrictive in practice, and that an explicit mapping mechanism from classes to actions is needed. Firstly, consider an application that is only interested in the positive matches produced by a record linkage algorithm. For this application, it makes sense to collapse the `U` and `R` classes into one, by associating the same action (eg. `discard`) to both. While this can be accomplished by specifying a set of condition-action pairs of the form $M(C, A) = \{\texttt{M} \rightarrow \texttt{allow}, \texttt{U} \rightarrow \texttt{discard}, \texttt{R} \rightarrow \texttt{discard}\}$, we could also write, using a set expression, $M(C, A) = \{\texttt{M} \rightarrow \texttt{allow}, \texttt{U} \cup \texttt{R} \rightarrow \texttt{discard}\}$.

The second observation is that, in some cases, it would be natural to use multiple classifiers in order to express different aspects of quality. We could then combine their outcome into a single quality classification, and then associate a quality action to the result. Consider for example a dataset of proteins, where functional annotation is performed either by expert curators, or is predicted algorithmically. Typically, such a dataset presents a trade-off between accuracy and timeliness of the annotations – this is because human annotations are more accurate but take longer to perform the task than automated annotations, that are subject to errors. In this situation we could have two separate quality classifications, one for timeliness and one for accuracy; the final quality classification would then be a function of the two (based for instance on a weighted average of the scores).

We extend the framework defined so far with condition-action pairs to account for both scenarios. Conditions on classes are set expressions that may involves the usual union, difference and intersection operators. Given two classifications $\mathcal{C}_1 = \{C_{11}, C_{12}, C_{13}\}$ and $\mathcal{C}_2 = \{C_{21}, C_{22}\}$, for example, we can write the expressions

$$\begin{aligned} e_1 = &\quad (C_{11} \cup C_{12}) \cap C_{21} \\ e_2 = &\quad\quad C_{13} \cap C_{22} \end{aligned}$$

With these, we define mappings from quality classes to actions as a set of condition-actions pairs: $M(C, A) = \{e_1 \ldots e_k\}$ with $k \leq |A|$. Figure 2.5 shows this for a configuration that includes two independent classifiers and the two expressions e_1 and e_2 above.

Note that the conditions effectively define a secondary classification on the

$\mathcal{C}_1 = \{C_{11} \ldots C_{1n}\}$
$\mathcal{C}_2 = \{C_{21} \ldots C_{2m}\}$

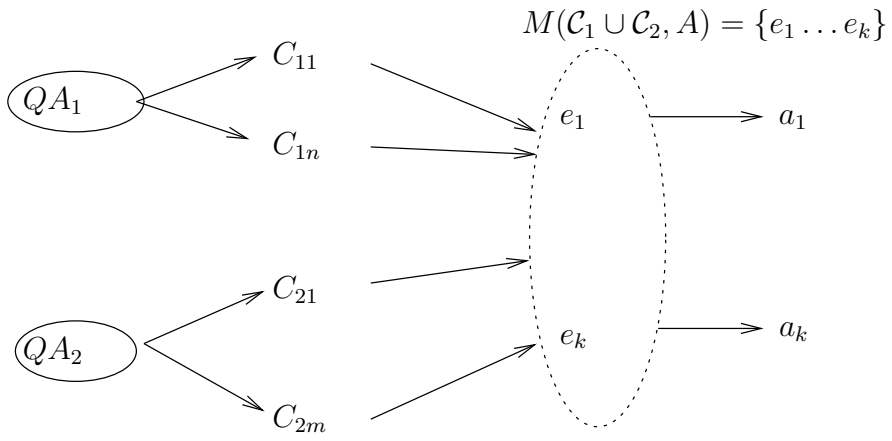


Figure 2.5: Multiple classifiers and explicit class-to-actions mapping

data. However, we can no longer assume that these new "quality classes" enjoy the total ordering property that we have postulated in our earlier definition[6]. Thus, these new classes only have a meaning in the context of condition-action mappings, rather than as quality classes of their own. Since, however, these new classes are defined not by the quality knowledge engineer, but by the users, this does not pose any problem in practice.

Note also that this definition of mappings also accommodates the task of deriving a classification from a score model, in a natural way. Specifically, since we can use $n-1$ thresholds to define a new n-way classification, it makes sense to allow a mix of conditional expressions that include numerical inequalities on the score value, with set operators. More importantly, one may use the mappings to define regular regions in a multi-dimensional space defined by more than one score model. An example is shown in Figure 2.6, with the expressions:

$$\begin{aligned} e_1 = &\quad (s_1 \geq t_{12}) \wedge (s_2 \geq t_{22}), \\ e_2 = &\quad (t_{11} \leq s_1 \leq t_{12}) \wedge (t_{21} \leq s_2 \leq t_{22}), \\ e_3 = &\quad (s_1 \geq t_{12}) \wedge (t_{21} \leq s_2 \leq t_{22}) \end{aligned}$$

In summary, by combining score intervals and discrete classification one can

[6]As a simple example, suppose that $C_1 \leq C_2 \leq C_3$. Then $C_{11} = C_1 \cup C_3$ and C_2 are not ordered.

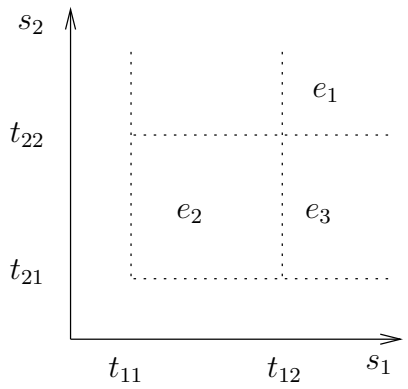


Figure 2.6: Expressions and corresponding regions in a bi-dimensional score space

write conditions like the following:

$$e = (C_{11} \cup C_{12}) \cap C_{21} \cap ((s_1 \geq t_{12}) \cap (t_{21} \leq s_2 \leq t_{22}))$$

which involve two classifiers (Figure 2.5) and two score models (Figure 2.6). We are going to use type of expressions as part of our language for Quality Views, in Chapter 4.

2.4 Discussion: the IQ lifecycle, refined

We use the IQ lifecycle introduced in Section 1.4.2, and illustrated in Figure 1.1, to summarize the elements of the IQ paradigm and to preview some of the elements that will be introduced in the following chapters. A refined version of the lifecycle diagram that takes into account our definitions appears in Figure 2.7. The role of the quality knowledge engineer in the inner loop is to contribute new Quality Assertion functions, as well as corresponding Annotation functions to provide the required input Quality Evidence. This results in a library of quality function implementations.

In the outer loop, users of quality knowledge select Quality Assertion functions that are appropriate for their data types, and compose them into more complex processes, as discussed in Chapter 4. These processes may include multiple classification models, and condition-action mappings as described earlier. As a result of executing one of these processes, therefore, the data is partitioned into quality classes and the associated actions are performed on each class. Users at this point may analyse the results, to determine whether there is a need to perform a new

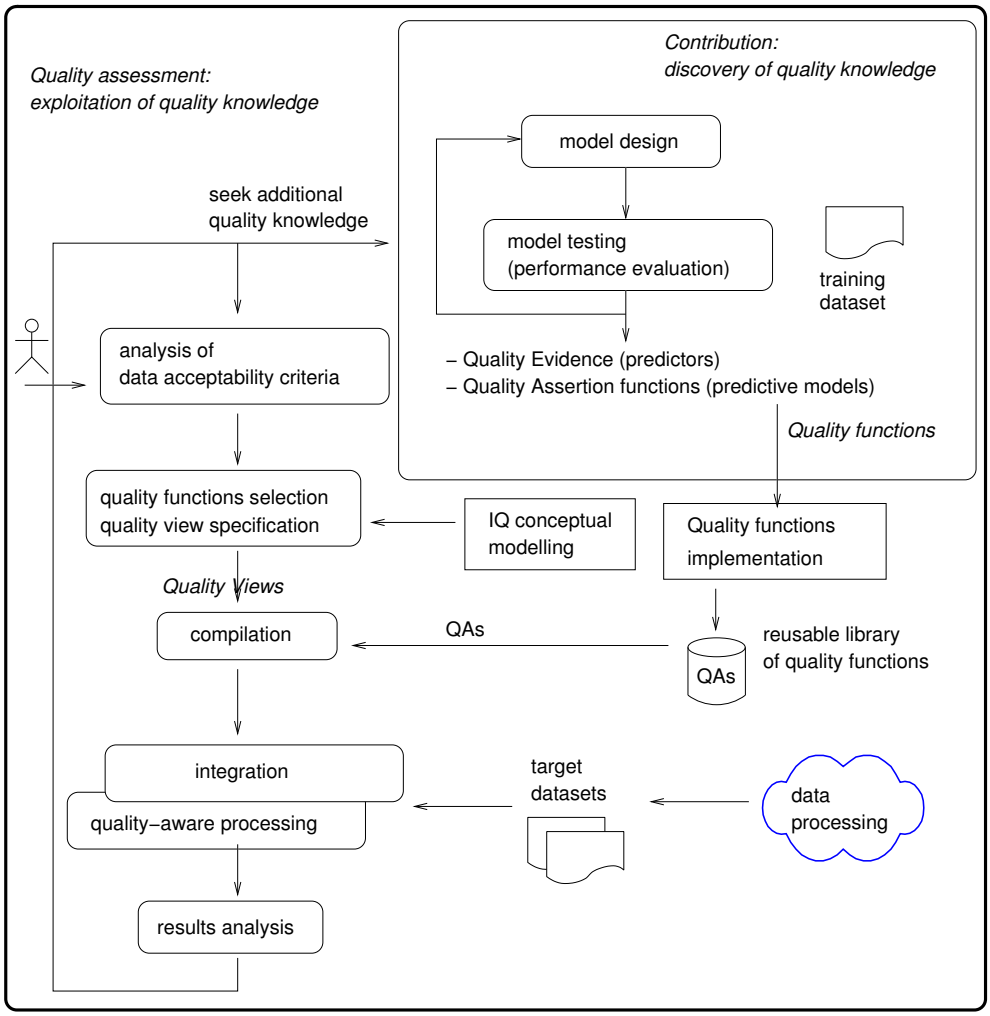


Figure 2.7: The Information Quality assessment lifecycle

iteration around the loop, or to seek additional quality knowledge.

We can view the model elements in Figure 2.8, where we make a further distinction between long-lived "quality commodities", that are the result of a KD engineering effort, and the definition of acceptability criteria as actions on the quality classes, that reflect the preferences of the different users of those quality functions.

Both these aspects involve iteration and experimentation; the Qurator workbench described in the rest of this thesis is designed to reduce the human effort involved in their definition. In particular, the next chapter presents a conceptual model that formalizes the framework elements, while Chapters 4 and 5 describe a process model for the execution of the steps above, and its realization as a

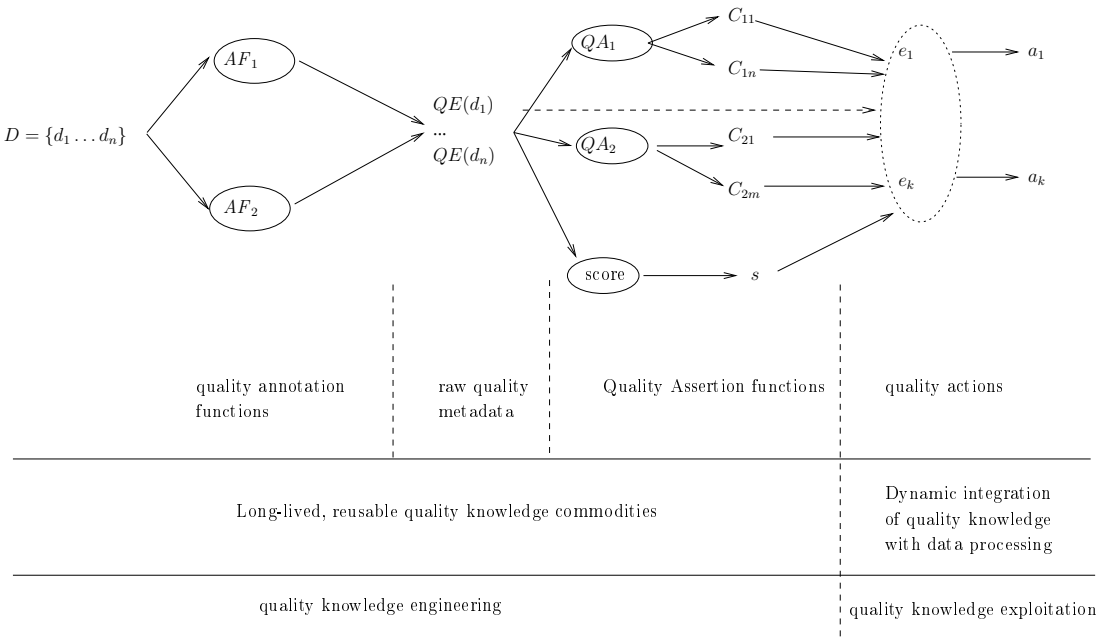


Figure 2.8: Summary of quality processing and modelling options

workflow, respectively. Finally, in Chapter 6 we will discuss our the Qurator workbench supports the tasks in the outer loop of this lifeycle.

At this early stage, meanwhile, we can justify the framework design on the basis of the two arguments of flexibility and reusability of its elements. Regarding flexibility, we observe that this structure offers scientists a spectrum of quality modelling options, ranging from the minimal to the fully personalized, so that some trade-offs between complexity and accuracy are available. While sophisticated users may define all of the elements above, the model also offers a reasonable default behaviour (for example, the default set {"accept", "reject"} of quality classes, with corresponding standard interpretation) to less demanding users, who can test simple quality hypotheses with limited effort.

The second argument concerns the reuse of other scientists' definitions: hopefully, the classifiers created by the quality knowledge engineer to operate on certain data types, along with the corresponding annotation functions, can be made available to other members of their scientific community who deal with similar datasets. The reuse argument – of abstract quality functions, of their implementations, and of quality processes that make use of those functions, is central to the thesis and will be made precise in the following chapters.

To conclude, we summarize the distinctive features of our approach that make

it stand out from those surveyed earlier, namely that (i) it is designed to capture the users' own definitions of quality properties, and is therefore inherently extensible, and (ii) it enforces a clear separation between objective quality assessment and the the subjective importance associated to quality in the context of data processing. As we have seen, we can use the model to capture the semantics of traditional quality assessment techniques, such as record matching.

Chapter 3

Semantic Modelling of Information Quality concepts

In the previous chapter we have proposed a simple paradigm to represent quality knowledge in terms of data classification. Here we address the problem of how this knowledge can be described to scientists in such a way that they can use it to specify quality processes. We approach the problem by proposing an original conceptual model for Information Quality. The IQ model includes an extensible classification of the types of data for which quality assessments can be made, and an extensible classification of quality indicators, as well as of quality functions that are computed using the indicators. It also includes a taxonomy of abstract quality properties, such as those discussed earlier, i.e., *correctness*, *accuracy*, etc., as well as a number of additional concepts that will be presented in detail below.

We begin by arguing that a semantic framework, i.e., an ontology, is appropriate for the IQ model (Section 3.1). We then present the ontology in detail in Section 3.2, emphasizing our axiomatic approach to the definition of quality concepts. Finally, in Section 3.3 we show that this approach is valuable as a way to enforce consistency constraints on the IQ concepts and their relationships, and thus provides a formal foundation for user-defined extensions. In the following chapters we are going to build upon the IQ model, to provide effective tools in support of the various steps of the IQ lifecycle.

3.1 Rationale for semantic modelling

The choice of the modelling framework for the conceptual IQ model, articulated also in [PME+06], has been driven mainly by practical requirements. Two key requirements are extensibility, i.e., the ability for domain experts to introduce new concepts that represent personal definitions of quality features, and the capability to share those definitions with other members of a community. These two elements provide the foundation for an incrementally growing body of quality knowledge. On the other hand, if we are to build upon the IQ model to offer effective tools to users, it is clear that this evolution should progress in some principled way. Thus, a third requirement is the support for some kind of formal consistency for instances of the model, and its enforcement using automated techniques.

While one may argue in favour of traditional frameworks like ER and UML, we have instead opted to explore the potential of semantic modelling, mainly for its promise of flexibility, as well as support for logic-based specification and automated reasoning. As a practical consideration, the increasing adoption of ontologies as a standard way of modelling complex domains means that we can leverage a substantial body of research knowledge and practical expertise in this area. Furthermore, the standardization of the OWL DL language for the Semantic Web has spurred the development of a variety of tools for ontology design and specification (among the most popular are Protege from Stanford (`protege.stanford.edu`) and Swoop, formerly developed at University of Maryland and now an open source project (`code.google.com/p/swoop`), as well as for automated reasoning (Pellet, for example (`pellet.owldl.com`)).

A further important requirement for the IQ model is its ability to drive the semi-automated generation of software components that implement some of the concepts, specifically Quality Assertion functions. This feature is important for the support of rapid prototyping of quality functions as part of the IQ lifecycle. Indeed, model-driven component generation and composition has long been a goal in Software Engineering, to which Semantic Web technology has given fresh momentum. Our approach to component generation follows in the track of current applications of Semantic Web technology to service description, and is based on some minimal semantic annotation of service interfaces, from which some of the service implementation can then be automatically derived. As we will show in Chapter 6, describing quality functions as abstract concepts in a semantic

model provides suitable input to the software generation process; this will lead to practical support for the implementation of Quality Assertions as services.

A further point in favour of semantic modelling for our purposes concerns the back-end data management support for instances of semantic data. By choosing to model data and quality indicator types (i.e., Quality Evidence) as concepts in an ontology, we need to make sure that the instances of those concepts, i.e., the actual data elements and indicators values, can be properly represented, as well. The distinction between concepts and instances here is similar to that between a logical database schema and the data; concept instances are typically stored in a database using an RDF model,[1] a W3C standard. The RDF model views data elements as triples that are connected to form a graph. Although not as mature as relational databases, RDF data management technology can now manage volumes of data of the order of many millions of triples with response times that are almost comparable to those of relational databases [AMMH07]; this is a sufficiently scalable and robust data management infrastructure for Qurator.

3.2 An ontology for Information Quality

The IQ model formalizes the IQ paradigm of the previous chapter. It consists of two sets of concepts and relationships among concepts, namely (i) quality concepts that are independent of any data and application domain, and thus define the core ontology structure; and (ii) domain-specific concepts, which can be user-defined and which extend the core in an incremental way. We refer to the core as the *Information Quality Upper Ontology* (IQUO). The term *upper ontology* is widely used within the Semantic Web community to denote a high-level semantic model that is reusable and extensible over multiple application domains. In this section we present the IQUO along with examples of user-defined extensions, with reference to our main proteomics example.

The main concepts in the ontology reflect the IQ paradigm, and include:

- a taxonomy of the types of data to which quality assessment can be applied;
- a taxonomy of types of Quality Evidence, i.e., types of metadata that can be used as quality indicators;

[1] RDF: http://www.w3.org/RDF/

- a collection of Annotation functions that compute Quality Evidence metadata for the data;
- a collection of Quality Assertions functions that compute quality values from Quality Evidence.

The upper ontology includes top-level classes for these taxonomies of concepts. In addition, it also includes elements of the software environment where the data is processed, namely the software tools used by e-scientists. To see why these are useful, note that many bioinformatics tools compute results with some degree of uncertainty, for example a BLAST program used to perform a partial match between gene or protein sequences. These programs often associate a measure of confidence to the result, e.g. the "e-value" computed by BLAST indicates the likelihood that a significant overlap is seen between two biologically unrelated sequences (for a valid match, we expect this to be a very small number) [APS03]. If the e-value is used as Quality Evidence, then we also want to capture the fact that the e-value is computed by BLAST as part of our model. This is relevant knowledge that is legitimately part of the model, to the extent that it tells users of the ontology that BLAST is a good source of quality indicators that other experts have used to make quality-based decisions on the data.

The ontology is defined using a particular Description Logic (DL), i.e., one of a family of logic formalisms for knowledge representation. Although Description Logics have been known for many years in the logic community, they have recently gained new popularity as the formal way of representing concepts within the Semantic Web framework [FvHH$^+$01a, Hor02, BHS05a]. In particular, the OWL language has been proposed as a W3C standard for encoding a particular DL, denoted "OWL DL", using a Web-friendly, XML-based syntax [HPSvH03].

The IQ ontology is written in OWL DL. With this design choice, we can benefit from a substantial body of theoretical and practical research carried out in the Semantic Web context. On the theoretical side, automated reasoning on concept subsumption can be performed on OWL DL ontologies, for which decidability and complexity results are available. On the practical side, a number of software implementations of automated reasoners, ontology engineering tools and best practices are now available, through the growing Semantic Web community.

The IQ ontology follows in the tracks of recent efforts to provide semantic models for a variety of scientific domains, as a way to enable "semantically-aware" e-science applications. These models vary in their complexity, ranging

from controlled vocabularies, characterized by simple hierarchical class structures, to full-fledged OWL DL ontologies where logic axioms are used to define complex concepts. Controlled vocabularies tend to contain a large number of concepts that comprehensively describe a particular scientific domain; their purpose is to facilitate the systematic annotation of data in their domain, as well as the automated interpretation of existing annotations. Notable examples are the Gene Ontology (`www.geneontology.org`), MESH (Medical Subject Headings) maintained by Medline (`www.nlm.nih.gov/mesh`), the MGED ontology developed by the Microarray Gene Expression Data Society (`www.mged.org`) [WPCea06], and the numerous controlled vocabularies at the Proteomics Standards Initiative, PSI (`www.psidev.info`). The OBO initiative (Open Biomedical Ontologies, `www.obofoundry.org`) maintains a comprehensive index of ontologies in the biomedical domain.

At the opposite end of the spectrum we find the ^{my}Grid ontology [WSG+03a], designed to provide a controlled vocabulary of terms used in the bioinformatics domain. The ^{my}Grid ontology is mainly geared towards the semantic description of bioinformatics services. It is logically separated into two distinct components, the service ontology and the domain ontology. The domain ontology acts as an annotation vocabulary including descriptions of core bioinformatics data types and their relationships to one another, while the service ontology describes the physical and operational features of web services, such as their inputs and outputs. ^{my}Grid was originally written in the DAML+OIL language [FvHH+01b], a precursor to OWL, and makes extensive use of logic axioms to define complex concepts. For instance, we can write axioms to state that "every protein_structure_record must have at least one identifier, which is a protein_structure_record_id", or that "something is a protein_family_id if and only if it is an identifier of a protein_family_record".

The IQ ontology is closer in structure to the ^{my}Grid ontology than to a controlled vocabulary, in that logic axioms are used to define constraints on a small collection of top-level concepts, and domain-specific concepts are added only when needed, in an incremental way. These constraints are necessary in order to ensure that user-defined extensions of the IQUO do not disrupt the ontology structure. As we will see in the next section, this can be stated formally as a property of logic consistency of the ontology with respect to the axioms, and can be checked, with some extent, using automated reasoning.

Constructor	**Syntax**
universal concept	$\top$
bottom concept	$\bot$
concept subsumption	$C_1 \sqsubseteq C_2$
concept equivalence	$C_1 \equiv C_2$
concept intersection	$C_1 \sqcap C_2$
concept negation	$\neq C$
value restriction	$\forall R \,.\, C$
existential quantification	$\exists R \,.\, C$
atmost cardinality restriction	$\leq nR$
atleast cardinality restriction	$\geq nR$
exactly cardinality restriction	$= nR$
qualified atmost cardinality restriction	$\leq nR \,.\, C$
qualified atleast cardinality restriction	$\geq nR \,.\, C$
qualified exactly cardinality restriction	$= nR \,.\, C$

Table 3.1: OWL DL constructors (partial list)

3.2.1 OWL DL terminology and notation

We now provide a brief overview of the OWL DL language and abstract notation that will be used to describe the IQUO. For brevity, this only includes the operators that are required to understand the material in the rest of this chapter. Additional details, as well as a definition of the formal semantics of a number of Description Logics operators, can be found in [BCM+03].

OWL DL is based on concepts, or classes, denoted C and roles, i.e., binary relations among classes, denoted R. An individual o may be a member of one or more classes, denoted $o \in C$. A suite of constructors allows *constructed classes*, i.e., complex concepts and roles, to be built recursively from simpler ones. The following constructed class is often cited as a typical use of OWL DL constructors:

$$\texttt{Woman} \sqcap \exists\, \texttt{hasChild} \,.\, \texttt{Person}$$

This defines the class of all mothers, i.e., individuals of class `Woman` who are also in the relation `hasChild` with at least one `Person`. Here `Woman` and `Person` are atomic concepts. We denote generic atomic concepts with A, B etc.

Table 3.2.1 shows the syntax of some of the most commonly used OWL DL constructors. A full reference table, complete with the semantics of the constructors, can be found in [HPSvH03]. The formal semantics of OWL DL is based on an *interpretation* $\mathcal{I}$ that consists of a non-empty set $\Delta^{\mathcal{I}}$, called the interpretation

domain, and an interpretation function which maps every atomic concept A to a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, and every atomic role R to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation of complex constructs is defined recursively from that of simpler constructs. As an illustration, the interpretation for the construct shown in the previous example is the following:

$$(\texttt{Woman} \sqcap \exists\ \texttt{hasChild}\ .\ \texttt{Person})^{\mathcal{I}} = \texttt{Woman}^{\mathcal{I}} \cap (\exists\ \texttt{hasChild}\ .\ \texttt{Person})^{\mathcal{I}}$$

and

$$(\exists\ \texttt{hasChild}\ .\ \texttt{Person})^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists\ y.\langle x, y\rangle \in \texttt{hasChild}^{\mathcal{I}} \wedge y \in \texttt{Person}^{\mathcal{I}}\}$$

Note in particular that $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$ and $\bot^{\mathcal{I}} = \emptyset$.

Next, we introduce *terminological axioms*, of the form:

$$C_1 \sqsubseteq C_2 \text{ (inclusion axiom)}$$
$$C_1 \equiv C_2 \text{ (equivalence axiom)}$$

These axioms describe how concepts are related to each other. Equivalence axioms are often used to provide new names for constructed classes (which are otherwise anonymous). A set $\mathcal{T}$ of terminological axioms is called a *terminology*, or a *TBox*. Here is an example of TBox:

$$\begin{aligned}
\texttt{Man} &\equiv \texttt{Person} \sqcap \neg\texttt{Woman} \\
\texttt{Woman} &\equiv \texttt{Person} \sqcap \neg\texttt{Man} \\
\texttt{Mother} &\equiv \texttt{Woman} \sqcap \exists\ \texttt{hasChild}\ .\ \texttt{Person} \\
\texttt{Father} &\equiv \texttt{Man} \sqcap \exists\ \texttt{hasChild}\ .\ \texttt{Person} \\
\texttt{Parent} &\equiv \texttt{Father} \sqcup \texttt{Mother}
\end{aligned}$$

Some OWL DL operators are derived from others, but are sufficiently common to deserve their own syntax (which is, therefore, simply syntactic sugar). Among these are constructs that define the *domain* and *range* of a role R, defined as follows:

$$\begin{aligned}
dom(R) = C_1 \text{ iff} \quad & \exists R\ .\ \top \sqsubseteq C_1 \\
range(R) = C_2 \text{ iff} \quad & \top \sqsubseteq \forall R\ .\ C_2
\end{aligned}$$

For example, let $R = \texttt{hasPart}$, $C_1 = \texttt{car}$. If the following inclusion axiom holds:

$$\exists\, \texttt{hasPart} \,.\, \texttt{wheel} \sqsubseteq \texttt{car} \tag{3.1}$$

(read as: "anything that has a wheel as one of its parts is a car"), then `car` is in $dom(\texttt{hasPart})$. Viceversa, if `car` is in $dom(\texttt{hasPart})$, then by definition (3.1) holds. Similarly, C_2 is the range of R if and only if $y \in C_2$ for all pairs $\langle x, y\rangle \in R$. Note that we could have defined *range*() equivalently, using the inverse relation of R; however the definition we have given here is less restrictive, in that it applies to versions of the OWL language that do not include the inverse operator.

Similar derived constructs, with definitions omitted for brevity, are also introduced to indicate that an object property is *symmetric*, *functional*, *inverseFunctional*, or *transitive* (see [HPSvH03] for details).

We say that an interpretation $\mathcal{I}$ *satisfies* an inclusion axiom $C_1 \sqsubseteq C_2$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$, and that it satisfies an equality axiom $C_1 \equiv C_2$ if $C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$. If $\mathcal{T}$ is a TBox, then $\mathcal{I}$ satisfies $\mathcal{T}$ if $\mathcal{I}$ satisfies every element of $\mathcal{T}$. If $\mathcal{I}$ satisfies an axiom (resp. a TBox), then $\mathcal{I}$ is a *model* for the axiom (resp. the TBox). Finally, two axioms are *equivalent* if they have the same model.

A knowledge representation system based on DL can perform several types of inference tasks on a TBox $\mathcal{T}$. The most important task is to determine whether, given $\mathcal{T}$ and a new concept C, adding C to $\mathcal{T}$ leads to a contradiction. In logical terms, there is no contradiction if we can find a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}}$ is a nonempty set. A concept C with this property is *satisfiable* with respect to $\mathcal{T}$ and *unsatisfiable* otherwise. A second inference task is to test whether a concept is more general than another concept; this is the *subsumption* problem. Formally, a concept C_1 is subsumed by a concept C_2 with respect to $\mathcal{T}$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{T}$. In this case we write $\mathcal{T} \models C_1 \sqsubseteq C_2$.

The inference mechanism provided by existing DL systems checks for the subsumption of concepts. It can be shown [BCM+03] that this is sufficient to prove satisfiability, as well:

$$C \text{ is unsatisfiable} \iff C \text{ is subsumed by } \bot$$

$$C_1 \text{ is subsumed by } C_2 \iff C_1 \sqcap \neg C_2 \text{ is unsatisfiable}$$

As an example, in the TBox above the concept $\texttt{Woman} \sqcap \texttt{Man}$ is unsatisfiable, i.e.,

we can prove that $\mathcal{T} \models \texttt{Woman} \sqcap \texttt{Man} \sqsubseteq \bot$. In this case we say that `Woman` and `Man` are *disjoint*. Also, if we define $\texttt{P} \equiv \geq 1\ \texttt{hasChild}$, then $\mathcal{T} \models \texttt{P} \equiv \texttt{Parent}$.

A number of specific implementations of DL reasoners are available, for example Pellet[2], used as part of the Qurator workbench. In particular, given a concept C, the reasoner will infer all concepts C' such that $C \sqsubseteq C'$[3]. The subsumption reasoning functionality that is available in practice makes OWL DL a particularly appealing choice of semantic language for the IQ conceptual model. Indeed, as we will see in the rest of this chapter, we are going to exploit this functionality to verify that user-defined extensions to the IQUO do not introduce contradictions into the model. This is important for any extensible ontology that depends on incremental user contributions for its success and, ultimately, for its uptaking by the community.

3.2.2 Modelling class constraints as axioms

With the OWL DL notation in place, we can now present the IQ ontology in more detail. In doing this, we are going to describe the axiomatic modelling style that we have adopted for it, and argue that it is well-suited for this type of user-extensible ontology. The following description, made by an e-scientist, reflects our running example in proteomics and is an example of a complex scenario that we would like to model using the IQ ontology:

> "*Imprint* is a type of bioinformatics tool that performs protein identification on peak list data obtained through Peptide Mass Fingerprinting (PMF) technology. Its output is a "protein hit list" of elements, and is the result of a partial match. As such, among other items each of its elements contains (i) a protein ID, expressed as a Uniprot accession number; (ii) the extent of the match (*Coverage*); and (iii) the number of peptides in the peak list (*Peptides Count*). We have provided a quality characterization of the elements in the hit list by taking a number of raw quality indicators, including *Coverage* and *Peptides Count*, and combining them into a function, named *PIScoreClassifier*. This function is a classifier that assigns each item in

[2]Pellet: http://pellet.owldl.com/ (freely available as a Java program).

[3]Reasoners also perform inference on the *ABox*, i.e., a set of individuals for the classes defined in the TBox. We are going to ignore this functionality here, since the IQ ontology defines a TBox but not an ABox.

> the list to a particular quality class, by looking at combinations of values for the raw indicators. It uses a predefined set of class labels, named *PIScoreClassification.*"

Figure 3.2.2 provides a reference illustration for the IQUO. We are going to introduce the concepts and relationships shown in the figure throughout the rest of this section.

We begin by introducing the semantic types that describe the data for which quality assessments are made. These are atomic concepts, structured into a class hierarchy with the `Data Entity` class, denoted `DE` for brevity, as its root. The hierarchy is incrementally extensible with new user-defined classes. As mentioned, the idea of using ontologies to describe data types for the bioinformatics domain has been pioneered by the ^{my}Grid ontology, cited earlier, which includes a taxonomy of data artifacts that are produced and consumed by bioinformatics services. We have shown in [MPE+05] how the `DE` taxonomy of the IQ ontology can be integrated with the data domain part of the ^{my}Grid ontology.

In the example, we use the data entity class `ImprintHitEntry` to denote the collection of all protein identifiers that form the output of the Imprint protein identification tool, and we have `ImprintHitEntry` $\sqsubseteq$ `HitEntry` $\sqsubseteq$ `MIAPEEntity` $\sqsubseteq$ `DE`. MIAPE (the acronym stands for "Minimum Information About a Proteomics Experiment") is used in the context of proteomics by the Human Proteome Organization (HUPO, `www.hupo.org`) and provides format and content guidelines for describing proteomics experiments. Thus, in the IQ ontology a `MIAPEentity` is a generic "umbrella" class to be extended with data entites that are associated to the MIAPE guidelines. We have extended this class with a generic concept for `HitEntry`, i.e., an entry in a hitlist produced by a protein identification tool. As a further extension, `ImprintHitEntry` is specific to any such output that is produced by the Imprint tool (as opposed to other tools, like `Mascot` for example).

Next, we introduce `DataCollection`, a top-level class used to describe type-uniform collections of data, as well as the `entity-in-collection` property, with domain `DE` and range DataCollection, used to specify which data entities are part of which collection. With these, we may now model the first part of our scenario, namely that `ImprintHitEntry` is part of collection `PI_HitList` $\sqsubseteq$ `DataCollection`. Having modelled `ImprintHitEntry` and `PI_HitList` as classes, we cannot simply assert that property `entity-in-collection` (which is a binary relation) holds between them, because properties in DL hold between individuals, not classes.

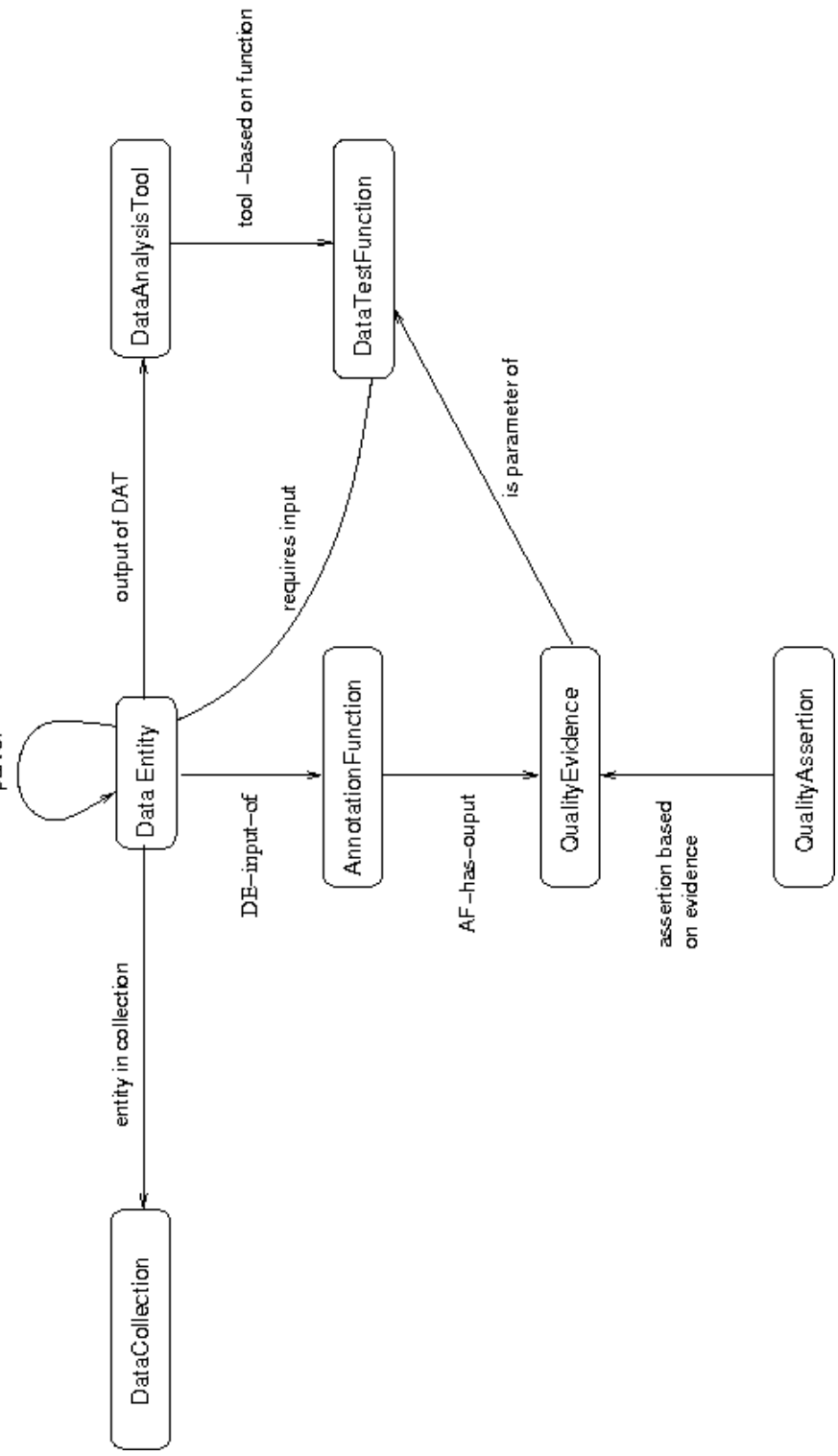


Figure 3.1: Main classes and properties in the Information Quality Upper Ontology. An arc label p from class D to R is interpreted as $dom(p) = D$, $range(p) = R$.

Thus, we have two main options. We can either model `ImprintHitEntry` and `PI_HitList` as individuals, rather than classes, (i.e., $\texttt{ImprintHitEntry} \in \texttt{DE}$ rather than $\texttt{ImprintHitEntry} \sqsubseteq \texttt{DE}$), and then assert the relation between them. Alternatively, we may use OWL DL constructors to represent the relationship as a class-level constraint, using axioms. If we choose the the second option here (this choice will be justified shortly), we may write:

$$\texttt{ImprintHitEntry} \sqsubseteq \exists\, \texttt{entity-in-collection} \,.\, \texttt{PI\_HitList} \tag{3.2}$$

$$\texttt{ImprintHitEntry} \sqsubseteq \forall\, \texttt{entity-in-collection} \,.\, \texttt{PI\_HitList} \tag{3.3}$$

$$\texttt{DE} \sqcap \texttt{DataCollection} \sqsubseteq \bot \tag{3.4}$$

We read the first axiom as "any data element of type `ImprintHitEntry` must be part, among other things, of some collection of type `PI_HitList`", while the second reads "if a data element of type `ImprintHitEntry` is part of any collection, then that collection is of type `PI_HitList`"[4].

As we can see, the combination of these two axioms ensures that any `ImprintHitEntry` data is part of some collection, which can only be of type `PI_HitList`. Note that this axiom-based modelling style follows naturally from our choice to represent domain-specific concepts as classes. Had we opted to model domain-specific concepts as individuals, we would have had an instance of the `entity-in-collection` relation, but no axioms. Each of the two options has its advantages and consequences and, in this case, there is no definite rule that we can follow to choose either of the two options. However we argue that, for a domain-extensible ontology, the axiom-based modelling style is preferrable.

The argument is based on three main observations. Firstly, that individuals cannot be further extended, an obvious limitation in our case. Secondly, that by using classes throughout, the role of individuals can be reserved for objects in the data space, i.e., actual data elements. An individual $\texttt{UniprotXYZ} \in \texttt{ImprintHitEntry}$ is a specific data element that is part of the actual output from a specific invocation of some instance of an Imprint tool. Reserving the role of individuals to "things" in the space of actual data leaves the domain expert free to extend the ontology class hierarchies indefinitely, to model concepts that

[4]Rector *et al.* provide a simple introduction to the intepretation of OWL DL operators and of constructed classes in [RDH+04]). The third axiom ensures, quite naturally, that `DE` and `DataCollection` are disjoint, i.e., data types do not overlap with data collection types.

are increasingly more domain-specific. Consistently with this view, an individual aPI_HitList ∈ PI_HitList is a particular dataset that is the result of some execution of an actual Imprint tool (which, as we will see, is itself modelled as an individual of some other class, namely Imprint). Thus, we could legitimately write ⟨UniprotXYZ , aPI_HitList⟩ ∈ entity-in-collection to represent a relation between instances, that holds in the data space. Note that, in addition to data, this distinction between classes and individuals applies to functions, as well. In Section 3.2.3 we propose to model quality functions as classes; consequently, the corresponding individuals are actual *software components* that realize the functions, typically as services.

Thirdly, and perhaps more interestingly, class-level axioms are effectively consistency constraints on the ontology that can be checked automatically using subsumption reasoning. To illustrate this important point, consider the following axiom:

$$\texttt{X} \sqsubseteq \exists\, \texttt{entity-in-collection} \,.\, \texttt{ImprintHitEntry} \tag{3.5}$$

where X is some new user-defined class. When we add this axiom to the TBox consisting of (3.2-3.4) and *dom*(entity-in-collection) = DE, *range*(entity-in-collection) = DataCollection, a DL reasoner makes the following inference:

$$\texttt{ImprintHitEntry} \sqsubseteq (\texttt{DE} \sqcap \texttt{DataCollection})$$

due to the range constraint. However, this contradicts (3.4), indicating an inconsistent ontology.

This simple example shows how reasoning may help detect user errors when updating the ontology (namely, that X can only be a data collection, not a data entity). Note that detecting an inconsistency is a special case of inferring the most specific subsumption relationship for a class (i.e., $\texttt{X} \sqsubseteq \bot$). In general, we can use subsumption reasoning to obtain a classification of classes that are not explicitly placed within any hierarchy. Unlike with other information models, this is possible and acceptable when using ontologies, exactly because we can count on the axioms to infer all subsumption relationships automatically. For example, if we had written (3.5) as

$$\texttt{X} \sqsubseteq \exists\, \texttt{entity-in-collection} \,.\, \texttt{DataCollection}$$

the reasoner would have inferred $\texttt{X} \sqsubseteq \texttt{DE}$ due to the domain constraint. We will present a further use of reasoning in Section 3.3, where we define a more complete TBox that allows a reasoner to infer logical associations between user-defined quality functions and abstract quality dimensions such as those discussed in the previous chapter.

Note that our proposed use for individuals also fits well with the natural meaning of a data collection, i.e., that of a database. Let us, for example, introduce the class $\texttt{ArrayExpress} \sqsubseteq \texttt{DataCollection}$. ArrayExpress (`www.ebi.ac.uk/arrayexpress`) is a well-known repository for microarray data maintained by the EBI. This new class therefore represents the collection of all possible deployments of an ArrayExpress database; an individual, say $\texttt{mainArrayExpress} \in \texttt{ArrayExpress}$, denotes a specific database instance (for example, the instance available at www.ebi.ac.uk/arrayexpress). Consistent with our distinction, the class `ArrayExpressEntity` denotes the type of all possible ArrayExpress data elements, whereas "P2E-MEXP-641", the identifier for an actual experiment stored in the EBI database, is an individual: $\texttt{P2E-MEXP-641} \in \texttt{ArrayExpressEntity}$. Thus, the relation $\langle \texttt{P2E-MEXP-641} \ , \ \texttt{mainArrayExpress} \rangle \in \texttt{entity-in-collection}$ may hold. These individuals, however, are not part of the ontology, because they belong to the dynamic data space rather than to the logical model. The set of all possible relation instances is instead captured by the two axioms:

$$\texttt{ArrayExpressEntity} \sqsubseteq \forall\, \texttt{entity-in-collection} \,.\, \texttt{ArrayExpress}$$

$$\texttt{ArrayExpressEntity} \sqsubseteq \exists\, \texttt{entity-in-collection} \,.\, \texttt{ArrayExpress}$$

Having established the distinction between classes and individuals, we will follow the axiomatic approach systematically in the rest of the IQ ontology. Property `contains-data-entity` provides a further example of this approach. It is used to model a *part-of* relationship among data entities, i.e., types that represent compound data structures (i.e., their domain and range are both `DataEntity`). Returning to our scientist's statement at the beginning of the section, we can use this property to model `ImprintHitEntry` as a complex type, consisting of `Mass`,

`Coverage`, and `UniprotEntity` (a Uniprot identifier):

$$\texttt{ImprintHitEntry} \sqsubseteq \exists \texttt{contains-data-entity} \,.\, \texttt{Mass}$$
$$\texttt{ImprintHitEntry} \sqsubseteq \exists \texttt{contains-data-entity} \,.\, \texttt{Coverage}$$
$$\texttt{ImprintHitEntry} \sqsubseteq \exists \texttt{contains-data-entity} \,.\, \texttt{UniprotEntity}$$

Together, these axioms are interpreted as "any `ImprintHitEntry` must contain, among other things, some `Mass` individual" (resp. `Coverage`, `UniprotEntity`). It is important to be aware of the different semantics implied by combinations of these axioms. The reasonable assumption that the three components are distinct, for example, must be stated explicitly as follows:

$$\texttt{Mass} \sqcap \texttt{Coverage} \sqcap \texttt{UniprotEntity} \sqsubseteq \bot$$

Also, note that the axioms above state conditions that are necessary but not sufficient: a compound structure that contains each of the three data entities need not be an `ImprintHitEntry`. Stating a sufficient condition requires an additional equivalence axiom, i.e.:

$$\texttt{ImprintHitEntry} \equiv \exists \texttt{contains-data-entity} \,.\, (\texttt{Mass} \sqcup \texttt{Coverage} \sqcup \texttt{UniprotEntity})$$

Finally, an individual of `ImprintHitEntry` may contain data entities of some other type. To exclude this possibility we must add a universal class restriction:

$$\texttt{ImprintHitEntry} \sqsubseteq \forall \texttt{contains-data-entity} \,.\, (\texttt{Mass} \sqcup \texttt{Coverage} \sqcup \texttt{UniprotEntity})$$

Choosing an appropriate level of detail for the model is an ontology engineering issue, for which only guidelines and "best practice" examples, rather than theoretical results, are available [GPFLC04]. The rest of the IQUO design, discussed in the remainder of this section, has been driven by the opportunity to exploit automated reasoning both to infer new knowledge that expert may consider useful, and to provide consistency guarantees on user actions. Examples of these applications are presented in Section 3.3 and in Section 4.4 of Chapter 4, respectively.

As a preview of the material ahead, we show a summary of the IQUO concepts and their properties in Table 3.2.2. The rest of this chapter provides a detailed account of these concepts.

Table 3.2: Summary of object properties in the IQUO

Property	Domain	Range
entity-in-collection	DataEntity	DataCollection
contains-data-entity	Data Entity	Data Entity
analysis-tool-based-on-function	Data Analysis Tool	Data Test Function
assertion-based-on-evidence	Quality Assertion	Quality Evidence
hasClassificationModel	ClassificationQA	ClassificationModel
input-of-annotator	Data Entity	Annotation Function
annotator-has-output	Annotation Function	Quality Evidence
QP-from-QA	Quality Property	Quality Assertion

3.2.3 Quality metadata and quality functions

We continue our overview of the IQUO by introducing the `Quality Evidence` class, `QE` for short, also in Figure 3.2.2 to model metadata that can be computed from the data and the data processing environment. An example of a user-defined hierarchy of Quality Evidence is `Coverage` $\sqsubseteq$ `PFMMatchReport` $\sqsubseteq$ `PIMatchReport` $\sqsubseteq$ `QualityEvidence`. Note that `Coverage` has already been introduced earlier, not as a kind of Quality Evidence but rather as a Data Entity. This is not an inconsistency; rather, it reflects the situation, common in practice, where a component of a data compound, `Coverage` in this case, is itself used as a piece of evidence to establish the quality of the entire compound, namely `ImprintHitEntry`. We will return to this point briefly in Section 3.2.5.

Continuing with Quality Evidence, we must now specify how subclasses of `QE` represent quality annotations, and how, in turn, these are used by QualityAssertion functions to compute the actual quality values upon which the user quality actions are based. To this end, we model functions as first-class citizens in the ontology. This design choice has two types of consequence. On the one hand, it presents the potential problem that functions can be sub-classes of other functions, i.e., $f_1 \sqsubseteq f_2$, thus requiring an explicit semantics for function subclassing. Shortly, we will present a simple axiomatization that is consistent with the standard Object-Oriented interpretation of function specialization. On the

other hand, this design choice allows us to describe aspects of the interface of those functions using OWL DL axioms; this gives us an opportunity for formal verification of some function composition properties. This is an important aspect of using quality functions in practice: as we will see in Chapter 4, quality functions are used as building blocks for *Quality Views*, i.e., processes obtained by composition of quality functions, which compute multiple quality classifications from the data.

One may note that relevant work has already been done on defining Semantic Web Services (SWS), i.e., services whose interfaces and, to some extent, internal behaviour are described using ontology concepts [MDBL07]. It is a common tenet of the SWS community that this will pave the way for large-scale service discovery and, subsequently, their automated composition into more complex services. Regarding this, we make two observations. First, while providing a rich description of the purpose of services is important to facilitate their discovery, we note that the type of application domain that quality functions, and therefore their possible incarnation as services, apply to (that of information quality) is quite restricted, probably making a full-fledged semantic discovery architecture unnecessary. And second, most current proposals regarding the semantic description of service behaviour are focused on the particular syntax used to describe the service's interface, for example WSDL-S (now SAWSDL [VS07a]), or OWL-S. Although quality functions are indeed eventually mapped to Web Services, their semantic description should be independent of this assumption, which is indeed not required in our case.

We now describe how quality functions are introduced in the IQ model, and how class subsumption and other OWL DL operators are used to define the semantics of function specialization.

The first step is to add two new class hierarchies to the IQUO, `AF` and `QA`, for the Annotation and Quality Assertion functions respectively. For example, `Imprint Annotator` $\sqsubseteq$ `AF` and `PIScore Classifier` $\sqsubseteq$ `QA`. Consistently with our earlier definition of Quality classification of data (Section 2.3), we further distinguish between two types of QA functions, those that compute a numerical score, `SQA` $\sqsubseteq$ `QA` for "Score QA", and those that assign class labels to data, `CQA` $\sqsubseteq$ `QA`, i.e., "Classification QA". Furthermore, ontology class `ClassificationModel` accounts for all possible class labelling that are available to `CQA` functions. Any subclass

of `ClassificationModel` defines a set of labels, as an enumeration of individuals, for example `PIScoreClassification`: {low_PI_Score, close_to_avg_PI_Score, high_PI_score}. Every CQA function is associated with exactly one classification model, so that the labels assigned to data elements during IQ assessment are all consistent with the model. This constraint is once again defined axiomatically, using a pattern similar to those shown earlier.

3.2.4 Modelling signatures of functions

As the next step, we cast our existing data and metadata classes as types that we can use to describe function signatures, as follows. Let $\mathcal{DE}$ denote the collection of all Data Entity classes, and let $\mathcal{QE}$ and $\mathcal{CM}$ denote the collections of all Quality Evidence and Classification Models, respectively. We use σ and τ to range over the types of the arguments and return values of functions: $\sigma, \tau \in \mathcal{DE} \cup \mathcal{QE} \cup \mathcal{CM}$. With this notation, a generic function f with n input parameters that returns an m-tuple of elements has the following signature:

$$f : \sigma_1 \times \cdots \times \sigma_n \rightarrow \tau_1 \times \cdots \times \tau_m \tag{3.6}$$

In particular, an Annotation function af has one input (a data element) and a m-tuple output:

$$\begin{gathered} af : \sigma \rightarrow \tau_1 \times \cdots \times \tau_m, \\ af \sqsubseteq \texttt{AF}, \;\; \sigma \in \mathcal{DE}, \;\; \tau_i \in \mathcal{QE} \;, \; 1 \leq i \leq m \end{gathered} \tag{3.7}$$

In contrast, a QA function qa has $n+1$ inputs (the data element and n annotations), and a single output:

$$\begin{gathered} qa : \sigma_0 \times \sigma_1 \times \cdots \times \sigma_n \rightarrow \tau, \\ qa \sqsubseteq \texttt{QA}, \;\; \sigma_0 \in \mathcal{DE}, \;\; \sigma_i \in \mathcal{QE}, 1 \leq i \leq n, \;\; \tau \in \mathcal{CM} \end{gathered} \tag{3.8}$$

The OO principles of function (i.e., method) polymorphism can be summarized by the following two rules:

1. if f has a signature as in (3.6), then it must accept an input consisting of an n-tuple $\langle x_1 \ldots x_n \rangle$ of type $\sigma'_1 \ldots \sigma'_n$, with $\sigma'_i \sqsubseteq \sigma_i$ for each i;

2. If function f' *specializes* f, then the type of each of its input parameter is a a sub-type of the corresponding parameter of f, and the type of the result is sub-type of those of f (and possibly of higher cardinality). Formally, let f be defined as in (3.6), and

$$f' : \sigma'_1 \times \cdots \times \sigma'_n \rightarrow \tau'_1 \times \cdots \times \tau'_k, \quad k \geq m \tag{3.9}$$

If $f' \sqsubseteq f$, then $\sigma'_i \sqsubseteq \sigma_i$ for $1 \leq i \leq n$, and $\tau'_j \sqsubseteq \tau_j$, $1 \leq j \leq m$.

To model these rules in the IQUO, we introduce the following properties[5]:

- `DE-input-of` with domain `DE` and range `AF`
- `AF-has-output` with domain `AF` and range `QE`
- `QE-input-of` with domain `QE` and range `QA`
- `CQA-has-cm` with domain `CQA` and range `CM`

Then, for an Annotation function *af* as in (3.7), the following axioms are added to the ontology:

$$\sigma \equiv \exists \texttt{DE-input-of} \,.\, af \tag{3.10}$$

$$af \sqsubseteq \exists \texttt{AF-has-output} \,.\, \tau_i, \; 1 \leq i \leq m \tag{3.11}$$

where $\sigma \sqsubseteq \texttt{DE}$, $\tau_i \sqsubseteq \texttt{QE}$. (3.10) defines a necessary and sufficient condition, namely that *af* has one single input, which is σ, while (3.11) states the necessary condition that τ_i is an output of *af* —but note that *af* may have a multi-valued output. As an example, here are the axioms for the `ImprintAnnotation` function:

$$\texttt{ImprintHitEntry} \equiv \exists \texttt{DE-input-of} \,.\, \texttt{ImprintAnnotation} \tag{3.12}$$

$$\texttt{ImprintAnnotation} \sqsubseteq \exists \texttt{AF-has-output} \,.\, \texttt{PeptidesCount} \tag{3.13}$$

$$\texttt{ImprintAnnotation} \sqsubseteq \exists \texttt{AF-has-output} \,.\, \texttt{Masses} \tag{3.14}$$

(the other output axioms are similar and omitted for brevity).

These axioms are consistent with properties (1) and (2) above, regarding the polymorphism for signatures:

[5]The different choices of domain and range for different properties, reflected in their names (e.g. `has-output` vs. `input-of`), has to do with the need to perform automated inference, and will become clear in the next section.

- regarding (1), let $\sigma' \sqsubseteq \sigma$. If $\sigma \sqsubseteq \exists$ `DE-input-of` . af, then it follows immediately that $\sigma' \sqsubseteq \exists$. `DE-input-of` . af, i.e., af accepts input of type σ';

- regarding (2), observe that if $af' \sqsubseteq af$, and $\sigma' \sqsubseteq \exists$. `DE-input-of` . af', then $\sigma' \sqsubseteq \exists$. `DE-input-of` . af. And because $\exists$ `DE-input-of` . $af \sqsubseteq \sigma$, then $\sigma' \sqsubseteq \sigma$.

It is also easy to see that if $af \sqsubseteq \exists$ `AF-has-output` . τ, i.e., af returns value of type τ, and $\tau \sqsubseteq \tau'$, then $af \sqsubseteq \exists$`AF-has-output` . τ' also holds.

The axioms for QA functions are similar to those for Annotation functions:

$$\sigma_i \sqsubseteq \exists \texttt{QE-input-of} \,.\, qa,\ 1 \leq i \leq n \tag{3.15}$$

$$qa \sqsubseteq \exists \texttt{CQA-has-CM} \,.\, \tau \tag{3.16}$$

3.2.5 E-science services as a source of quality indicators

We now elaborate on the case, briefly touched upon in Section 3.2.3, of a `DataEntity` class, such as `Coverage`, that is also a kind of `Quality Evidence`. This situation is common when the data entity is the output of some e-science service. This is typically the case for values computed by predictive tools like Imprint or Mascot that associate a measure of confidence, i.e., a score, with their prediction. For instance, as mentioned earlier, well-known algorithms for gene or protein sequence alignment, like BLAST, provide a statistical estimate of the probability that the resulting alignment occurs by chance (the *e-value*).

Thus, the output from these algorithms is modelled naturally as a compound data structure consisting of the data that is the subject of the quality assessment, as well as some quality evidence. In the IQUO we acknowledge this situation by introducing class `Data Analysis Tool` (`DAT` for short). Imprint and MASCOT, for example, are two DATs:

$$\texttt{Imprint} \sqsubseteq \texttt{PMFMatchAnalysisTool} \sqsubseteq \texttt{PIAnalysisTool} \sqsubseteq \texttt{DataAnalysisTool}$$

$$\texttt{MASCOT} \sqsubseteq \texttt{PMFMatchAnalysisTool} \sqsubseteq \texttt{PIAnalysisTool} \sqsubseteq \texttt{DataAnalysisTool}$$

With this class, however, we cannot properly tell the difference between Imprint and Mascot: they are both a kind of `PMFMatchAnalysisTool`. To allow for a finer description of these tools, we also introduce class `Data Test Function` (`DTF`) to

represent some functionality or algorithm, so that a DAT can be described as a composition of simpler DTFs. Consider for example the DTF `PMFMatchFunction`, a subclass of `Data Test Function` that represents the common function of protein identification on peptide peaklists obtained by Protein Mass Fingerprinting (PMF). We can state that `Imprint` and `MASCOT` each include a version of `PMFMatchFunction`, as follows:

$$\texttt{Imprint} \sqsubseteq \exists\ \texttt{analysis-tool-based-on-function}\ .\ \texttt{Imprint Match}$$

$$\texttt{MASCOT} \sqsubseteq \exists\ \texttt{analysis-tool-based-on-function}\ .\ \texttt{MASCOT-PMFMatch}$$

with `Imprint Match` $\sqsubseteq$ `PMFMatchFunction` and `MASCOT-PMFMatch` $\sqsubseteq$ `PMFMatchFunction`. With these axioms and the new property `analysis-tool-based-on-function` between `DataAnalysisTool` and `Data Test Function` (see also Figure 3.2.2 on page 72), we have now clarified the similarities and differences betweeen `Imprint` and `MASCOT`, namely that they include similar functionality, but require a different implementation of that functionality. In practice, we use DTFs to describe a few, selected features of a DAT that are relevant as a source of Quality Evidence.

Furthermore, we use property `requires-input-parameter` to capture the input type of these tools:

$$\texttt{Imprint} \sqsubseteq \exists\texttt{requires-input-parameter}\ .\ \texttt{PeptidePeaklist} \tag{3.17}$$

$$\texttt{MASCOT} \sqsubseteq \exists\texttt{requires-input-parameter}\ .\ \texttt{PeptidePeaklist}$$

and `PeptidePeaklist` $\sqsubseteq$ `MIAPE Entity`, a type of proteomics data entity. Depending on the amount of knowledge that we want to capture regarding a certain tool, a domain expert may use additional axioms to describe stronger properties, for instance:

$$\texttt{Imprint} \sqsubseteq \forall\texttt{requires-input-parameter}\ .\ \texttt{PeptidePeaklist} \tag{3.18}$$

The combination of (3.17) and (3.18) is interpreted as "Imprint requires an input parameter of exactly one type, namely a peptide peaklist" (i.e., it won't accept any other type of input, and the peptide peaklist is required). Finally, observe that some DATs can compute Quality Evidence values, because we allow a DAT

to be a subclass of some Annotation function[6].

To summarize, we have introduced class hierarchies to account for the fact that bioinformatics tools may compute both data and quality metadata. By adding appropriate axioms on these classes, we may describe the relationship between different tools, by enumerating some of their common elementary functionalities, and identify the specific types of quality metadata they produce.

3.3 Further role of reasoning in the IQ ontology

Our use of reasoning so far has been to guarantee that user extensions to the IQUO do not result in logical inconsistencies (an indication of modelling errors). In particular, we have made the implicit assumption that users may introduce new classes and axioms, but not new properties, as these form the "backbone" of the ontology. As a complete example of these extensions, Table 3.3 provides a summary of the user-defined classes and concepts that capture the example scenario proposed at the beginning of the previous section. A corresponding visual rendering of the relevant class hierarchies is shown in Figure 3.2[7].

We now present a different application of reasoning, that is made possible by our axiomatic approach. In the introductory chapter we have briefly mentioned a number of *quality dimensions*, for example *Currency*, *Completeness*, and *Accuracy*, that the data management community has adopted as a reference framework for Data Quality. Some of these dimensions are defined in abstract terms: the accuracy of a value v for example is defined in terms of the degree of similarity between v and a reference value v', which is considered correct. A specific definition of Accuracy contains a parameter, the similarity function.

In the IQ model presented so far, these dimension concepts do not seem to have a role. Indeed, we have been arguing that scientists should be able to provide different definitions of quality depending on the data types and context of use, rather than by relying on abstract quality dimensions. Nevertheless, we also recognize that those quality dimensions may represent a useful common vocabulary that can facilitate the classification of user-defined quality properties. In this section we demonstrate on a practical example that the IQ ontology can be used to establish such a classification. In order to do this, we are going to (i)

[6]In OWL DL, classes may have multiple parents.

[7]The color-coding in this and in the following figures is an artifact of the protege tool used to produce it, and has no particular meaning.

Table 3.3: Summary of axioms for the Imprint proteomics example

ImprintHitEntry ⊑ DataEntity Mass ⊑ DataEntity UniprotEntity ⊑ DataEntity Mass ⊑ QualityEvidence	Data
Masses ⊑ QualityEvidence Coverage ⊑ QualityEvidence PeptidesCount ⊑ QualityEvidence	Quality Evidence
ImprintHitEntry ⊑ ∀ entity-in-collection . PI-HitList	Data and data collections
ImprintHitEntry ⊑ ∃contains-data-entity . Mass ImprintHitEntry ⊑ ∃contains-data-entity . UniprotEntity	Compound data structures
ImprintAnnotation ⊑ AnnotationFunction ImprintHitEntry ≡ ∃ DE-input-of . ImprintAnnotation ImprintHitEntry ⊑ (∃ is-output-of-DAT. Imprint) ImprintAnnotation ⊑ ∃ AF-has-output . PeptidesCount ImprintAnnotation ⊑ ∃ AF-has-output . Masses ImprintAnnotation ⊑ ∃ AF-has-output . Mass ImprintAnnotation ⊑ ∃ AF-has-output . Coverage	Annotation function, their inputs and outputs An ImprintHitEntry is, among other things, the input to ImprintAnnotation An ImprintHitEntry is, among other things, the output of some Imprint tool. Multiple outputs of ImprintAnnotation
Imprint ⊑ PMFMatchAnalysisTool ⊑ DataAnalysisTool ImprintMatch ⊑ DataTestFunction Imprint ⊑ ∃analysis-tool-based-on-function . ImprintMatch PeptidePeaklist ⊑ dataEntrity peptide-peaklist ⊑ (∀ entity-in-collection . PIExperimentDataCollection) Imprint ⊑ ∃requires-input-parameter . PeptidePeaklist Imprint ⊑ ∀requires-input-parameter . PeptidePeaklist	Imprint is an Analysis Tool Imprint is based, amongst other things, on function ImprintMatch peptide-peaklist can only be part of collection PIExperiment DataCollection Imprint, amongst other things, requires input parameter PeptidePeaklist and no other type of parameter
PIScoreClassifier ⊑ QualityAssertion PIScoreClassifier ⊑ (∃ assertion-based-on-evidence . Coverage) PIScoreClassifier ⊑ (∃ assertion-based-on-evidence . Mass) PIScoreClassifier ⊑ (∃ assertion-based-on-evidence . Masses) PIScoreClassifier ⊑ (∃ assertion-based-on-evidence . PeptidesCount) PIScoreClassifier ⊑ (∃ hasClassificationModel . PIScoreClassification) PIScoreClassifier ⊑ (∀ hasClassificationModel . PIScoreClassification)	Quality Assertions PIScoreClassifier, amongst other things, is based on multiple types of evidence PIScoreClassifier has exactly one type of classification model, PIScoreClassification

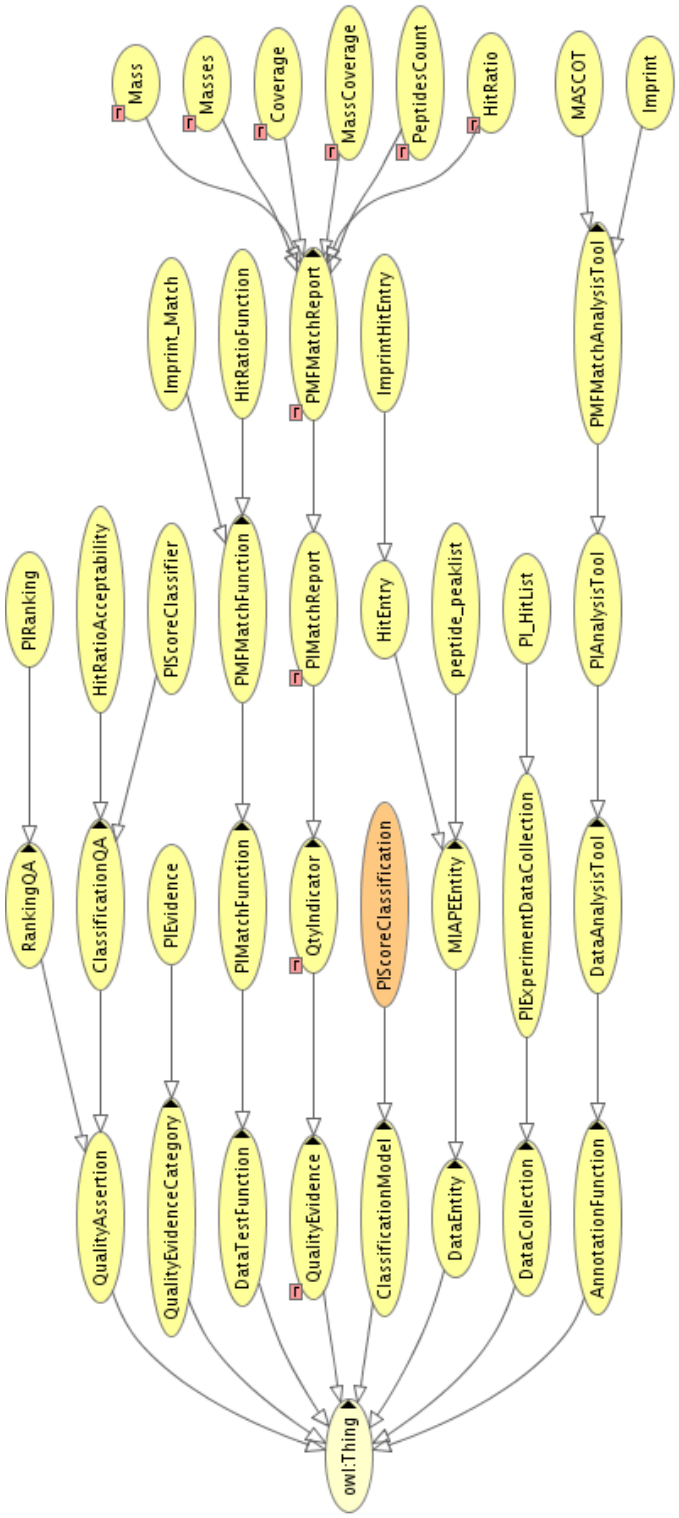


Figure 3.2: Partial view of the class hierarchy for the Information Quality ontology, with concept for the Imprint example

add axiomatic definitions for abstract quality dimensions to the IQUO, and (ii) use a DL reasoner to automatically infer that some user-defined quality properties of data are in fact a specialization of those quality dimensions. A preliminary version of the material presented in this section can be found in [MPE+05].

We proceed incrementally. The first step is to extend the IQUO with the new top-level `QualityProperty` class, along with a number of sub-classes that reflect, in part, the taxonomy presented in Chapter 2 (Figure 3.3). For the sake of illustration, we are only going to provide a definition for the `Accuracy` class. The same modelling pattern used in the example, however, can be applied to other dimensions as well. Note also that the taxonomy includes a domain-specific class, `PI-acceptability` $\sqsubseteq$ `QualityProperty`; this is the only class in the hierarchy that is user-defined and not part of the IQUO. We are going to show that, by suitable modelling of domain classes for the proteomics example, we can infer that `PI-Acceptability` $\sqsubseteq$ `Accuracy`, i.e., that the scientist's effort in producing personal quality definitions results in a specialization of a known dimension. We argue that adding this newly acquired information to the ontology is useful in terms of reuse of quality knowledge, because it provides an answer to queries such as, "what types of Accuracy-related quality functions have been defined for the proteomics domain?".

The second step is to establish a relationship between the axiomatic quality properties and the operational quality definitions. As we recall, these definitions consist of Quality Evidence types and Quality Assertions. Correspondingly, we add two new features to the ontology.

First, we introduce a new collection of `Quality Characterization` classes, i.e., `Confidence-QC`, `Specificity-QC`, `Reputation-QC`, etc, and a new property: `has-QC` ("has Quality Characterization"), with domain `Quality Evidence` $\sqcup$ `DataTestFunction` and range `Quality Characterization`. We are going to define the "Accuracy" property in terms of these new, intermediate concepts. The goal is to provide scientists with a way to associate explicit quality characterizations to their own QE types, as well as to the families of software tools that compute QE values. For example:

$$\texttt{PIMatchReport} \sqsubseteq \exists\ \texttt{has-QC}\ .\ \texttt{Confidence-QC}$$

associates a quality characterization to a user-defined QE type. Specifically, the

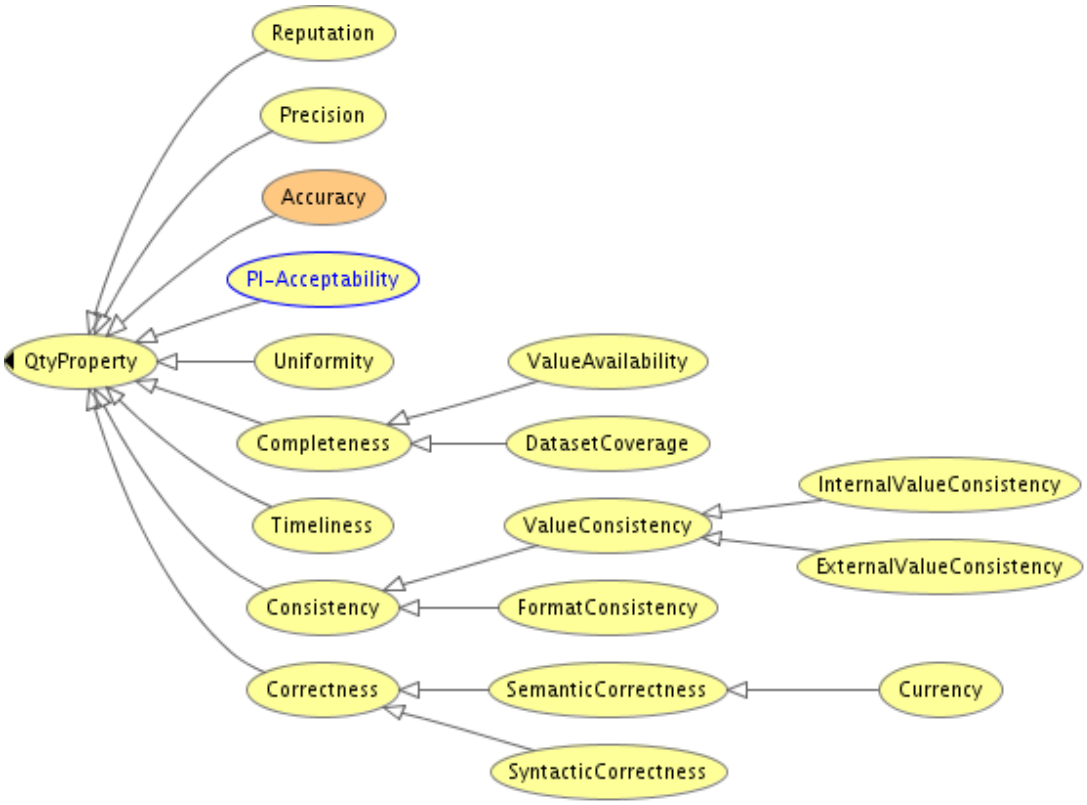


Figure 3.3: Part of the generic Quality Properties classes

axiom states that anything that is a `PIMatchReport` must have a quality characterisation `Confidence-QC`.

Second, we introduce a new property `QP-from-QA` (a shorthand for "Quality Property from Quality Assertion"). Since we know from Section 3.2.3 how to link `Quality Assertion` classes to their QE input, this is all we need to close the gap between the "user level", consisting of concrete quality functions, and the generic ones consisting of abstract quality properties. Thus, we can now define `PI-Acceptability` in terms of a Quality Assertion class, in this case `PIScoreClassifier`, as follows:

$$\texttt{PI-Acceptability} \sqsubseteq \exists\ \texttt{QP-from-QA}\ .\ \texttt{PIScoreClassifier} \tag{3.19}$$

Intuitively, with this axiom we are stating that the abstract quality property that describes the "acceptability" of a data element can be measured in practice using the `PIScoreClassifier` Quality Assertion function. (note that the existential restriction indicates that the same property may be additionally defined in terms

of other Assertion functions, as well). In other words, the `PI-Acceptability` property reflects the specific "fitness for use" criteria that is defined by the `PISCoreClassifier` Assertion.

Our third step is to create new constructed classes that represent QE types that have the same quality characterization. For example:

$$\texttt{ConfidenceEvidence} \equiv \texttt{QualityEvidence} \sqcap (\exists\ \texttt{has-QC}\ .\ \texttt{ConfidenceQC}\ )$$

is the class of all and only the Quality Evidence types that also have a `ConfidenceQC` quality characterization (similarly, we could define `ReputationEvidence`, etc. using the same pattern). Intuitively, we can interpret `ConfidenceQC` as a "label" associated by users to items of quality metadata, to indicate that such metadata is used to measure the confidence in the value of the underlying data. For example, a sequence alignment program like BLAST may produce a sequence, along with a value (the e-value) to indicate the level of confidence in that sequence. In this case, we interpret the e-value as a piece of Quality Evidence that has a `confidenceQC` characterisation in terms of quality. The axiom above introduces an explicit `ConfidenceEvidence` class to represent the set of all such QE types. A more complete version of the same axiom follows. In this version, we allow for the possibility that software tools that produce QE values, in addition to the values themselves, can also be tagged with a specific quality characterisation:

$$\begin{aligned}
&\texttt{ConfidenceEvidence} \equiv \\
&\texttt{QualityEvidence} \sqcap (\exists\ \texttt{is-output-of-DAT}\ .\ (\exists\ \texttt{has-QC}\ .\ \texttt{ConfidenceQC})) \\
&\sqcup \\
&\texttt{QualityEvidence} \sqcap (\exists \texttt{is-parameter-of}\ .\ (\exists\ \texttt{has-QC}\ .\ \texttt{ConfidenceQC})) \\
&\sqcup \\
&\texttt{QualityEvidence} \sqcap (\exists\ \texttt{has-QC}\ .\ \texttt{ConfidenceQC}\ )
\end{aligned}$$

Note that we can use anonymous constructed classes in axioms, for example to specify "any class that is output of any DAT that has a Confidence Quality characterization".

With these definitions, a reasoner infers the following class subsumption relationships:

$$\texttt{PIMatchReport} \sqsubseteq \texttt{ConfidenceEvidence}, \tag{3.20}$$

$$\texttt{MassCoverage} \sqsubseteq \texttt{ConfidenceEvidence}, \tag{3.21}$$

$$\texttt{PMFMatchRanking} \sqsubseteq \texttt{ConfidenceEvidence} \tag{3.22}$$

Using these new classes we can now provide a formal definition of the Accuracy property. Consider the following axiom:

$$\begin{aligned} \texttt{Accuracy} \equiv\ & (\exists\ \texttt{QP-from-QA}\ . \\ & (\exists\ \texttt{assertion-based-on-evidence}\ . \\ & (\ \texttt{SpecificityEvidence}\ \sqcup\ \texttt{ConfidenceEvidence}\ ))) \end{aligned} \tag{3.23}$$

This defines Accuracy as a quality property that corresponds to any QA that is based on any QE with a quality characterization of either Specificity or Confidence. It is important to clarify that this is only one of many possible definitions of Accuracy, and that it is not our goal to offer a definitive view on specific quality dimensions. Rather, the point here is that the IQ ontology is sufficiently expressive to allow this type of definition to be given in a formal way, so that it can be used to make interesting automated inferences on user-defined classes. Indeed, we may finally prove that

$$\texttt{PI-Acceptability} \sqsubseteq \texttt{Accuracy} \tag{3.24}$$

Intuitively, the proof proceeds as follows. From

$$\texttt{PiScoreClassifier} \sqsubseteq \exists\ \texttt{assertion-based-on-evidence}\ .\ \texttt{MassCoverage}$$

and (3.21) we derive

$$\texttt{PiScoreClassifier} \sqsubseteq \exists\ \texttt{assertion-based-on-evidence}\ .\ \texttt{ConfidenceEvidence}$$

From this and (3.19), it follows that

$$\texttt{PI-Acceptability} \sqsubseteq (\exists\ \texttt{QP-from-QA}\ .\ (\exists \texttt{assertion-based-on-evidence}\ .\ \texttt{ConfidenceEvidence}))$$

which is the definition of `Accuracy` (3.23).

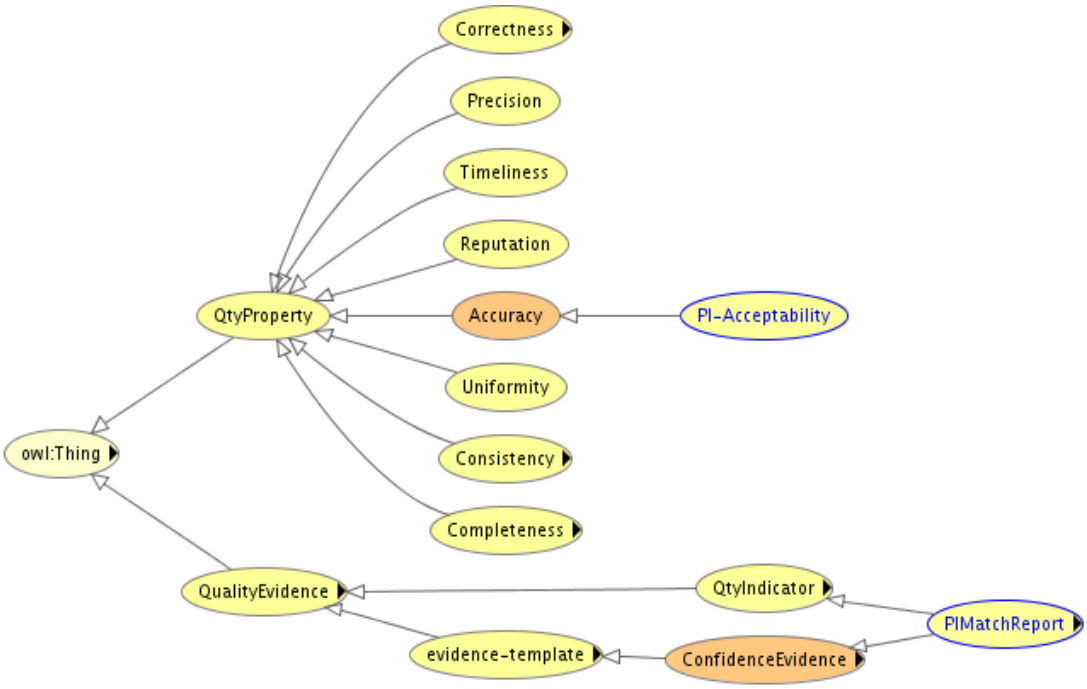


Figure 3.4: Inferred hierarchy for the `PI-Acceptability` class

Figure 3.4 shows a new version of the ontology fragment of Figure 3.3, after the reasoning process.[8] In conclusion, we have shown how the axiomatic definition of complex concepts in the IQ ontology can be leveraged to perform automated classification of user-defined classes. In this case, the effect of of the classification is to describe operational definitions of quality in terms of a general framework for quality dimensions.

[8]The `evidence-template` class is simply a "container" class where we collect all the class definitions for quality characterization.

3.4 Summary and conclusions

In this chapter we have presented an ontology of Information Quality concepts. We have described the IQ upper ontology (IQUO), and shown how our proteomics example can be modelled by extending the IQUO with domain-specific concepts. We have also demonstrated value to users by showing how we can systematically use OWL DL axioms in the ontology to define consistency constraints, and how to exploit OWL DL reasoning to enforce the constraints. We have further shown how user-defined quality dimensions can be automatically classified as part of a IQUO taxonomy of dimensions, drawn from traditional Data Quality literature.

In the next chapter we build upon the IQ ontology to define Quality Views, an abstract process model for computing quality features. The ontology provides a semantic interpretation of Quality Views, opening the way to the definition of *consistent* Quality Views and to an algorithm for checking consistency.

Chapter 4

Quality Views

A rational man is one who makes a proper use of reason: and this implies, among other things, that he correctly estimates the strength of evidence. [†]

In the previous chapter we have described a semantic model for Information Quality, designed to provide abstract definitions for quality functions, which we called *Quality Assertions* (QA). Building upon that model, this chapter is concerned with the composition of Quality Assertions into more complex quality processes. Our goal is to automate, as much as possible, the process of computing Quality Assertions on user datasets, in the context of existing data processing environments, notably dataflows and query processing.

Central to this idea is the notion of a *Quality View* (QV), i.e., a view on the data that reflects a user's personal definition of quality criteria. The term Quality View was chosen to suggest that a variety of different quality definitions can be applied to the same data, resulting in multiple views. The information provided by a particular view allows a user to make informed decisions regarding the acceptance of data elements during the course of data processing. Informally, a Quality View is the specification of an abstract process that involves (i) collecting Quality Evidence values for a dataset, (ii) applying one or more QA functions to the Evidence to compute quality classes, and (iii) associating quality actions to the quality classes and performing those actions.

The main research contribution presented in this chapter is a formalization

[†]A. J. Ayer, Probability and Evidence, 1972, Columbia University Press.

of the Quality View concept. Specifically, we first describe an XML-based surface syntax for the Quality Views language, and then define its formal semantics in purely functional terms, by presenting a complete QV interpreter written in the Haskell functional programming language. Then, we define the formal notion of a *consistent* Quality View, grounded in the Information Quality ontology presented in the previous chapter. As Quality Views specify a composition of functions, we need to provide guarantees that the composition is consistent with the input/output requirements of the functions involved. Consistent views have the property that all the input requirements for the functions involved are satisfied. Thus, consistency is a pre-requisite for computing Quality Views in practice – the topic of the next chapter. In the last section of the chapter we present a practical algorithm to support the users in this phase of the lifecycle, by providing an interactive, visual environment for the specification of QVs with consistency guarantees. We find that, although the algorithm is based upon the semantic IQ model, users can be guided in their specification of Quality Views without the need for any knowledge of ontologies.

The rest of the chapter is organized as follows. We begin with an informal description of Quality Views in Section 4.1, along with their surface XML syntax (Section 4.2), and an example QV that defines quality controls on the proteomics data that is processed by the example workflow. The formal definition of QV is presented in Section 4.3, where the Haskell interpreter is introduced. Finally, Section 4.4 gives a definition of Quality View consistency, along with the algorithm for verifying consistency using a DL reasoner, and provides an account of the user environment for the specification of consistent Quality Views.

4.1 Overview of Quality Views

Informally, a Quality View consists of one or more QA functions, along with the specification of Annotation Functions (AF) that compute the input values for each QA, and of *quality actions*, which indicate what should be done with the data once the QA values are available. It should be clear from this definition that the elements involved in a QV, with the exception of actions, are all defined as part of the IQ ontology. Thus, a QV is essentially the declarative specification of a process that coordinates the application of functions, i.e., AF and QA, for which an abstract definition is given in the ontology, followed by a mapping of

quality classes to actions, and by the execution of the actions.

The rationale for allowing multiple QAs as part of a single QV was discussed earlier, in Section 2.3. To recall the argument briefly, we noted there that we may want to compare two slightly different score models to see how they perform on the same input data. After all, a QA function is a predictive model that assigns either a class or a score to the data, based on input vectors of attributes (the QE). Since such assignments are predictive, it is important to be able to compare their output side by side, on the same data. Users may then decide to use one of the predictors as a source of quality value, or more generally, they may want to define a combination of their values. Such combinations can be specified using actions as part of the Quality View.

As a particular case, one QA function may compute an assertion that is derived from that of other functions within the same QV. Thus, in our library of functions we have both a `PIScore` and a `PIScoreClassifier`: the first computes the score described in [SPB06], while the second simply performs a discretization of the score into classes (i.e., "buckets") based on the score distribution, and assigns each data element to one of the classes. Having both functions computed by a single Quality View allows users to view both the raw score and the classification side by side, facilitating the decision process.

4.1.1 Role of semantics in Quality Views

In addition to providing a definition for the functions involved in a QV, we also use the semantic IQ model to determine which QV can be applied to which actual dataset that is part of the data processing environment: this defines the *scope* of a QV relative to the space of all possible datasets. More precisely, the scope is an ontology class `DE'` $\sqsubseteq$ `DE`, for instance, `HitEntry`, and is specified as part of the QV definition. This is interpreted as "the QV can be computed on any data element that is an individual of class `DE'`, or of any of its sub-classes".

This requires, of course, that actual datasets of interest be associated to ontology classes that describe data entities, i.e., that they be *semantically annotated*. For example, we must be able to state that a particular dataset that is computed by some protein identification algorithm consists of elements that are all individuals of the `HitEntry` class. In general, a process of semantic annotation associates some element of a conceptual model –ontologies being a popular example, to some object in a different space, e.g. a data element, a service, or other artifact.

While creating such annotations may not be a trivial task for the QV designer or the e-scientist, this process is broadly recognized within the Semantic Web community as a pre-requisite for the semantic analysis of data and services [PKPS02, LH03, HZB+]. By requiring a semantic annotation of the data involved in the workflow (or in query processing), our definition of QV scope follows in the tracks of an established approach that has found acceptance in a relevant community. This approach has three important advantages. Firstly, it improves the chances for QV reusability, because a QV is tied not to a specific dataset, but rather it can be applied to any dataset whose semantic type is within the QV scope. Secondly, it decouples the general task of semantically annotating the data from that of specifying a QV: the QV designer will just assume that the input data is within the logical scope defined in the ontology, while a QV user is responsible for associating actual datasets to ontology classes. Finally, as noted previously, it makes it possible to leverage existing taxonomies of popular data types for e-science, notably (but not only) the *my*Grid ontology [WSG+03b, WAH+07].

4.1.2 Quality View components

In this section we give an informal description of the QV process, using the IQ model as a reference. In the following, we assume that each data element that is the subject of quality assessment, which may be an arbitrary data structure, has a unique identifier, called a *dataref*, and that an access path (for example, a database query) is available to retrieve the element given its dataref. Initially, the QV takes a finite collection D of datarefs as input, and computes QE annotations by applying one or more AFs to them. Each AF is identified by a reference to an ontology class $\texttt{AF}' \sqsubseteq \texttt{AF}$, and computes QE values for each $d \in D$. A QE type $\texttt{QE}'$ is itself a reference to an ontology class in the taxonomy rooted at QE: $\texttt{QE}' \sqsubseteq \texttt{QE}$.

As an example, consider Uniprot accession number `Q9JLJ2`, a unique identifier for a protein, as input dataref, and annotation function `ImprintAnnotator`, which computes Quality Evidence values of types `Mass` and `Coverage`. The output of `ImprintAnnotator` looks like the following: $\langle \texttt{Q9JLJ2}, [\langle \texttt{Mass}, 58855.6\rangle, \langle \texttt{Coverage}, 14.2\rangle]\rangle$.

The QE values are then used as input to Quality Assertion functions, which are again identified using ontology classes $\texttt{QA}' \sqsubseteq \texttt{QA}$. As mentioned, QAs compute, for each data element, either a class label from

a finite set (called a `ClassificationModel` in the ontology), or a numerical score. In the example we use a QA called `PIScoreClassifier` that takes as input the output of `ImprintAnnotator` and, for each pair defined as above, uses the QE annotations to compute a label (i.e., one element from the set {`low_PI_Score`, `close_to_avg_PI_Score`, `high_PI_Score`}), and associates it to the dataref, for instance: $\langle$Q9JLJ2, [$\langle$Mass, 58855.6$\rangle$, $\langle$Coverage, 14.2$\rangle$, $\langle$PIScoreClassifier, close_to_avg_PI_Score$\rangle$]$\rangle$

In the IQ paradigm presented in Chapter 2, we defined *actions* as abstract processes that are associated to quality classes through conditional expressions on the output of QAs. The expressions that define a mapping from the quality classes (one for each QA) to the abstract actions are the last part of the QV specification. To illustrate, consider our default actions, i.e., "accept" and "reject", based on a single condition defined on the QE and QA values computed as described above. The mapping from quality classification to actions might be (`Mass` $>$ 60000 $\cap$ `PiScoreClassifier` $\neq$ ``low_PI_Score'') $\rightarrow$ `accept`. Note that QE elements are allowed as part of the condition.

As discussed in Chapter 2, at this point actions are abstract. Their binding to actual processes depends on the environment where the QV is executed, and is not specified as part of the QV. When the QV is executed in the context of a workflow, for instance, the step of the QV that computes the classes-to-actions mapping may be implemented as a workflow processor that evaluates the condition and inhibits the output of the "reject" elements. Thus, we have a *logical mapping* of classes to actions, defined as part of the QV, and a *physical mapping*, defined as part of the QV implementation for a specific data processing environment. The physical mapping defines the precise behaviour of the "accept" and "reject" actions. For a QVs that is implemented as part of a query processor, a new physical mapping for the same logical mapping could be implemented as an additional selection step within the query plan.

To emphasize this point further, consider an environment where other types of actions are possible, in addition to "accept" and "reject". For example, suppose that a data presentation application makes it possible to highlight data records differently depending on their quality. In this case, one may use a 3-way classification, i.e., {"green", "yellow", "red"}, along with the following logical mapping

conditions, say:

$$\texttt{PiScoreClassifier} = \texttt{low\_PI\_Score} \rightarrow \texttt{red}$$
$$\texttt{PiScoreClassifier} = \texttt{close\_to\_avg\_PI\_Score} \rightarrow \texttt{yellow}$$
$$\texttt{PiScoreClassifier} = \texttt{high\_PI\_Score} \rightarrow \texttt{green}$$

Only the logical mapping is specified as part of the QV. The enactment of the action, i.e., the actual colouring of the data, is done by the physical mapping, which is specific to the application.

The QV syntax allows for the specification of a generic type of action, called "n-way Splitter", involving n conditions $[c_1 \dots c_n]$ and $n+1$ user-defined classes $[cl_1 \dots cl_{n+1}]$. These conditions need not be mutually exclusive, i.e., the Splitter simply evaluates each c_i on a data element d, and assigns it to *all* cl_i for which $c_i(d)$ is true. This semantics is formalized in Section 4.3. The particular case of a Filter Action type, i.e., a binary classification, is sufficiently common to deserve its own syntax.

QV parameters

From the description given so far, it would appear that the input to annotation and QA functions consists only of data elements, or of quality values computed from the data elements. However, consider a QA that assigns a score to a data element based on its similarity to other elements in the input dataset, as in record matching. The similarity function involves a configurable threshold value, which is not a function of the input data. If we want the algorithm to be configurable at execution time, then we need to supply the threshold value to the QA as an actual parameter.

To account for this situation, the QV language allows parameter-passing to functions, by optionally including a set of formal parameter names as part of the QV declaration. These parameters are global to the entire QV. In addition, formal parameter names may also appear within each function declaration in the QV. This allows functions to use their private parameter names, by declaring how they are mapped to the global names.

With this additional specification, the input to a QV now includes a set of actual parameters in addition to the data input. The values are passed on to each function that has a parameter declaration by binding them to local actual

parameters using the local-to-global name mapping.

4.2 Quality View syntax

In this section we present the XML syntax of Quality Views in some detail (the XML schema for the language is depicted graphically in Figure 4.2). In the next chapter we will see how QVs written in this language are translated into executable services. The main purpose of this presentation is to give an idea of the effort required from scientists who decide to specify a new QV. In Section 4.4, however, we will describe a graphical tool that facilitates the specification task. Thus, the syntax presented here can be regarded as the output of the tool, and the input to the QV translator.

4.2.1 XML Elements

The code in Figure 4.1 shows the specification of a Quality View that includes the QAs designed for our proteomics example. We use variables for the inputs and outputs of Annotation functions and Quality Assertion functions. Thus, variable `Coverage` denotes Quality Evidence values, one for each input dataref, that represent one of the outputs of `ImprintAnnotator` and one of the inputs of `PIScoreClassifier`. The same variables can be used in the conditions that map the output of QAs to actions.

One important feature of the QV specification language is that all variables have a *semantic type*, i.e., a reference to some class in the IQ ontology. In practice, the QV consists of XML syntax that is semantically annotated using the ontology. As mentioned earlier, and explained in detail in Section 4.4, the ability to provide a complete semantic interpretation of the QV variables makes it possible to define a formal model of semantic consistency for QVs, and to perform static consistency validation of a QV using the ontology.

Let us now walk through the QV example in detail. The top element of the language, <`QV`>, has an attribute to indicate that this view can be computed on data of type `q:ImprintHitEntry`, a reference to a class in the `DE` taxonomy within the IQ ontology.[1] As mentioned, we assume that appropriate semantic tagging of datasets of interest has been performed, so that this declaration makes the scope

[1]Throughout the example, `q` is a prefix for the Qurator namespace of the IQ ontology.

Figure 4.1: A Quality View for the proteomics example

```
<QV name ="QV_Imprint" data = "q:ImprintHitEntry">
    <Annotator serviceName="ImprintAnnotator"
       serviceType="q:ImprintAnnotator">
       <variables repositoryRef="ImprintAnnotations">
          <var metricName="q:PeptidesCount" variableName="PeptidesCount"/>
          <var metricName="q:Coverage" variableName="Coverage"/>
          <var metricName="q:Mass" variableName="Mass" />
          <var metricName="q:Masses" variableName="Masses"/>
       </variables>
    </Annotator>
    <QualityAssertion serviceName="HR_MC_PIScore"
       serviceType="q:HR_MC_PIScore"
       tagName="HR_MC_PIScore">
       <variables repositoryRef="ImprintAnnotations">
          <var metricName="q:PeptidesCount" variableName="PeptidesCount"/>
          <var metricName="q:Masses" variableName="Masses"/>
          <var metricName="q:Coverage" variableName="Coverage"/>
          <var metricName="q:Mass" variableName="Mass"/>
       </variables>
    </QualityAssertion>
    <QualityAssertion serviceName="PIScoreClassifier"
       serviceType="q:PIScoreClassifier"
       tagName="PIScoreClassifier">
       <variables repositoryRef="ImprintAnnotations">
          <var metricName="q:PeptidesCount" variableName="PeptidesCount"/>
          <var metricName="q:Masses" variableName="Masses"/>
          <var metricName="q:Coverage" variableName="Coverage"/>
          <var metricName="q:Mass" variableName="Mass"/>
       </variables>
    </QualityAssertion>
    <action name="initial_filter_action">
       <filter xmlns="" mode="noauto" interactive="true">
       <variables repositoryRef="ImprintAnnotations">
          <var metricName="q:PeptidesCount" variableName="PeptidesCount"
          <var metricName="q:Masses" variableName="Masses"
          <var metricName="q:Coverage" variableName="Coverage"
          <var metricName="q:Mass" variableName="Mass"
       </variables>
          <expression>((Mass &gt; 100000) and (Coverage &lt; 10)) ||
             PIScoreClassifier in "high_PI_score", "close_to_avg_PI_Score"
          </expression>
       </filter>
    </action>
</QV>
```

of applicability of this QV unambiguous.

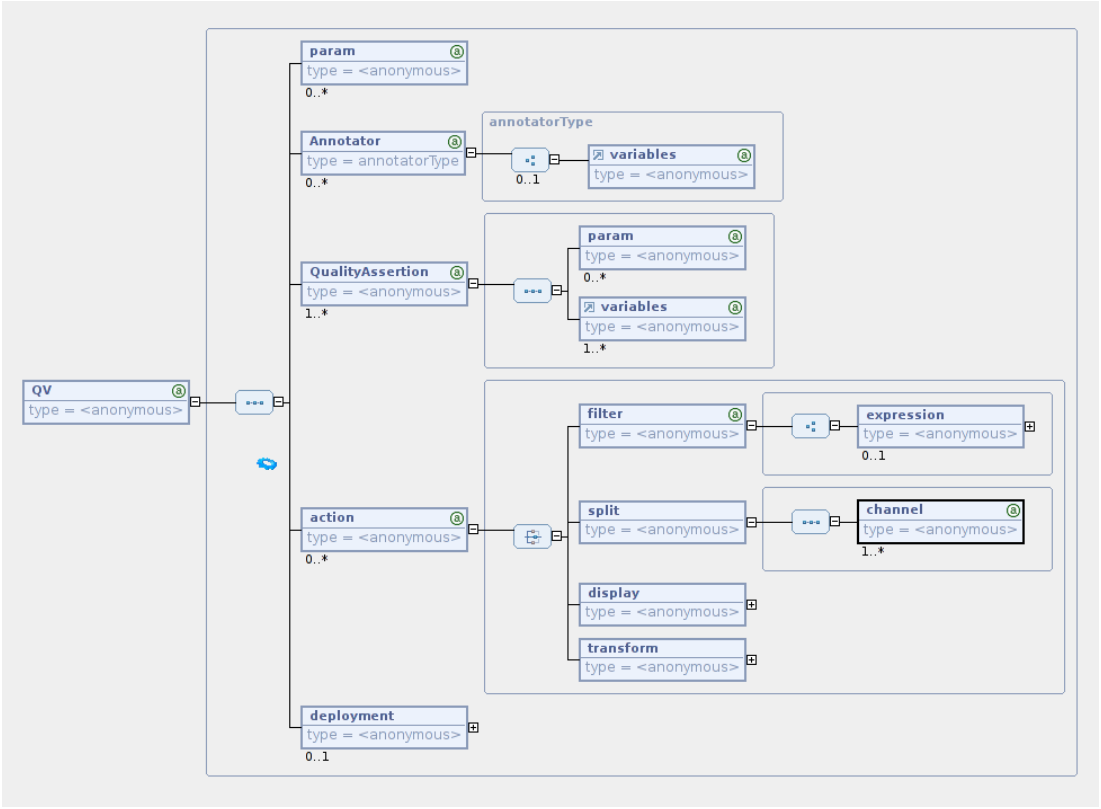


Figure 4.2: Graphical depiction of XML schema for Quality Views syntax

Annotation functions are specified using one or more `Annotator` elements. The nested <`variables`> element declares new variables for the Quality Evidence values computed by the Annotation functions. In addition to the semantic type, e.g. `q:PeptidesCount`, these variables also have a *syntactic type*, expressed using the XML Schema type system (`www.w3.org/2001/XMLSchema`). This type is not mentioned explicitly in the specification, because it is associated with the semantic type and thus it can be retrieved by querying the ontology. Note that, as expected, the Annotation function itself has a semantic type, namely `q:ImprintAnnotator`, a subclass of `q:AnnotatorFunction`. We discussed the meaning and implications of function hierarchies in Section 3.2.3.

The specification of Quality Assertions is syntactically similar to that of Annotation functions, except for the additional `tagName`, for instance `HR_MC_PIScore`, that represents the computed Quality Assertion value. Note that neither the semantic nor the syntactic types of this name need to be mentioned. Again, this is because these are defined as properties of the `q:HR_MC_PIScore` class in the

ontology. Note also that, although the elements used for the QA variables are the same as those for used for the Annotators, in this case they are interpreted, without any ambiguity, as the *input* of the QA rather then its output.

The generic <Action> element admits nested elements for specific action types. As mentioned in the previous section, the syntax currently supports Splitter actions (a particular decision tree with cascading conditions). The example shows a filter action with an associated conditional expression. The expression may contain references to any of the variables defined as part of the <Action> element. Intuitively, we can see the role of variables here: they are bound to values when Annotation functions are executed, these values are used by QA functions to compute quality classifications, and their values can be part of Action expressions.

The BNF grammar of the conditional expression language for QV Actions can be found in Appendix A.

4.2.2 Formal parameters

As noted, our running example does not involve the use of any parameter. Therefore, we present here a simple additional example.

```
<QV name ="SomeQV" data = "q:someData">
    <param name="p1"/>
    <param name="p2"/>
    <param name="annotationContext"/>
    <Annotator serviceName="someAnnotator"
       serviceType="q:someDataAnnotator">
       <param localName="context" globalName="annotationContext"/>
       etc...
    <QualityAssertion serviceName="QAWithParameters"
       serviceType="q:someQA"
       tagName="QAOutput">
          <param localName="x" globalName="p1"/>
          <param localName="y" globalName="p2"/>
       etc...
```

Here, the implementation of `QAWithParameters` expects actual parameters named `x` and `y`. When this QA is used as part of a QV, the additional mapping from local names to global names ensures that the correct actual parameters at the QV level are bound to the QA local parameters.

4.2.3 Semantic naming constraints

A QV is *syntactically valid* if it conforms to the XML schema shown in figure 4.2. In addition, however, we need to define constraints on the names of the variables that appear in the QV. The purpose of these constraints is to ensure that the QV can be mapped to a process model with a well-defined semantics. Although these are semantic constraints, they are presented here because they can be checked as part of traditional static language analysis. This is in contrast to the semantic consistency of a QV, discussed in Section 4.4.

Informally, the following constraints on variable names are defined:

1. the sets of output variables for each Annotator must be mutually disjoint;
2. the set of input variables to a QA must be a subset of the union of all output variables over all Annotators;
3. the set of variables that form the scope of an Action expression must be a subset of the union of all output variables over all Annotators, and of all output tag names over all Quality Assertions.

These constraints ensure that no Annotators may override each other's values, when computed as part of the same QV, and that an Action expression may only predicate on declared variables.

One last constraint involves the use of constant class names in set membership conditions of the form "*var* `IN` $\{$`c1`, `c2`, $\dots\}$", that may appear in Action expressions. Such class names (`c1`, `c2`, etc.) cannot be arbitrary; rather they must be defined as part of some classification model in the IQ ontology. More precisely, all classification models associated with the QAs included in the QV can be used as literals in the actions expressions. For example, consider ontology class `PIScoreClassifier`, which is associated with class `q:PIScoreClassification`, a sub-class of `q:ClassificationModel`, by way of property `q:has-classification-model`. Class `q:PIScoreClassification` is defined as an enumeration, consisting of individuals "high_PI_score",

"close_to_avg_PI_Score", and "low_PI_score". These are all and only the literals `c1`, `c2`, ... that may appear in a set membership condition. Note that, unlike the constraints in the previous list, which can be checked without the use of of the IQ ontology, this requires access to the ontology. More complex semantic checks are discussed in Section 4.4.

4.2.4 Additional attributes

The syntax showcased by the QV example above includes a few additional attributes which have not yet been discussed, since they are used only to control the execution-time behaviour of a QV. Briefly, the `mode` and `interactive` attributes in the `action` elements determine whether actions should be carried out interactively, under the control of the user, or silently (without interaction from the user, i.e., in batch mode). In interactive actions, users are allowed to inspect the runtime values of the QV variables and to edit the conditions associated with the actions before they are evaluated. This gives users an additional chance, while the QV is being computed, to experiment with different classes-action mappings, and to fine-tune them right in the context of the surrounding data processing environment, i.e., a user workflow.

The `repositoryRef` attribute in the <`variables`> element has to do with the physical allocation of QE values to specific metadata repositories. The `persistent` attribute determines the lifetime of the annotations, i.e., whether they should be deleted from the repository upon completion of the QV execution (this is useful when the scope and lifetime of annotations are known to coincide with the data processing that the QV is part of). For example, in our proteomics experiment the QE values, for instance of `Coverage`, are computed from the Imprint output. This means that these values are only valid within the scope of the experiment in which Imprint is invoked. When Imprint is invoked again, a new set of QE values must be computed. This makes the metadata non-persistent: they lose meaning at the end of one experiment execution. By contrast, QE values that annotate data that is persistently stored in a database, is itself persistent: it is not necessary to recompute it every time we need to assess the quality of the persistent data. This idea can be generalized, in the future, by specifying conditions under which a QE value becomes invalid, and must be recomputed (for example, because the underlying data has changed).

4.3 Quality Views Semantics

We move now to the definition of the formal semantics of Quality Views. A QV can be described as a function that takes a collection of data elements as input, and computes an assignment of the data elements to quality classes, defined in the previous section as the output of QV Actions. Here we give a formal definition of this semantic function, which was described informally earlier, by presenting a QV interpreter written in the Haskell functional language [HHJW07].

Although the interpreter is designed primarily as a formal specification of QV semantics, its prototype implementation also provides a testbed that can be used to experiment with QV language extensions, a topic of current research. The code has been tested using the Haskell Glasgow Compiler[2] [JHH+93].

The complete interpreter code appears in Appendix B.

4.3.1 Environment

The interpreter consists of a composition of functions. A shared data structure to which all functions have access, the *environment*, is used to manage the intermediate results, i.e., the outputs of functions that are to be used as input to some other function.

The environment is a matrix with one row for each input data element d_i, i.e., a dataref. A cell in row i holds the value of an item of quality metadata for element d_i, i.e., a Quality Evidence value or the value of a Quality Assertion. Thus, the number of columns in the matrix is equal to the number of variables declared by each Annotation function in the QV, summed over all such functions, plus one column for each Quality Assertion in the QV. Each column is identified by its corresponding variable name, along with its semantic type. Here is the Haskell declaration for this data structure:

```
data QTriple    =
        QTriple { _Name :: String, _Class :: URI, _Value :: String }
type QTripleSet = [ QTriple ]
data EnvRow     = EnvRow { d :: URI, qSet :: QTripleSet }
type Env        = [ EnvRow ]
```

[2]See also http://haskell.org/haskellwiki/Research_papers for an extensive list of papers. The compiler is freely available at www.haskell.org/ghc.

where the URI is simply a formatted string. Thus, a `QTriple` represents the value of a single cell. Note that the name and semantic type are associated with cells, rather than with columns, simply as a technical convenience.

A collection of functions is provided to manipulate the environment, namely to add a `QTriple` to a row, to add entire rows, to write values into cells, retrieve rows and cells, and so forth. In particular, `fetchAnnotations` returns the list of cells corresponding to a set of input variable names, for a specific dataref:

```
fetchAnnotations :: Dataref -> [String] -> Env -> [QTriple]
```

while `allQTriples` returns an entire row:

```
allQTriples :: Dataref -> Env ->  [ QTriple ]
```

and `getQTriple` returns an individual cell:

```
getQTriple :: String -> String -> Env -> [QTriple]
```
[3]

4.3.2 Formal representation of Quality Views

We now present the Haskell definition of a QV, corresponding to the XML syntax described previously. At the core of the QV specification are the Annotation functions and QA functions. In Haskell, these are described by the following types, respectively:

```
type FormalParamName = String
type ActualParamValue = String
type LocalFormalParam = (FormalParamName, FormalParamName)
type BoundActualParam = (FormalParamName , ActualParamValue)

type Af = Dataref -> [BoundActualParam] -> [QTriple]
```

and

```
type QAf = [AnnotatedData]->
                    [BoundActualParam]->
                          [(Dataref,[QTriple])]
```

An Annotation function computes values for some columns in the environment, for each input dataref and given actual parameters, while a QA takes a list of

[3]Note that `getQTriple` returns a list for purely technical reasons. The list is always a singleton.

annotations (i.e., a row) and actual parameters, and computes a new QTriple (a singleton list). Note that these functions do not update the environment directly: this is the responsibility of the interpreter functions that coordinate their invocations, as we will see shortly.

Here are some examples of these functions:

```
af1  = \x -> \plist -> [QTriple
                          { _Name = "e1",
                            _Class = "e1Class",
                            _Value = (x ++ " e1 annot")},
                         QTriple { _Name = "e2",
                            _Class = "e2Class",
                            _Value = (x ++ " e2 annot")} ]
```

Function `af1` computes values for two Quality Evidence variables, `e1` and `e2`. Similarly:

```
af2  = \x -> \plist -> [QTriple
                        { _Name = "e3",
                          _Class = "e3Class",
                          _Value = (x ++ " e3 annot")}]
```

computes one annotation value, for `e3`. Here is an example QA function:

```
qa1  = \ad -> \plist -> [ (d, [QTriple
                                 { _Name = "q1",
                                   _Class = "q1Class",
                                   _Value = "QA1 for "++" d " }]) |
                                   (d, _) <- ad ]
```

Note that a QA takes an entire annotated dataset as input and computes one value for each dataref in the set (in this fictional example, the QE values are provided but not used). In these examples, prepared for illustrative purposes only, the functions do not perform any useful computation. In a realistic implementation, their actual behaviour would be implemented using external services. In the workflow implementation of QVs, described in the next chapter, functions are mapped to workflow processors, which execute by invoking external Web Services.

The following data structures correspond to the QV elements `<Annotator>` and `<QualityAssertion>`, respectively:

```
data AnnSpec = AnnSpec { _Af :: Af,
                         _outputVars :: [String],
                         _AnnParameters :: [LocalFormalParam] }
```

and

```
data QASpec = QA { _QAf :: QAf,
                   _inputVars :: [String],
                   _outputVar :: String,
                   _QAParameters :: [LocalFormalParam] }
```

which include instances of `Af` and `QAf` functions. The following examples use the functions defined above:

```
ann1 = AnnSpec { _Af = af1,
                 _outputVars = ["e1", "e2"],
                 _AnnParameters = [("p1","p1Local")] }
```

for `AnnSpec`, and

```
_QASpec1 = QA { _QAf = qa1,
                _inputVars = ["e1", "e2"],
                _outputVar = "q1",
                _QAParameters = [("p3","p3")] }
```

for `QA`.

Actions are represented by the `QTest` data structure:

```
type CondExpr = Dataref -> Env -> Bool

data QTest = QTest {
                     action :: String,
                     actionDescr :: String,
                     cond :: CondExpr }
```

Field `action` is the quality class label that is assigned to a dataref when the conditional expression `cond` evaluates to true, while `channelDescr` provides a natural language description of the quality class (this could also be fetched from the ontology when needed). This facilitates its interpretation outside the scope of the QV, but is not used by the interpreter.

In Haskell, the condition is itself a function of a dataref and the environment, for example:

```
_aCond = \d ->  \e ->
                      (_Value (head (getQTriple d "q1" e))
                       == "QA1 for "++ d )
```

This condition returns true iff the QA value assigned to variable `q1` matches the string on the right hand side. Here is a `QTest` that assigns the datarefs that satisfy the condition to quality class "white":

```
_Qtest1 = QTest { action = "action1",
                  actionDescr = "white",
                  cond = _aCond}
```

With these definitions, the entire QV specification is described by the following data structure:

```
data QVSpec  = QVSpec { _formalParams :: [FormalParamName],
                        _ann :: [AnnSpec],
                        _QA :: [QASpec],
                        _QT :: [QTest] }
```

The following QV uses the examples that we have seen so far (plus others, similar, that are not shown for brevity):

```
testQV = QVSpec { _formalParams = ["p1", "p2", "p3"],
                  _ann = [ann1, ann2],
                  _QA = [_QASpec1, _QASpec2],
                  _QT = [_Qtest1, _Qtest2] }
```

4.3.3 Functional interpretation of Quality Views

We are now ready to describe the QV interpreter. The top-level function `qv`:

```
qv :: [Dataref] -> [ActualParamValue] -> QVSpec -> [DatarefQualityClass]
```

takes a `QVSpec` along with its input (a list of datarefs and the actual parameters, a list of Strings), and computes a list of quality classes, defined by the following structure:

```
type DatarefQualityClass = (Dataref, String, QTripleSet)
```

where the second element is the quality class label associated to the dataref, and the third is the list of annotations used in the classification.

The complete interpreter definition follows:

```
qv _D  _actualParams _QVSpec =
   let env = initEnv _D (collectVars _QVSpec)
       _boundParams = paramBinding
                        (_formalParams _QVSpec)
                        _actualParams
   in act _D (_QT _QVSpec)
       (qAssert _D  (_QA _QVSpec) _boundParams
           (annotate
               _D
               (map   (\x -> ((_Af x),
                           (globalToLocalParams
                                _boundParams
                                (_AnnParameters x)
                            )
                       )
                   )
               (_ann _QVSpec)
               ) env
           )
       )
```

The `let` construct provides a binding for variables, prior to their use in the functions following the `in` keyword. Here it is used to initialize the environment `env` and to bind the formal parameters `_boundParams` to the actual parameters. The composition involving functions `annotate`, `qAssert` and `act` reproduces the sequence of operations described informally in Section 4.1. A walkthrough of its definition follows.

Annotate.

The signature for `annotate` is as follows:

```
annotate :: [Dataref] -> [(Af, [BoundActualParam])] -> Env -> Env
```

Reading the interpreter definition above starting with the innermost expression, the `map` creates a list of pairs `(Af, actualParameters)` by extracting all annotation specifications of type `AnnSpec` from the input `_QVSpec`, and associating the actual parameters to each of them. The `annotate` function is defined recursively on this list:

```
annotate _ [] e  = e
annotate _D (h:rest) e = annotate _D rest (annotate1 _D h e)
```

and

```
annotate1 :: [Dataref] -> (Af, [BoundActualParam]) -> Env -> Env
annotate1 _D (_Af, _parms) e = multiUpdateEnv
                               [ (d, ( _Af d _parms )) | d <- _D ]
                               e
```

The list comprehension syntax:

```
[ (d, ( _Af d _parms )) | d <- _D ]
```

is interpreted as the iterative application of `_Af` to each `d` in the dataset `_D`. The environment is then updated with the result (all the rows needed for the full data set are in the environment).

QAssert

As mentioned, `annotate` returns an updated version of the environment, which is then read by the `QAssert` function. Its signature is the following:

```
qAssert :: [Dataref] -> [QASpec] -> [BoundActualParam] -> Env -> Env
```

Note that, while Annotation functions operate on one dataref at a time, QAs need access to the entire dataset at once. This reflects a typical requirement of predictive model algorithms (for instance, classifiers), which operate globally at the dataset level despite the fact that class labels are assigned to individual data elements.

The function definition is recursive on the list of `QASpec` functions:

```
qAssert _ [] _ e = e
qAssert _D (h:rest) _actualParams e =
                    qAssert _D rest _actualParams
                    (qAssert1 _D h _actualParams e)
```

and

```
qAssert1 :: [Dataref] -> QASpec -> [BoundActualParam] -> Env -> Env
qAssert1 _D _QASpec _actualParams e  =
                     multiUpdateEnv
                     (
                        (_QAf _QASpec)
                        [ (  d,
                             (fetchAnnotations
                                  d
                                  (_inputVars _QASpec) e
                              )
                           ) |
                           d <- _D
                        ]
                        (globalToLocalParams
                              _actualParams
                             (_QAParameters _QASpec)
                        )
                     )
                     e
```

In the list comprehension, datarefs are paired with their annotations by environment lookup, prior to being fed to the `QAf` function. The environment is updated to include the new `QTriple` computed for each dataref by the `QAf` function.

QAct.

Finally, the updated environment obtained from the application of `Assert` is used by `Act`, which evaluates the conditional expressions associated to Actions and assigns datarefs to quality classes:

```
act :: [Dataref] -> [QTest] -> Env -> [DatarefQualityClass]

act _D []  e     = [ (d, "all data", (allQTriples d e)) | d <- _D ]
act _D _Tests e  = [ (d, (channelName aTest), (allQTriples d e)) |
                                d <- _D, aTest <- _Tests,
                                (cond aTest) d e  == True ]
```

Consistent with the informal semantics specified for Actions in Section 4.1, conditions are evaluated independently from one another.

Note that the condition is stored as a string within the `QTest` data structure, i.e. `(cond aTest)` where `aTest` is bound to each of the input `_Tests` (one for each `QTest` in the QV). This string is evaluated by applying it (as if it were a function) to input `d`. Note also that in this case the recursion is on the list of actions, while a double iteration takes place within each application of `act`, namely on each of the tests for the action, and on each dataref in the dataset.

4.4 Formal consistency of Quality Views

In the previous sections we have presented Quality Views as abstract processes, obtained by a composition of Quality Evidence and Quality Assertion functions. Nothing has been said so far, however, regarding the *correctness* of the composition. In principle, we could assemble a QV using any combination of functions from the `QE` and `QA` class hierarchies; the semantics defined in the previous section does not restrict the choice of such functions in any way. This may result in type mismatches between the actual input to a function, and the input type expected by the function. In this section, we fill this gap by providing a formal definition of type compatibility constraints needed by QVs, as well as an algorithm to test them. When the constraints are satisfied, we say that the QV is *consistent*. As we will see in the next chapter, a consistent QV can be successfully translated into an executable software component that computes the abstract QV process.

To define compatibility constraints, we leverage the axiomatic formulation of function signatures, proposed in the previous chapter (Section 3.2.3). In particular, we are going to show that (i) the axioms associated to the `DE`, `AF`, `QE`, and `QA` classes are sufficient to give a formal definition of QV consistency; (ii) the type

compatibility constraints are consistent with the rules for function polymorphism, and (iii) constraints can be verified using an algorithm based on DL reasoning.

4.4.1 QV consistency constraints

Suppose that we are to specify a QV for input data of type `ImprintHitEntry`. One question that the user scientist can pose is, what Annotation functions are available in the ontology that accept input of this type? The equivalence axiom:

$$\texttt{ImprintHitEntry} \equiv \exists\ \texttt{DE-input-of}\ .\ \texttt{ImprintAnnotation} \tag{4.1}$$

listed as part of Table 3.3 on page 84, provides an answer, by asserting that `ImprintHitEntry` is the only input to `ImprintAnnotation`. Thus, we know that adding `ImprintAnnotation` to our QV does not cause any incompatibility between the expected function input and the input data type. Note that we may use the same axiom to answer the complementary question: "what are all the valid input data types for a QV that includes `ImprintAnnotation` as one of its Annotation functions?" In other words, what are the data types to which `ImprintAnnotation` can be applied? In particular, given both `ImprintHitEntry` and `ImprintAnnotation`, we can verify that the former is a valid input for the latter.

In this particular example, such verification can be carried out by direct inspection of axiom (4.1). We can, however, generalize this to the case where (4.1) is inferred by DL reasoning, rather than being asserted. Following this intuition, we can formalize our first QV consistency constraint, as follows.

Definition 1 ***(DE-AF consistency)*** *Let $\mathcal{T}$ be the TBox consisting of all the axioms in the IQ ontology, and let $DE' \sqsubseteq DE$, $AF' \sqsubseteq AF$. We say that DE' and AF' are consistent with respect to property $DE\text{-}input\text{-}of$, iff*

$$\mathcal{T} \models DE' \sqsubseteq \exists\ DE\text{-}input\text{-}of\ .\ AF'$$

We use an example to illustrate the need for proper logical inference, as opposed to a simple syntactic lookup of axioms in the TBox. Consider the `DE` and `AF` hierarchies in Figure 4.3, with the axioms

$$\texttt{DE}_1 \sqsubseteq \exists\ \texttt{DE-input-of}\ .\ \texttt{AF}_1$$

$$\texttt{DE}_2 \sqsubseteq \exists\, \texttt{DE-input-of} \,.\, \texttt{AF}_2$$

that is, $\texttt{AF}_1$ takes input of type $\texttt{DE}_1$ and $\texttt{AF}_2$ takes input of type $\texttt{DE}_2$. It follows immediately from Def. 1 and $\texttt{DE}' \sqsubseteq \texttt{DE}_1$ that $\texttt{DE}' \sqsubseteq \exists\, \texttt{DE-input-of} \,.\, \texttt{AF}_1$, i.e., $\texttt{AF}_1$ and $\texttt{DE}'$ are consistent too; note how this conclusion requires an inference step. In this particular example, one could still work around the need for inference, by traversing the DE hierarchy from the bottom up. We argue, however, that this strategy would not be general enough. We will return to this point when presenting a general algorithm for constraint checking, in the next section.

It is important to note that the definition of QV consistency does not violate the rules for function polymorphism, given in Section 3.2.3. We repeat them here for convenience. Let f take input of type σ, let σ' be a sub-type of σ, and f' override f. Then:

1. both f and f' take input of type σ', and
2. f' takes input of type σ.

Note that, consistent with these rules, $\texttt{DE}'$ is *not* consistent with $\texttt{AF}_2$, since $\texttt{DE}' \sqsubseteq \exists\, \texttt{DE-input-of} \,.\, \texttt{AF}_2$ does not follow from the axioms.

Furthermore, consider function subclassing: $\texttt{AF}_1 \sqsubseteq \texttt{AF}$. According to the polymorphism rules, we expect AF to accept input of type $\texttt{DE}'$. Indeed, this follows immediately from the general inference pattern (a standard DL inference step):

$$\{(X \sqsubseteq \exists\, p \,.\, Y), (Y \sqsubseteq Y')\} \models X \sqsubseteq \exists\, p \,.\, Y'$$

that is:

$$\{(\texttt{DE}' \sqsubseteq \exists\, \texttt{DE-input-of} \,.\, \texttt{AF}_1), (\texttt{AF}_1 \sqsubseteq \texttt{AF})\} \models \texttt{DE}' \sqsubseteq \exists\, \texttt{DE-input-of} \,.\, \texttt{AF}$$

We can use Def. 1 to express our earlier two questions regarding feasible inputs to an Annotation function, as two complementary consistency problems:

1. **(AF consistency given DE:)** Let $\mathcal{T}$ be the TBox consisting of all the axioms in the IQ ontology, and $\texttt{DE}' \sqsubseteq \texttt{DE}$. Find all $\texttt{AF}' \sqsubseteq \texttt{AF}$ such that $\texttt{DE}'$ and $\texttt{AF}'$ are consistent with respect to DE-input-of, relative to $\mathcal{T}$;
2. **(DE consistency given AF:)** Let $\mathcal{T}$ be the TBox consisting of all the axioms in the IQ ontology, and $\texttt{AF}' \sqsubseteq \texttt{AF}$. Find all $\texttt{DE}' \sqsubseteq \texttt{DE}$ such that $\texttt{DE}'$ and $\texttt{AF}'$ are consistent with respect to DE-input-of, relative to $\mathcal{T}$.

Proceeding in a similar fashion, we could define consistency between the outputs of Annotation functions and a set of Quality Evidence types, and between the latter and the input to QA functions. Observing that these constraints are very similar to one another, however, it is more convenient to define them as instances of a general framework involving (i) generic properties and (ii) classes that belong to the domain and range of those properties. To begin, let us define two complementary functions, *requiredRanges*() and *domains*(), that generalize the domain and range of a property, as follows.

Definition 2 ***(required ranges of a property)*** *Let $\mathcal{T}$ be the TBox consisting of all the axioms in the IQ ontology, p a property and $C \sqsubseteq domain(p)$. We define $requiredRanges(p, C)$ to be the set of all classes X such that $\exists\, p\, .\, X$ subsumes C:*

$$requiredRanges(p, C) = \{X \sqsubseteq range(p) \mid \mathcal{T} \models C \sqsubseteq \exists\, p\, .\, X\}$$

We have chosen the name *requiredRanges* to emphasise that this is the set of all classes that are in the range of p, and furthermote *are known to be* the target of property p in the ontology, as opposed to the set of all classes that *may* be the target of p.

Definition 3 ***(domains of a property)*** *Let $\mathcal{T}$ be the TBox consisting of all the axioms in the IQ ontology, property p and $C \sqsubseteq range(p)$. We define $domains(p, C)$ to be the set of all classes X that are subsumed by $(\exists\, p\, .\, C)$:*

$$domains(p, C) = \{X \sqsubseteq domain(p) \mid \mathcal{T} \models X \sqsubseteq \exists\, p\, .\, C\}$$

Observe that DE-AF consistency (Def. 1) can also be written using *requiredRanges*():

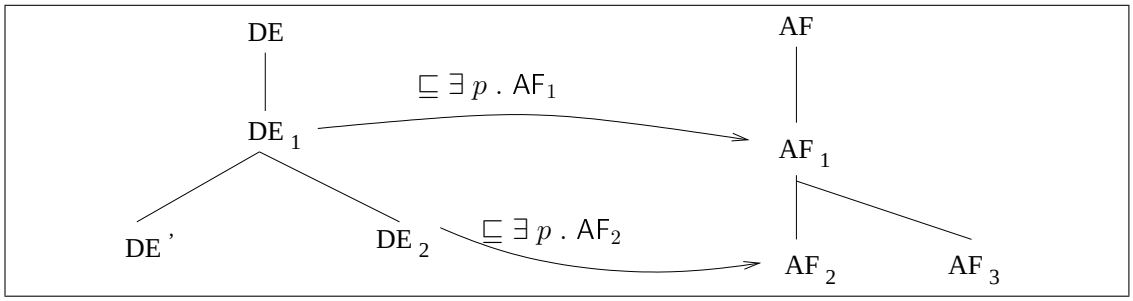


Figure 4.3: Fragments of the `DE` and `AF` hierarchies used to illustrate QV consistency constraints.

Definition 4 ***(DE-AF consistency by ranges)*** *$\texttt{DE}' \sqsubseteq \texttt{DE}$ and $\texttt{AF}' \sqsubseteq \texttt{AF}$ are consistent with respect to $\texttt{DE-input-of}$ iff $\texttt{AF}' \in$ requiredRanges$(\texttt{DE-input-of}, \texttt{DE}')$.*

Indeed, we can use *domains*() and *requiredRanges*() to provide a general formulation of all three consistency problems listed above, as follows:

Definition 5 ***(Consistency problems)***

Consistency verification: *let p be a property, $D \sqsubseteq domain(p)$ and $R \sqsubseteq range(p)$. Test whether $\mathcal{T} \models D \sqsubseteq \exists\ p\ .\ R$, i.e., whether $R \in requiredRanges(p, D)$;*

Find all consistent R given D and p: *this amounts to computing $requiredRanges(p, D)$;*

Find all consistent D given R and p: *this amount to computing $domains(p, R)$.*

The remaining consistency constraints for Quality Views can now be presented in terms of this framework.

Definition 6 ***(AF-QE consistency)*** *$\texttt{AF}' \sqsubseteq \texttt{AF}$ and $\texttt{QE}' \sqsubseteq \texttt{QE}$ are consistent with respect to $\texttt{AF-has-output}$ iff $\texttt{QE}' \in$ requiredRanges$(\texttt{AF-has-output}, \texttt{AF}')$.*

Definition 7 ***(QE-QA consistency)*** *$\texttt{QA}' \sqsubseteq \texttt{QA}$ and $\texttt{QE}' \sqsubseteq \texttt{QE}$ are consistent with respect to $\texttt{assertion-based-on-evidence}$ iff $\texttt{QA}' \in$ requiredRanges$(\texttt{assertion-based-on-evidence}, \texttt{QE}')$.*

To provide a concrete example, the Quality View presented earlier in this chapter (Section 4.2) is consistent. Indeed, let `ImprintHitEntry` be the data type for which the QV is specified. We can easily verify, using the axioms of Table 3.3 at page 84, that `ImprintHitEntry` and `ImprintAnnotation` are DE-AF consistent, `ImprintAnnotation` is AF-QE consistent with respect to `PeptidesCount`, `Masses`, `Mass`, `Coverage`; and `PIScoreClassifier` is QE-QA consistent with respect to each of these QE classes.

4.4.2 Checking consistency

With the definitions given so far, we have reduced the QV consistency problem to the problem of computing *domains*() and *requiredRanges*(). We now show that computing these functions is a straightforward application of DL reasoning.

Indeed, note that computing $domains(p, C)$ is accomplished directly by *querying* the reasoner. A query to a reasoner, relative to a TBox $\mathcal{T}$, consists of a concept C, either atomic or constructed. With this input, the reasoner returns the set of all the classes C' that are subsumed by C, i.e., $\{C' \mid \mathcal{T} \models C' \sqsubseteq C\}$ (these are also called the *inferred subclasses* of C). Thus, to compute $domains(p, C)$ we simply need to submit a query consisting of the constructed class $(\exists p \,.\, C)$. For example, with reference to Figure 4.3, we have $domains(\texttt{DE-input-of}, \texttt{AF}_1) = \{\texttt{DE}_1, \texttt{DE}', \texttt{DE}_2\}$. Note that, in general, if $C \in domains(R)$ then $C' \in domains(R)$ for any R and any C' such that $\mathcal{T} \models C' \sqsubseteq C$.

The algorithm for computing $requiredRanges(p, C)$ is slightly more involved, since the reasoner does not support this functionality directly. It is tempting to derive a simple algorithm from the example depicted in Figure 4.3. To compute $requiredRanges(\texttt{DE-input-of}, \texttt{DE}')$, we could first check whether there are relevant axioms of the form $\texttt{DE}' \sqsubseteq \exists\ \texttt{DE-input-of}\ .\ X$, and add X to the result. In this case, we would need to traverse the path from $\texttt{DE}'$ to the root, repeating the test at every step. This would yield $\texttt{DE}_1 \sqsubseteq \exists\ \texttt{DE-input-of}\ .\ \texttt{AF}_1$, so we would add $\texttt{AF}_1$ to *requiredRanges*(). Note that $\texttt{DE}_1$'s ancestors are also part of the result: in general, if $Y \in requiredRanges(p, X)$, then $Y' \in requiredRanges(p, X)$ for any Y' such that $\mathcal{T} \models Y \sqsubseteq Y'$.

The problem with this approach is that it only works with subsumption relationships that are *asserted*, rather than inferred. This means that the result will be correct but, in general, not complete: the algorithm will fail to consider all relationships $C_1 \sqsubseteq C_2$, where C_2 can be a constructed class, that are not asserted explicitly in the ontology but can be inferred, i.e., $\mathcal{T} \models C_1 \sqsubseteq C_2$. To clarify this point, consider the following new axiom:

$$\texttt{DE}_1 \sqsubseteq \exists\ \texttt{DE-input-of}\ .\ (\exists\ \texttt{AF-has-output}\ .\ \texttt{QE}_1)$$

This asserts, legitimately, that $\texttt{DE}_1$ is input to at least one Annotation function

that has $\texttt{QE}_1$ as one of its outputs. Now let the new class X be defined as follows:

$$\texttt{X} \equiv \exists\ \texttt{AF-has-output}\ .\ \texttt{QE}_2$$

and $\texttt{QE}_2 \sqsubseteq \texttt{QE}_1$. From these three axioms we infer

$$\texttt{DE}_1 \sqsubseteq \exists\ \texttt{DE-input-of}\ .\ \texttt{X}$$

that is, $\texttt{X} \in \mathit{requiredRanges}(\texttt{DE-input-of}, \texttt{DE}_1)$. However, the algorithm given above fails to find this solution.

This is an important point, as it highlights the difference between a semantic use of the ontology, i.e., one where we consider all relationships that are logically inferred from the axioms, and a *syntactic* one, where one only looks are the ontology schema (i.e., by navigating up and down an asserted hierarchy). The latter corresponds to using a logical model without axioms, for instance a relational or semi-structured data model, rather than a full-fledged ontology. Having argued, in the previous chapter, for a semantic model, the trade-off between the expressivity of such a model and the computational complexity associated with its practical use now becomes clear in the context of QV consistency checking.

The following algorithm for *requiredRanges*() takes inferred relationships into account. It makes use of *domains*(), following the observation that, given class C_2 and property p, by definition we have $C_1 \in \mathit{domains}(p, C_2)$ iff $\mathcal{T} \models C_1 \sqsubseteq \exists p\ .\ C_2$. Thus, the idea for computing $\mathit{requiredRanges}(p, C_1)$ is to descend down the subsumption relationships rooted at $\mathit{range}(p)$; for each class C_2 in this hierarchy, we check whether $C_1 \in \mathit{domains}(p, C_2)$. If this is the case, we add C_2 to $\mathit{requiredRanges}(p, C_1)$. The algorithm terminates at the bottom of the class hierarchy. Note that here we use the *inferred* hierarchy, i.e., the one containing all inferred subclassed of C_2 —this is computed by querying the reasoner with input C_2. Note also that, if C_2 is not in $\mathit{requiredRanges}(p, C_1)$, then no descendent of C_2 is in $\mathit{requiredRanges}(p, C_1)$, either. We use this property to prune the set of candidate classes that need to be tested. The algorithm is shown in Figure 4.4.2.

We can easily verify that this algorithm computes the same set $\mathit{requiredRanges}(\texttt{DE-input-of}, \texttt{DE}') = \{\texttt{AF}_1, \texttt{AF}\}$ as our syntactic algorithm above, as expected. In addition, however, it also correctly computes $\mathit{requiredRanges}(\texttt{DE-input-of}, \texttt{DE}_1)$, something that cannot be done with the syntactic approach. Here is a sketch of the algorithm in action when computing

```
computing requiredRanges(p, C):
input: property p, class C, TBox T
output: the set {X ⊑ range(p) | T ⊨ C ⊑ ∃ p . X}

        sol = ∅;
        Q = ∅; // Queue of candidate solutions
        Q.add(range(p)); initialize queue with root of ranges candidates hierarchy
        while (Q is not empty) {
                candidate = Q.remove();
                D = domains(p, candidate);
                if (C ∈ D) { // this means C ⊆ ∃p.candidate
                        sol = sol ∪ {candidate};
                        Q.add(subclassesOf(candidate)); inferred subclasses
                }
        }
```

Figure 4.4: Pseudo-code for the *requiredRanges*() algorithm.

requiredRanges(DE-input-of, $\mathtt{DE}_1$). Here X is defined as in the previous example: $\mathtt{X} \equiv \exists$ AF-has-output . $\mathtt{QE}_2$:

1. Initially, *candidate* = AF, and $\mathcal{D}$ = *subclassesOf*($\exists$ DE-input-of . AF) = $\{\mathtt{DE}_1\}$. Therefore, AF $\in$ *requiredRanges*(DE-input-of, $\mathtt{DE}_1$);

2. Next, observe that $X \in$ *subclasses*(AF), hence *candidate* = X. Since

$$\mathcal{D} = \mathit{subclassesOf}(\exists\ \mathtt{DE\text{-}input\text{-}of}\ .\ \mathtt{AF}) = \{\mathtt{DE}_1\},$$

   we conclude that $X \in$ *requiredRanges*(DE-input-of, $\mathtt{DE}_1$).

In terms of performance, the repeated calls to a reasoner (equal to the number of inferred subclasses of *range*(p) in the worst case) are the most expensive steps in the algorithm. This is a general and well-known issue with axiom-based ontologies. Nevertheless, the ontology-driven user interface for Quality View specification, described in the next section, makes use of this algorithm with good practical performance, thanks to the small size of the class hierarchies involved. In particular, since we can use *domains*() and *requiredRanges*() to provide support for all three consistency problems presented in Def. 5, we are able to provide consistency guarantees to users regardless of the order in which Quality Views elements are specified.

4.5 Supporting consistent Quality View specification in practice

We have put the theory developed so far into practice, to ensure the consistency of QVs that are specified by e-scientists. As suggested earlier, *domains*() and *requiredRanges*() can be used both to verify consistency constraints, and to find ontology classes that can be added to a partially defined QV to make it consistent. As part of the Qurator workbench, we have implemented a visual environment for the specification of QVs that provides consistency support using *domains*() and *requiredRanges*(). A screenshot of the QV definition GUI appears in Figure 4.5.

Using the three panes on the left of the figure, users may browse the Data Entity, Quality Evidence, and Quality Assertions hierarchies, while the panes on the right display the users's selections. Suppose that the users want to create a new QV for the `Imprint HitEntry` data type, using Quality Evidence types that have to do with proteomics. They may start by selecting the data type from the top left pane, and then proceed to select a Quality Evidence type, say `HitRatio`. In this case, the system verifies that `ImprintHitEntry` and `HitRatio` are consistent, by finding at least one Annotation function `AF′` such that `ImprintHitEntry` and `AF′` are DE-AF-consistent, and `AF′` and `HitRatio` are AF-QE-consistent. In the example, `ImprintAnnotation` fits the requirements. If no such function can be found, then the system alerts the user of the inconsistency (by highlighting the selected QE class).

Users, however, may not be familiar with the type of QEs that are available for the selected data type. In this case, they can request QE recommendations from the system rather than making a selection. In this case, the system computes the set of Annotation functions `AF′` such that `ImprintHitEntry` and `AF′` are DE-AF-consistent, and the set of `QE′` classes such that `AF′` and `QE′` are AF-QE consistent. Again, if either of these is the empty set, an inconsistency is reported (in this case, the only explanation is that no Annotation functions are available for the input data type). Finally, one may begin by selecting QE types, and letting the system determine a set of Annotation functions that compute values of those types.

We apply the same model to deal with QE-QA consistency. Thus, by proceeding backwards from QA functions, users may have the system determine the set of data types on which a fully specified QV can be computed. A further useful

Figure 4.5: Screenshot of the Quality View visual specification environment

usage pattern made possible by the user interface involves the specification of an input data type, as well as of one or more QA functions. The system in this case determines whether there exists a consistent combination of Quality Evidence types and Annotation functions such that the selected QA functions can be computed on the evidence retrieved for the input data.

Note that Annotation functions are not shown anywhere in the interface. This is done to simplify the specification process, under the assumption that two Annotation functions that satisfy the input / output requirements are functionally equivalent, and that either of them can be selected. In the current model we do not introduce any additional features of these functions that may further differentiate them. By letting users specify data and Quality Evidence requirements, the system will use any Annotation function that meets those requirements: in this case, only the result of the system selections is shown to the user. Users may, however, decide to change these selections. Since any of these changes may affect consistency, they trigger a re-evaluation of the relevant *domains*() and *requiredRanges*() sets.

Note also that *requiredRanges*(`ED-input-of`, `DE`′) may include multiple Annotation functions, some of which are subsumed by others. Recall that, in general, if `AF`′ ∈ *requiredRanges*(`ED-input-of`, `DE`′) then `AF`″ ∈ *requiredRanges*(`ED-input-of`, `DE`′) for all `AF`″ that subsume `AF`′. Intuitively, there is no point in considering all of these functions: by definition, the functionality provided by the more general ones is also provided by the more specific functions within the same hierarchy. Therefore, whenever *requiredRanges*() is computed, only the most specific functions are retained. In addition, note that some of the functions included in the same *requiredRanges*() set may not be in a specialization relationship, for example $\texttt{AF}_2$ and $\texttt{AF}_3$ in Figure 4.3. All such functions are retained for the purpose of performing QV consistency checks. By extension, we apply the same criteria to all other types of classes, other than functions, that are computed by the GUI using *requiredRanges*(). For example, `HitRatio` is retained as the most specific output of `ImprintAnnotator`, while `QE` is not. A complementary situation occurs with *domains*(): if $C_1 \in$ *domains*(), then $C_2 \in$ *domains*() whenever $\mathcal{T} \models C_2 \sqsubseteq C_1$. Therefore, in this case the *most specific* classes in the subsumption relationship are the most informative.

We conclude the presentation of the QV specification interface by noting that the bottom pane is dedicated to the specification of actions. Here the metaphor

of "channels" is used to denote quality classes associated to conditions. In particular, the interface allows for the definition of multiple splitter actions, each consisting of multiple channels. One condition is associated to each channel. To facilitate the definition of the conditions, the environment provides a graphical expression editor where only the variables that have been defined in the QV can be used. Furthermore, type checking is performed to ensure that any expression built using the editor is consistent with the scope of variables and with their types.

To summarize, this visual environment exposes the IQ ontology to QV designers, providing consistency guarantees that facilitate their work without requiring any understanding of semantic technology.

4.6 Summary and conclusions

In this chapter we have introduced the notion of Quality Views (QV), a specification of abstract quality processes that combine the functionality of multiple Quality Assertion functions. QVs are at the core of the IQ model and of the Qurator workbench; they embody the paradigm of operational quality described in Chapter 2, in that they use Annotation and QA functions to compute quality values from the data. We have provided both a simple syntax (based on XML) and a functional definition of Quality Views, by presenting a complete interpreter for QVs written in Haskell, a functional programming language.

Although QVs are specified using an XML-based syntax, in the last section we have described a visual environment for QV specification that hides the XML syntax from the user altogether. Furthermore, having defined the notion of QV consistency formally by means of OWL DL axioms, the specification environment incorporates an algorithm that exploits the functionality of a standard OWL DL reasoner to perform incremental consistency checks. This greatly facilitates the user's task of specifying consistent QVs, i.e., the "quality function selection" of the IQ lifecycle, by supporting the specification of consistent QVs by hiding from the user the complexity of the ontology model.

In summary, Quality Views represent a novel approach to modelling quality from the user perspective, and as such they advance the state of the art in the specification of personal quality criteria. In the next chapter we will show how

such formal definition makes it possible to automatically compile QVs into executable software components; this will demonstrate a practical implementation of the "quality assessment" step of the IQ lifecycle.

Chapter 5

Quality Views as workflows

In the previous chapter we have introduced Quality Views as functions that associate Quality Assertions to the input data. We now present an automated translation of QVs into executable software components. Existing e-science infrastructures, such as ^{my}Grid and other Grid middleware [WAH+07], follow a service-oriented approach whereby data access and analysis tools are realized as services, and experiments are defined as a composition of those services. Following this view, we are going to translate Quality Views into services, specifically Web services, so that they can integrate easily within the e-science infrastructure: once a QV is a service, it can be used as part of an *in silico* experiment like any other component, effectively becoming a commodity.

There are several options for compiling a QV into a Web service. Note however that QVs are themselves simple compositions of more elementary services, the quality functions. Thus, if we assume that quality functions are implemented as Web services, then we can generate a QV service from an abstract QV specification by using service composition operators.

Workflow models provide the operators we need: we can define a QV service as a workflow, whose elementary tasks include Quality Assertion services, as well as Annotation services. From the technical standpoint, this choice has the main advantage that workflow models are described using a high-level, declarative specification language, making it relatively simple to translate into from a higher-level language; we call the resulting service a *quality workflow*. At the same time, this approach results in a white-box implementation of quality services, that users with only moderate programming expertise can hopefully understand more easily than a black-box service implementation (users can also edit and modify a

compiled quality workflow directly, if needed).

One additional advantage of this approach is that workflows can be easily integrated with other workflows using few graph manipulation primitives. This ease of integration makes it possible to add quality workflows to any e-science experiment that is itself described as a workflow, an increasingly realistic assumption for the e-science paradigm.

A number of workflow models and execution engines are freely available, both for the scientific community (e.g. Kepler [LABe05, MBL06], Triana [CGH+06, TSWH07]) and for the business domain (e.g. BPEL [EBC+05]). In our implementation we have chosen Taverna (`taverna.sourceforge.net`), a workbench for the design and execution of scientific workflows [OAF+04, HWS+06]. The main consideration that makes Taverna appealing as a specific target is a practical one. By making it very easy for third-party organizations to add services that can be used as workflow processors (any distributed Web Service can in principle become a processor), the Taverna model has gained considerable popularity, especially within the life sciences community. This makes the choice of Taverna interesting in terms of potential impact of the technology solutions proposed in this work.

It is important to note that the choice of workflow technology does not limit the application of Quality Views to workflow-based experiments: we can still view a quality workflow as a service (that requires a workflow engine to execute), and use it as part of any service-oriented computing infrastructure. In Chapter 6 we provide an example of this "black-box" usage, when we propose quality-enhanced XML queries. As we will see, the only required computing infrastructure for this is the ability for an XQuery processor to call an external service.

With this motivation, the focus of this chapter is on the automated translation of Quality Views into Taverna workflows. We proceed in steps, as follows. We begin in Section 5.2 with an informal description of the structure of quality workflows. This is followed in Section 5.3 by the definition of the structural operational semantics of quality workflows, using the formal model proposed by Turi *et al.* [TMR+07]. With this, we can finally define the formal translation rules from QVs to workflow (Section 5.4). Intuitively, these rules should result in a workflow that is consistent with the QV semantics given earlier, in Chapter 4. Indeed, we show this formally in Section 5.5.

Finally, in Section 5.6 we return to the more practical aspects of Quality View

application, to show how a Quality workflow resulting from the translation process can be embedded into a pre-existing Taverna workflow, using simple graph editing primitives. The chapter concludes by presenting the standalone and embedded Quality workflows for our proteomics running example.

To help illustrate each of these steps, we use an example scientific workflow that involves our protein identification use case. We begin with a description of the example, in the next section.

5.1 A scientific workflow for the proteomics example

In Chapter 4 we have defined a Quality View that formalizes our proteomics quality example, first introduced in Chapter 1. The example continues here with the description of a workflow designed to discover the set of proteins that are expressed by particular organisms or cells [AM03]. To recap briefly, the lab technique used to analyze protein spots is peptide mass fingerprinting (PMF). In PMF, a cell sample is processed in the lab using a mass spectrometer, resulting in a representation of the proteins contained in the sample as a list of individual masses, called a *peak list*. The data-intensive portion of the experiment involves using the peak list to search a reference database of known proteins. Ideally, the search returns a list of proteins that are present in the original sample.

As mentioned in the introduction, this type of experiment is subject to various types of error, which manifest themselves in the presence of false positives in the list of proteins reported by the match. When this process is part of a more complex workflow, these errors are likely to propagate to other processes, possibly leading to misleading results. Consider for example a follow-up *in silico* experiment designed to exploit the results of a protein identification process, as specified by the ISPIDER project [BEF+05] on proteomic data integration. A scientist trying to understand the behaviour of a cell under particular circumstances performs a PMF experiment, from which many identifications result. Rather than the identifications *per se*, however, the scientist is more interested in the functional roles of the proteins within the cell. The identified proteins are therefore mapped to descriptions of their biomolecular function, by querying the GOA database, which links protein accession numbers with terms describing molecular function, expressed using terms from the Gene Ontology (GO) (www.geneontology.org), a

standard controlled vocabulary. Thus, in this example data errors introduced at one step, i.e., false positives, propagate to downstream steps, making the results of the overall functional analysis swamped by noise.

An implementation of this process as a Taverna workflow, from protein identification to the functional analysis of those proteins, is shown in Figure 5.1.

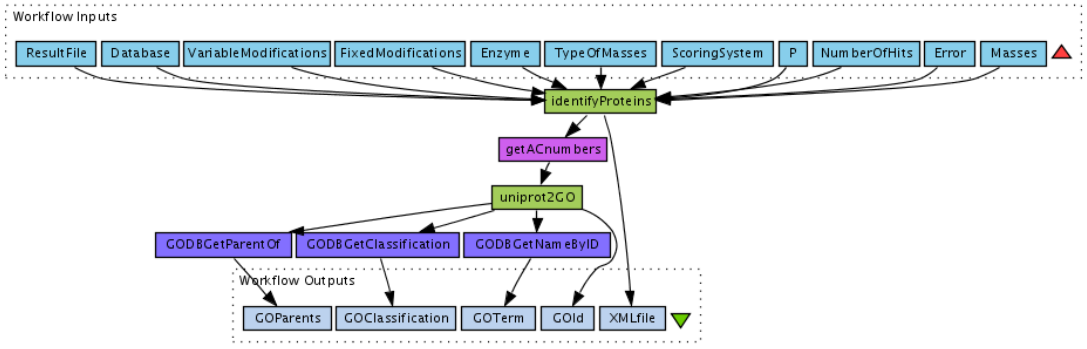


Figure 5.1: Example Proteomics Analysis Workflow

In the first step, represented by the "IdentifyProtein" box, a set of peak lists are retrieved from the Pedro database and used for protein identification, using the Imprint analysis tool along with some configuration parameters and the reference protein sequence database. Imprint computes ranked identifications, along with additional indicators; in our example, we will use *Hit Ratio* (HR) and *Mass Coverage* (MC), which we have already introduced as Quality Evidence types in the IQ ontology. To recall, HR gives an indication of the signal to noise ratio in a mass spectrum, and MC measures the amount of protein sequence matched [SPB06]. Finally, the GOA database is queried to retrieve the functional annotations for each identified protein. At this point, scientists have the necessary elements to draw their conclusions to interpret the biological process within the cell.

The QV defined in the previous chapter provides an abstraction for describing quality filters that (i) provide an estimate of the likelihood that a protein in the list is a false positive, and (ii) provide functionality to filter out, either automatically or interactively, the undesired elements. In the rest of this chapter we will see how the workflow of Figure 5.1 can be augmented with a "quality sub-workflow" that represents an executable Quality View.

5.2 Quality workflows

Let us begin with an informal description of quality workflows. First, we briefly recall the Taverna workflow model. A Taverna workflow consists of a collection of processors with data and control links among them. Processors may have multiple inputs and outputs, called *ports*; a data link establishes a dependency between the output port of a processor and the input port of another. A control link indicates that a processor can begin its execution only after some other processor has successfully completed its execution. Processors are implemented either as local Java classes, or as Web Services, with input and output ports that correspond to the operations parameters defined in the service WSDL interface. The data that flows over the links is encapsulated within XML messages. The workflow execution engine, called FreeFluo, schedules the invocation of the service operations, making sure that the dependencies are not violated, and manages the flow of data between the processors.

Figure 5.2 shows the result of compiling our example QV (Figure 4.1 in the previous chapter) into a workflow. This can be viewed as an instance of a generic Quality workflow that includes three types of processors, corresponding to Annotation functions, Quality Assertion functions, and Actions, respectively. The processors are organized into layers, interconnected using data and control links as shown in the figure. A number of ancillary processors, used to supply constant values to the main processors, are omitted from the figure to avoid clutter.

To understand this workflow structure and its behaviour[1], let us analyse the operations required during the execution of a QV. Initially, the QV receives an input dataset along with a set of actual parameters. The `FetchParameters` processor is responsible for binding actual parameters values to the formal parameters names declared as part of the Annotation and Assertion functions.

The next step in the workflow execution is to compute the values of quality evidence, as required by the Assertion functions. Here we need to distinguish between short-lived quality evidence that is computed when the quality workflow is executed, as in the proteomics example; and long-lived evidence that is computed ahead of time, as in the Uniprot annotation example briefly described in the first chapter (Section 1.3.2). This is useful when annotations involve long computations, which may even include user interaction. While the former type requires

[1]We will see in Section 5.5 that the workflow behaviour is consistent with the functional description of abstract Quality Views given in Chapter 4.

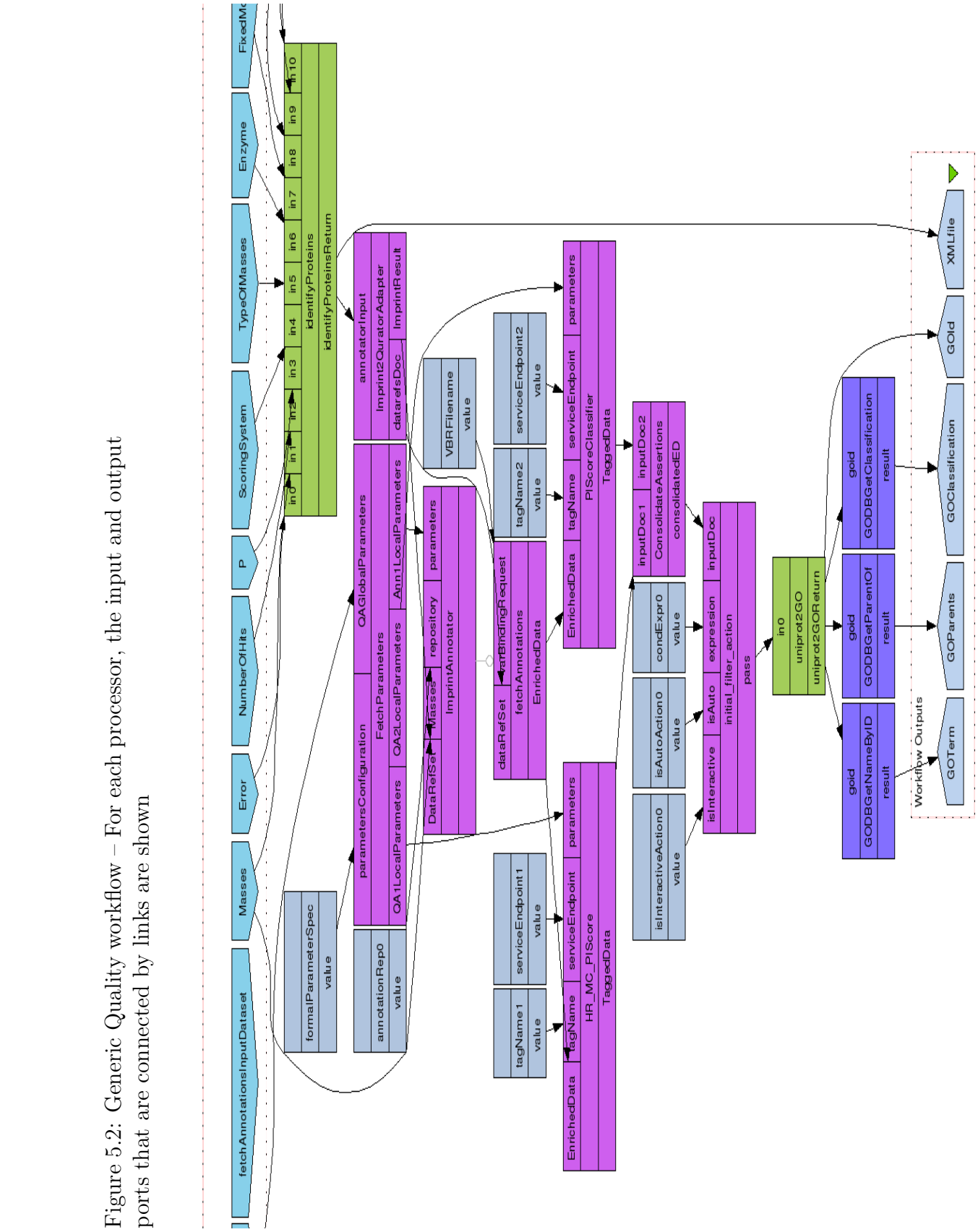


Figure 5.2: Generic Quality workflow – For each processor, the input and output ports that are connected by links are shown

Annotation processors as part of the workflow, the latter type do not, because evidence is stored ahead of time and persistently in the evidence repository

To account for the possibility that there are no Annotation functions in the QV, this part of the workflow includes a single `fetchAnnotations` processor, that is responsible for reading quality evidence values from a persistent repository. If there are no Annotation services, then this is all that is needed. When they are present, Annotation services must write the computed quality evidence values to the repository. Instead of being connected to the rest of the workflow using data links, they are all connected using control links to the singleton `fetchAnnotations` processor. The control link ensures that the Annotation processors and the `fetchAnnotations` processor are executed sequentially. Thus, this workflow configuration implements a multiple writers-single reader pattern where the writer and reader processors are synchronized by the workflow engine. It is easy to see that, in this pattern, long-lived annotations are just a special case.

As stated in the preceding chapter (Section 4.2.3), annotation functions are further subject to the requirement that the Quality Evidence variables managed by each Annotator be disjoint. This requirement, enforced by the compiler (please see Section 5.4 below), ensures that Annotators do not override each other's values, which would result in inconsistencies.

As a next step in workflow execution, the Quality Evidence values retrieved by `fetchAnnotations` are forwarded to each of the QA processors. The execution of these processors produces the Quality Assertion values, which are then consolidated into a single data structure by the `ConsolidateAssertions` processor. Note that this data structure is an implementation of the QV environment, as defined in Section 4.3.

The next and final step in the QV involves Actions. As mentioned in Section 4.1, the QV syntax currently supports n-way Splitter Actions, which assign each data element to one of n possible quality classes. In the Quality workflow, a Splitter Action is rendered using elementary Filter processors, i.e., 2-way Splitters that assign data to classes "accept" or "reject". Filters are implemented as Taverna processors that receive the environment and evaluate the conditions associated to the action. Optionally, at this point, the QV designer may have indicated that Actions be performed interactively. In this case, the processor presents the contents of the environment through a graphical interface to the user, who performs a manual filtering of the data elements.

5.3 Formal syntax and semantics of Quality workflows

We now formalize the intuitive description just given. In turn, this allows us to formulate the translation rules of QV into workflows in a precise way (in Section 5.4), and to show the correctness of the translation process (Section 5.5).

We begin by introducing the formal notation for describing Taverna processors and workflows, used in [TMR+07]. We then define the specific structure of quality workflows, by giving composition rules for transforming the inputs into the outputs. These are both syntax rules, describing data types transformations, and semantic rules that define the structure of complex workflows in terms of the structure of its components, inductively [2].

5.3.1 Notation for the Taverna workflow language

The Taverna language is defined using the *computational lambda calculus* [Mog91]. The use of lambda calculus is motivated by the fact that the Taverna workflow language can be defined in functional terms, although it uses Web Services as its building blocks. The use of functional languages to give formal meaning to workflows is not new; an example is the use of the Haskell functional programming language to give a formal definition of a particular workflow for the Ptolomey II system [LA03] (a precursor to Kepler). The computational lambda calculus is obtained by augmenting the lambda calculus with suitable *monads* [Wad90, Wad95] to model real-life behaviour of functional programs. The list operator, mapping a set A to the set $L(A)$ of all lists formed with elements of the set A, is one such monad.

Types

Taverna represents all domain-specific data types used in bioinformatics applications as strings. These include complex types such as those represented by XML documents. We denote the string data type with $\mathbf{s}$. One can construct arbitrarily nested lists starting from the base types, i.e., $L(\mathbf{s})$, $L^2(\mathbf{s})$, etc.[3] Taverna also allows for multiple inputs and outputs, hence products have also to be included.

[2] We would like to give credit to D. Turi for the definition of the syntax and semantic rules of the Taverna language.

[3] We write L^n for n applications of L.

For instance: $\mathbf{s} \times \mathbf{s}$, $\mathbf{s} \times L^3(\mathbf{s})$, $L^2(\mathbf{s}) \times \mathbf{s} \times \mathbf{s} \times L(\mathbf{s})$. We use the special symbol $\perp$ when a processor has no output. Formally:

$$\tau ::= \mathbf{s} \mid L(\tau) \mid \tau \times \tau \mid \perp$$

We are going to use σ and τ to denote types. For example, in a Quality workflow the input dataset is a list of data references, formatted as an XML document, which is formally represented here as a list of strings, $L(\mathbf{s})$, while the Quality Evidence annotations retrieved by the `fetchAnnotations` processor are represented as triples of the form $\langle name, class, value \rangle$, of type $\mathbf{s} \times \mathbf{s} \times \mathbf{s}$, denoted $\mathbf{s}^3$.

Contexts

A context Γ defines the unbound typed input variables of a processor, and, by extension, of a workflow. It is described as a list of (typed) inputs:

$$\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$$

where $x_1, \ldots, x_n$ are input variables of type $\sigma_1, \ldots, \sigma_n$. Given context Γ above, we write $\Gamma, x : \sigma$ to denote the context $x_1 : \sigma_1, \ldots, x_n : \sigma_n, x : \sigma$. Contexts can be empty, i.e., n can be 0. We write $Type(x_i) = \sigma_i$ to denote the type of variable x_i.

For example, we can write $D : L(\mathbf{s})$ to denote the typed input dataset to our Quality workflow, and $annot : \mathbf{s} \times \mathbf{s} \times \mathbf{s}$ for the output of the `fetchAnnotations` processor.

Workflows and Processors

To represent workflows we use the following *sequent* notation:

$$\Gamma \vdash W : \tau \tag{5.1}$$

where variable W denotes a generic workflow, context Γ represents the workflow's input variables, and the output of W is of type τ. Function *Type* is extended naturally from variables (i.e., workflow inputs) to workflow outputs. Thus, in (5.1), $Type(W) = \tau$. A workflow consists of a composition of processors, obtained by linking the outputs and inputs of the component processors. We will use *sequent calculus* to describe the composition rules for quality workflows. The

calculus is based on *axioms* of the form

$$\Gamma \vdash \mathsf{p} : \tau$$

Note that the axiom involves one particular processor p, for example the following axiom:

$$D : L(\mathbf{s}) \vdash \mathsf{fetchAnnotations} : \mathbf{s} \times \mathbf{s} \times \mathbf{s} \tag{5.2}$$

defines a processor fetchAnnotations with input D, a list of strings, and three string outputs, represented by the product types. Note also that a processor is a special case of a workflow.

Finally, the axiom:

$$\vdash \mathsf{foo} : \mathbf{s}$$

denotes a processor with no input, i.e., a constant value of type string. We use the convention that the name of the processor is also the value of the output, i.e., the output of the foo processor is the string "foo".

Syntax rules and Structural Operational Semantics

We describe workflows using type inference and semantic rules. Type inference rules define the input and output types of workflows obtained by application of some composition operator. Consider for example the *pairing* operator $\langle P, Q \rangle$, used to model the concurrent execution of processors P and Q that have no data or control dependencies amongst them. The typing rule for this operator is the following:

$$\frac{\Gamma \vdash P : \sigma \quad \Gamma \vdash Q : \tau}{\Gamma \vdash \langle P, Q \rangle : \sigma \times \tau} \tag{5.3}$$

This rule defines the output type of a workflow obtained by pairing P and Q, given the output types of P and Q.

The semantics rules for Taverna follow from the general computational lambda calculus theory [Abr90]. They are *structural* [Plo81] in the sense that they describe the structure of complex workflows in terms of their components. It is important to realize that Taverna processors are black-box functions (implemented as Web services), therefore we cannot formally describe how the workflow inputs are transformed into the outputs. Instead, the structural operational semantics used here gives rules for describing, inductively, the inputs and outputs of a compound workflow, given those of its components. The semantic rules for atomic

processors are axioms in this theory, and represent the base of the induction.

As an example, and a way to introduce the notation, let us consider the type inference rule for pairing, above. Its corresponding structural semantic rule is the following:

$$\frac{P \Downarrow u \quad Q \Downarrow v}{\langle P, Q\rangle \Downarrow \langle u, v\rangle} \tag{5.4}$$

We read this rule as follows: if the execution of two workflows P and Q terminates successfully with outputs u and v, respectively, then the workflow obtained by pairing: $\langle P, Q\rangle$ terminates with output $\langle u, v\rangle$. Note that the output value is consistent with the typing rule (Eq. 5.3).

The general notation used for the semantics is the following. Let P be a processor defined by the sequent:

$$\Gamma \vdash P : \tau$$

where $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n, x : \sigma$ are the input variables. Then $P[v_1/x_1 \ldots v_n/x_n] \Downarrow u$ denotes the successful execution of P, where each input variable x_i is bound to value v_i, and u is the output value. Furthermore, the rule requires the workflow to be *closed*, that is, all the input variables must be bound.

Note also that we expect to have a 1-1 correspondence between type and semantic rules, and that the output specified by the semantic rule must be consistent with the expected type.

Rather than describing all composition operators and their corresponding rules here, we are going to introduce them as needed, during the course of our description. However, as a reference, Table 5.1 (adapted from [TMR+07]), summarizes all the syntax and semantic composition rules for the Taverna language.

5.3.2 Quality workflows processor types

To begin the description of quality workflows, let us recall the definition of the core data types used by the workflow, given in Section 5.2. A dataset D is a list of strings (references to data, or *datarefs*): $D : L(\mathbf{s})$. Data annotations are triples of the form $\langle name, class, value\rangle$, of type $\mathbf{s} \times \mathbf{s} \times \mathbf{s} = \mathbf{s}^3$. Thus, we use type $\mathbf{s} \times L(\mathbf{s}^3)$ to represent the association of annotations to a dataref. For example, $\langle d, \langle \mathrm{C}, \mathrm{q{:}coverage}, 32.5\rangle\rangle$ denotes an annotation with name C, type "q:coverage" (a reference to an IQ ontology class) and value 32.5, associated to dataref d.

Table 5.1: Summary of Taverna syntax and structural semantics

Composition rule	**Syntax**	**Semantics**
Pairing	$\frac{\Gamma\vdash P{:}\sigma \quad \Gamma\vdash Q{:}\tau}{\Gamma\vdash\langle P,Q\rangle{:}\sigma\times\tau}$	$\frac{P\Downarrow u \quad Q\Downarrow v}{\langle P,Q\rangle\Downarrow\langle u,v\rangle}$
Projection	$\frac{\Gamma\vdash P{:}\sigma\times\tau}{\Gamma\vdash \mathtt{fst}(P){:}\sigma} \quad \frac{\Gamma\vdash P{:}\sigma\times\tau}{\Gamma\vdash \mathtt{snd}(P){:}\tau}$	$\frac{P\Downarrow\langle u,v\rangle}{\mathtt{fst}(P)\Downarrow u} \quad \frac{P\Downarrow\langle u,v\rangle}{\mathtt{snd}(P)\Downarrow v}$
Simple composition	$\frac{\Gamma\vdash P{:}\sigma \quad \Gamma,x{:}\sigma\vdash Q{:}\tau}{\Gamma\vdash \mathtt{let}\ x\leftarrow P\ \mathtt{in}\ Q{:}\tau}$	$\frac{P\Downarrow u \quad Q[u/x]\Downarrow v}{\mathtt{let}\ x\leftarrow P\ \mathtt{in}\ Q\Downarrow v}$
Iterative composition	$\frac{\Gamma\vdash P{:}L(\sigma) \quad \Gamma,x{:}\sigma\vdash Q{:}\tau}{\Gamma\vdash \mathtt{let}\ x\leftarrow P\ \mathtt{in}\ Q{:}L(\tau)}$	$\frac{P\Downarrow\vec{u} \quad \{Q[u_i/x]\Downarrow v_i\}_{i=1..\lvert\vec{u}\rvert}}{\mathtt{let}\ x\leftarrow P\ \mathtt{in}\ Q\Downarrow\vec{v}}$
Wrapped composition	$\frac{\Gamma\vdash P{:}\sigma \quad \Gamma,x{:}L(\sigma)\vdash Q{:}\tau}{\Gamma\vdash \mathtt{let}\ x\leftarrow P\ \mathtt{in}\ Q{:}\tau}$	$\frac{P\Downarrow u \quad Q[[u]/x]\Downarrow v}{\mathtt{let}\ x\leftarrow P\ \mathtt{in}\ Q\Downarrow v}$
Flattening	$\frac{\Gamma\vdash P{:}L^2(\tau)}{\Gamma\vdash \mathtt{flatten}(P){:}L(\tau)}$	$\frac{P\Downarrow[[w_{11},..,w_{1m}],..,[w_{n1},..,w_{nm}]]}{\mathtt{flatten}(P)\Downarrow[w_{11},..,w_{ij},..,w_{nm}]}$
Cross product	$\frac{\Gamma\vdash P_1{:}L(\sigma_1) \quad \Gamma\vdash P_2{:}L(\sigma_2) \quad \Gamma,x_1{:}\sigma_1,x_2{:}\sigma_2\vdash Q{:}\tau}{\Gamma\vdash \mathtt{let}\ x_1\ltimes x_2\leftarrow P_1\ltimes P_2\ \mathtt{in}\ Q{:}L^2(\tau)}$	$\frac{P_1\Downarrow\vec{u} \quad P_2\Downarrow\vec{v} \quad \{Q[u_i/x_1][v_j/x_2]\Downarrow w_{ij}\}_{j=1..m}^{i=1..n} \quad \lvert\vec{u}\rvert{=}n \ \ \lvert\vec{v}\rvert{=}m}{\mathtt{let}\ x_1\ltimes x_2\leftarrow P_1\ltimes P_2\ \mathtt{in}\ Q \quad \Downarrow[[w_{11},..,w_{1m}],..,[w_{n1},..,w_{nm}]]}$
Dot product	$\frac{\Gamma\vdash P_1{:}L(\sigma_1) \quad \Gamma\vdash P_2{:}L(\sigma_2) \quad \Gamma,x_1{:}\sigma_1,x_2{:}\sigma_2\vdash Q{:}\tau}{\Gamma\vdash \mathtt{let}\ x_1\odot x_2\leftarrow P_1\odot P_2\ \mathtt{in}\ Q{:}L(\tau)}$	$\frac{P_1\Downarrow\vec{u} \quad P_2\Downarrow\vec{v} \quad \lvert\vec{u}\rvert=\lvert\vec{v}\rvert \quad \{Q[u_i/x_1,v_i/x_2]\Downarrow w_i\}_{i=1..\lvert\vec{u}\rvert}}{\mathtt{let}\ x_1\odot x_2\leftarrow P_1\odot P_2\ \mathtt{in}\ Q\Downarrow\vec{w}}$

We use three types of processors in Quality workflows, namely Annotation, Quality Assertion, and Actions. A generic **Annotation processor** A is defined by the sequent:

$$D : L(\mathbf{s}) \vdash A : L(\mathbf{s} \times L(\mathbf{s}^3)) \tag{5.5}$$

A computes a list $\vec{a}$ of annotations for each input dataref $d \in D$, and associates them to d, resulting in the pair $\langle d, \vec{a} \rangle$. The corresponding semantic rule is:

$$\mathsf{A}[\vec{d}/D] \Downarrow [\langle d_1, \vec{a_1} \rangle, \langle d_2, \vec{a_2} \rangle \dots] \tag{5.6}$$

where each $\vec{d} = [d_1, d_2 \dots]$ and $\vec{a_i}$ is of type $L(\mathbf{s}^3)$.

A generic **Quality Assertion** processor QA takes input y of type $L(\mathbf{s} \times L(\mathbf{s}^3))$, i.e., a list of datarefs, each with an associated list of annotation triples. Formally:

$$y : L(\mathbf{s} \times L(\mathbf{s}^3)) \vdash QA : L(\mathbf{s} \times L(\mathbf{s}^3)) \tag{5.7}$$

with semantics:

$$QA[\vec{w}/y] \Downarrow \vec{v} \tag{5.8}$$

As we can see, the output is structurally identical to the input. However, the output value $\vec{v}$ is obtained by appending to each input annotation list in the input $\vec{w}$ one new Quality Assertion triple.

The definition of Action processors deviates slightly from the one given in Section 5.2. We model the output ports as logical quality classes (of type $\mathbf{s}$), and define a generic **Quality Test**, QT_k, $k : 1 \dots r$, each responsible for assigning a dataref d to one quality class, as follows:

$$w : \mathbf{s} \times L(\mathbf{s}^3) \vdash QT_k : \mathbf{s} \times \mathbf{s} \tag{5.9}$$

with the following semantics:

$$QT_k[\langle d, \vec{a} \rangle / w] \Downarrow \langle d, c \rangle \tag{5.10}$$

where c is a quality class label. Note that the result of applying r independent Quality Test processors is a list of lists of class-labelled datarefs, for instance $[\langle d_1, \text{“accept”} \rangle, \langle d_2, \text{“reject”} \rangle]$ and $[\langle d_1, \text{“green”} \rangle, \langle d_2, \text{“red”} \rangle]$ (so that d_1 is both “accept” and “green”, etc.).

In addition, we define one further processor, called Merger, to account for

the `fetchAnnotation` as well as the `AssertionConsolidation` Taverna processors. Merger consolidates multiple lists of annotation triples, each computed by a different annotation processor for the same input dataset, into a single list of annotations:

$$x_1 : \mathbf{s} \times L(\mathbf{s}^3) \ldots x_n : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{Merger} : \mathbf{s} \times L(\mathbf{s}^3)$$

Note that we are going to assume the use of data links between Annotators and `fetchAnnotation`, rather than control links; this is consistent with the last remark in the previous section, i.e., that control links in Taverna can be implemented using only data links.

5.3.3 Composition rules for quality workflows

The axioms given above define processor types. A specific quality workflow consists of a composition of processor instances described by these types, and corresponding to Quality View functions. In this section we give the syntax and structural semantic rules for quality workflows.

Let us assume that the Quality View includes n Annotation functions, m Quality Assertion functions, and r condition-action pairs. We denote these as A_i, $i : 1 \ldots n$; QA_j, $j : 1 \ldots m$; and QT_k, $k : 1 \ldots r$, respectively. As a further matter of notation, we use curly brackets to denote a collection of multiple processors of the same type that take part in the same rule, for example:

$$\{x_i : \sigma \vdash \mathsf{P}_i : \tau\}_{i:1\ldots n}$$

denotes n structurally identical processors. Similarly, the following semantic rule:

$$\{\mathsf{P}_i[v_i/x_i] \Downarrow y_i\}_{i:1\ldots n}$$

denotes the execution of the n processors, with the corresponding variable bindings. We also write $\vec{x}$ as a shorthand to denote a value of type list.

Merging Quality Evidence annotations

As suggested above, we use a Merger processor to account for the `fetchAnnotation` function. In the following rule, we model a data link from each of the n Annotation processors, to the singleton Merger processor. For this, we use a combination of

two operators, **iterative composition** and **dot product**. Let us define these first.

The iterative composition operator is used to compose two workflows P with output $L(\sigma)$ and Q, where Q's input includes $x : \sigma$. Note the cardinality mismatch between the output of P, a list, and the input of Q. The following syntax rule:

$$\frac{\Gamma \vdash P : L(\sigma) \quad \Gamma, x : \sigma \vdash Q : \tau}{\Gamma \vdash \texttt{let}\ x \leftarrow P\ \texttt{in}\ Q : L(\tau)} \tag{5.11}$$

is interpreted using an iteration, as follows: each value in the output list of P in turn is bound to input x of Q, and Q is then executed. At the end of P's output list (with elements of type τ), we have a complete output, of type $L(\tau)$.

The corresponding semantic rule defines the relationships among the values. Let P terminate with output $\vec{u} = [u_1, \ldots, u_n]$, and v_i be the output of Q when its input variable x is bound to u_i. Then, the output of $\texttt{let}\ x \leftarrow P\ \texttt{in}\ Q$ is $\vec{v} = [v_1, \ldots, v_n]$. Formally:

$$\frac{P \Downarrow \vec{u} \quad \{Q[u_i/x] \Downarrow v_i\}_{i=1..|\vec{u}|}}{\texttt{let}\ x \leftarrow P\ \texttt{in}\ Q \Downarrow \vec{v}} \tag{5.12}$$

The second operation required for Merger is the dot product. Its syntax rule is as follows:

$$\frac{\Gamma \vdash P_1 : L(\sigma_1) \quad \Gamma \vdash P_2 : L(\sigma_2) \quad \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash Q : \tau}{\Gamma \vdash \texttt{let}\ x_1 \odot x_2 \leftarrow P_1 \odot P_2\ \texttt{in}\ Q : L(\tau)} \tag{5.13}$$

We see that this is an application of the previous operator: since P_1 and P_2 produce list types, Q is executed repeatedly on this input in order to produce the final list of type $L(\tau)$. Consider the corresponding semantic rule:

$$\frac{P_1 \Downarrow \vec{u} \quad P_2 \Downarrow \vec{v} \quad |\,\vec{u}\,| = |\,\vec{v}\,| \quad \{Q[u_i/x_1, v_i/x_2] \Downarrow w_i\}_{i=1..|\vec{u}|}}{\texttt{let}\ x_1 \odot x_2 \leftarrow P_1 \odot P_2\ \texttt{in}\ Q \Downarrow \vec{w}} \tag{5.14}$$

Here we take one element from each of the two inputs, namely u_i and v_i, bind them to Q' input variables x_1 and x_2, and execute Q; we repeat this for the length of the input lists (note that we require that their lengths be the same).

With these operators we can now write the rules for the Merger operator. This is the syntax rule:

$$\frac{\{D : L(\mathbf{s}) \vdash \mathsf{A}_i : L(\mathbf{s} \times L(\mathbf{s}^3))\}_{i:1\ldots n} \quad x_1 : \mathbf{s} \times L(\mathbf{s}^3) \ldots x_n : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{Merger} : \mathbf{s} \times L(\mathbf{s}^3)}{D : L(s) \vdash \texttt{let}\ x_1 \odot \ldots \odot x_n \leftarrow \mathsf{A}_1 \odot \ldots \odot \mathsf{A}_n\ \texttt{in}\ \mathsf{Merger} : L(\mathbf{s} \times L(\mathbf{s}^3))} \tag{5.15}$$

Each element in the output of each Annotation processor is a pair $\langle \texttt{dataref}, \texttt{annotations} \rangle$. We also assume that the datarefs appear *in the same order* in each of these lists. This means that, at any one iteration, Merger will process all annotations for the same dataref: each input x_i is bound to one element $\langle d, \vec{a}_i \rangle$ for some d, and Merger coalesces all annotations $a_1 \ldots a_n$ for that d. The semantic rule defines this behaviour:

$$\frac{\{\mathsf{A}_i[\vec{d}/D] \Downarrow \vec{a_i}\}_{i:1\ldots n} \qquad \mathsf{Merger}[\langle d_j, \vec{a}_1 \rangle / x_1, \langle d_j, \vec{a}_2 \rangle / x_2 \ldots] \Downarrow \langle d_j, \vec{a}'_j \rangle}{(\texttt{let}\ x_1 \odot \ldots \odot x_n \leftarrow \mathsf{A}_1 \odot \ldots \odot \mathsf{A}_n\ \texttt{in}\ \mathsf{Merger})[\vec{d}/D] \Downarrow [\langle d_1, \vec{a'}_1 \rangle, \langle d_2, \vec{a'}_2 \rangle \ldots]} \tag{5.16}$$

We write WF_1 to denote the workflow resulting from this composition, up to this point.

Computing Quality Assertions

WF_1 is then composed with a QA processor QA, using a simple composition operator. This operator is a simpler version of iterative composition, which applies when there is no type cardinality mismatch among the outputs and inputs of connected processors, as illustrated by the following rules:

$$\frac{\Gamma \vdash P : \sigma \quad \Gamma, x : \sigma \vdash Q : \tau}{\Gamma \vdash \texttt{let}\ x \leftarrow P\ \texttt{in}\ Q : \tau} \tag{5.17}$$

and

$$\frac{P \Downarrow u \quad Q[u/x] \Downarrow v}{\texttt{let}\ x \leftarrow P\ \texttt{in}\ Q \Downarrow v} \tag{5.18}$$

WF_1 is composed with QA using the following rules:

$$\frac{D : L(\mathbf{s}) \vdash \mathsf{WF}_1 : L(\mathbf{s} \times L(\mathbf{s}^3)) \quad y : L(\mathbf{s} \times L(\mathbf{s}^3)) \vdash \mathsf{QA} : L(\mathbf{s} \times L(\mathbf{s}^3))}{D : L(\mathbf{s}) \vdash \texttt{let}\ y \leftarrow \mathsf{WF}_1\ \texttt{in}\ \mathsf{QA} : L(\mathbf{s} \times L(\mathbf{s}^3))} \tag{5.19}$$

and

$$\frac{\mathsf{WF}_1[\vec{e}/D] \Downarrow \vec{w} \quad \mathsf{QA}[\vec{w}/y] \Downarrow \vec{v}}{(\texttt{let}\ y \leftarrow \mathsf{WF}_1\ \texttt{in}\ \mathsf{QA})[\vec{e}/D] \Downarrow \vec{v}} \tag{5.20}$$

where $\vec{v} = [\langle d_1, \vec{a}''_1 \rangle \ldots \langle d_m, \vec{a}''_m \rangle]$ includes the new annotations computed by QA.

Merging of Quality Assertion values

Assuming that we have m processors QA_1 through QA_m, let $\mathsf{P}_{\mathsf{QA_h}}$, $h : 1 \ldots m$ denote each of the m workflow fragments resulting from the application of this composition to each QA_k. The next step involves merging the outputs of all these workflow fragments, again using the Merger processor seen earlier. These are the corresponding rules, where the semantics of the involved operators should now be clear.

$$\frac{\{D : L(\mathbf{s}) \vdash \mathsf{P}_{\mathsf{QA_h}} : L(\mathbf{s} \times L(\mathbf{s}^3))\}_{h:i\ldots m} \quad x_1 : L(\mathbf{s}^3) \ldots x_m : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{Merger} : \mathbf{s} \times L(\mathbf{s}^3)}{D : L(s) \vdash \mathtt{let}\ x_1 \odot \ldots \odot x_n \leftarrow \mathsf{P}_{\mathsf{QA_1}} \odot \ldots \odot \mathsf{P}_{\mathsf{QA_l}} \quad \mathtt{in}\ \mathsf{Merger} : L(\mathbf{s} \times L(\mathbf{s}^3))}$$

with semantics:

$$\frac{\{\mathsf{P}_{\mathsf{QA_h}}[\vec{d}/D] \Downarrow \vec{v}_h\}_{h:i\ldots m} \quad \mathsf{Merger}[\langle d, \vec{a}_j \rangle / x_j\}_{j:1\ldots m}] \Downarrow \langle d, \vec{a} \rangle}{(\mathtt{let}\ x_1 \odot \ldots \odot x_m \leftarrow \mathsf{P}_{\mathsf{QA_1}} \odot \ldots \odot \mathsf{P}_{\mathsf{QA_m}}\ \mathtt{in}\ \mathsf{Merger})[\vec{d}/D] \Downarrow \vec{v}} \tag{5.21}$$

Similar to the previous use of Merger, $\vec{v}$ is the input dataset where each element is associated to its corresponding annotations.

Performing Quality Tests

As a final step the resulting workflow, denoted WF_2, is composed with each of the r Quality Test processors $\mathsf{QT}_1 \ldots \mathsf{QT}_r$ using iterative composition:

$$\frac{D : L(\mathbf{s}) \vdash \mathsf{WF}_2 : L(\mathbf{s} \times L(\mathbf{s}^3)) \quad w : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{QT}_k : \mathbf{s} \times \mathbf{s}}{D : L(\mathbf{s}) \vdash \mathtt{let}\ w \leftarrow \mathsf{WF}_2\ \mathtt{in}\ \mathsf{QT}_k : L(\mathbf{s} \times \mathbf{s})} \tag{5.22}$$

The corresponding semantics is:

$$\frac{\mathsf{WF}_2[\vec{d}/D] \Downarrow \vec{v} \quad \mathsf{QT}_k[\langle d_i, \vec{a}'_i \rangle / w] \Downarrow \langle d_i, c_i \rangle}{(\mathtt{let}\ w \leftarrow \mathsf{WF}_2\ \mathtt{in}\ \mathsf{QT}_k)[\vec{d}/D] \Downarrow [\langle d_1, c_1 \rangle, \ldots \langle d_m, c_m \rangle]} \tag{5.23}$$

This step yields a set of $k : 1 \ldots r$ independent workflows, denoted $\mathsf{WF}_{3,k}$:

$$\mathsf{WF}_{3,k} \equiv D : L(\mathbf{s}) \vdash \mathtt{let}\ w \leftarrow \mathsf{WF}_2\ \mathtt{in}\ \mathsf{QT}_k : L(\mathbf{s} \times \mathbf{s})$$

Since these workflows are independent, the final outputs from the entire quality workflow can be obtained by pairing. Assuming w.l.o.g. $r = 2$, this is written:

$$\frac{D : L(\mathbf{s}) \vdash \mathsf{WF}_{3,1} : L(\mathbf{s} \times \mathbf{s}) \quad D : L(\mathbf{s}) \vdash \mathsf{WF}_{3,2} : L(\mathbf{s} \times \mathbf{s})}{D : L(\mathbf{s}) \vdash \langle \mathsf{WF}_{3,1}, \mathsf{WF}_{3,2} \rangle : L(\mathbf{s} \times \mathbf{s}) \times L(\mathbf{s} \times \mathbf{s})} \tag{5.24}$$

with corresponding semantics:

$$\frac{\mathsf{WF}_{3,1}[\vec{d}/D] \Downarrow [\langle d_1, c_{11}\rangle, \langle d_2, c_{12}\rangle \dots] \quad \mathsf{WF}_{3,2}[\vec{d}/D] \Downarrow [\langle d_1, c_{21}\rangle, \langle d_2, c_{22}\rangle \dots]}{\langle \mathsf{WF}_{3,1}, \mathsf{WF}_{3,2}\rangle[\vec{d}/D] \Downarrow \langle [\langle d_1, c_{11}\rangle, \langle d_2, c_{12}\rangle \dots], [\langle d_1, c_{21}\rangle, \langle d_2, c_{22}\rangle \dots]\rangle} \tag{5.25}$$

where the two sets of class labels $\{c_{11}, c_{12} \dots\}$ and $\{c_{21}, c_{22} \dots\}$ are independent from each other.

5.4 Translating Quality Views into Quality workflows

We use the formal definition of Quality workflows to specify translation rules from Quality Views to Quality workflows. This is now straightforward, as it amounts to showing a simple mapping from the surface XML syntax for QVs, given in Section 4.2, and the formal syntax given here.

Annotator: An `<Annotator>` construct takes the form:

```
<Annotator type="sc">
   <variables>
     <var variableName="var1" metricName="cl1" />
     <var variableName="var2" metricName="cl2" />
     <var ...  />
   </variables>
</Annotator/>
```

where *sc* is a reference to an ontology subclass of `AF`. This maps to a processor A_{sc} defined in (5.2):

$$D : L(\mathbf{s}) \vdash \mathsf{A}_{\mathsf{sc,V}} : L(\mathbf{s}^3) \tag{5.26}$$

The set $V = \{var_1, var_2, \dots\}$ is mentioned explicitly as part of the processor name. This makes it clear that the requirement of disjointness of variable sets among Annotation processors can be enforced by static analysis, i.e., that $V_i \cap V_j = \emptyset$ for any two $\mathsf{A}_{\mathsf{i,V_i}}$, $\mathsf{A}_{\mathsf{j,V_j}}$.

We use the variables' declarations to further specify the semantics given in Eq. 5.6, as follows:

$$\mathsf{A}_{\mathsf{sc}}[\vec{e}/D] \Downarrow [\langle var_1, cl_1, val_1\rangle, \langle var_2, cl_2, val_2\rangle, \dots]$$

Note that now the variable names and classes in each annotation triple are explicitly specified in the output.

QualityAssertion : A QA construct of the form

```
<QualityAssertion type = "sc" tagName="q">
   <variables>
     <var variableName="var1" metricName="cl1" />
     <var variableName="var2" metricName="cl2" />
     <var ...  />
   </variables>
   </QualityAssertion>
```

corresponds to a QA processor:

$$y : L(\mathbf{s} \times L(\mathbf{s}^3)) \vdash \mathsf{QA}_{\mathsf{sc,V,q}} : L(\mathbf{s} \times L(\mathbf{s}^3)) \tag{5.27}$$

where sc is a reference to an ontology subclass of `QA`, $V = \{var_1, var_2 \dots\}$ is the set of input variables, and q is the name of the output variable. Again, the semantics is a refinement of (5.8):

$$\mathsf{QA}_{\mathsf{h}}[\vec{w}/y_h] \Downarrow \vec{v}_h$$

where $\vec{v}_h$ includes the new annotation for variable q. Formally, let $\vec{w} = [\langle d_1, \vec{a}_1\rangle \dots \langle d_m, \vec{a}_m\rangle]$. Then

$$\vec{v}_h = [\langle d_1, \vec{a}'_1\rangle \dots \langle d_m, \vec{a}'_m\rangle]$$

where $a'_j = a_j \cdot \mathsf{up}(\langle q, sc, v\rangle)$ (here up wraps an element a to a one-element list $[a]$, and "$\cdot$" is list concatenation).

Splitter Action. The specification of this type of action takes the form:

```
<action>
   <split>
     <variables>
          <var variableName="var_1" metricName="cl_1" />
          <var variableName="var_2" metricName="cl_2" />
          <var ...  />
     </variables>
     <channel name="o_1">"expr_1"</channel>
     <channel name="o_2">"expr_2"</channel>
   </split>
</action>
```

This construct maps to a set of Taverna processors of type `QT`, according to the syntax (5.9), one for each channel:

$$w : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{QT}_{\mathsf{expr_k,V,o_k}} : \mathbf{s} \times \mathbf{s} \tag{5.28}$$

where $V = \{var_1, var_2 \dots\}$ is the set of variables, $expr_k$ is a valid string in the expression language specified in Appendix A, and it only contains variables in V.

The corresponding semantics is:

$$\mathsf{QT_k}[\langle d, \vec{a}\rangle / w] \Downarrow c$$

where: (i) the list $\vec{a}$ of annotations contains triples $\langle var_i, cl_i, v_i\rangle$ for each $var_i \in V$, and (ii) $c = \langle d, o_k\rangle$ if $eval(expr_k[\{v_i/var_i\}_{i:1\cdots|V|}]) =$ `true`, and null otherwise.

Filter Action. A Filter is just a simplified version of a Splitter, with only one condition and implicit quality classes, i.e., "accept" and "reject". The filter construct:

```
<action>
   <filter>
     <variables>
          <var variableName="var_1" metricName="cl_1" />
          <var variableName="var_2" metricName="cl_2" />
```

```
            <var ...  />
        </variables>
        <expression>"expr"</expression>
    </filter>
</action>
```

maps to a single Taverna processor of type `QT`, with syntax and semantics as described above.

The QV-to-Taverna compiler implements the formal translation rules just described. More specifically, the translation process consists of the following main steps:

- performing syntactic and semantic analysis of the QV, to ensure that the QV is *valid* according to the definition given in Section 4.2, namely that (i) the XML syntax conforms to the XML schema defined for QVs, and (ii) the semantic naming constraints of Section 4.2.3 are satisfied;
- checking the *formal consistency* of the QV, according to the rules defined in Section 4.4;
- creating instances of processors of type (5.26), (5.27), and (5.28). This step makes use of a registry that maps ontology classes for Annotation and QA functions onto deployed Web Services (i.e., to actual Web Service endpoints) that provide an implementation of those functions. The registry is a feature of the Qurator workbench, which is fully described in the next chapter;
- connecting the processors according to the rules defined in Section 5.3. This includes adding `Merger` processors as required;
- configuring some of the processors. This involves supplying parameters to processors, to specify their behaviour during execution. The specific repository where computed annotations are to be stored, for example, is a parameter of the Annotator processor (it is specified using the `repositoryRef` attribute described in Section 4.2.4). Also, some of the required functionality is obtained by configuring a generic, predefined processor; for example, a filter with condition c is instantiated by configuring a generic, predefined filtering processor with parameter c. Configuration is achieved in Taverna by

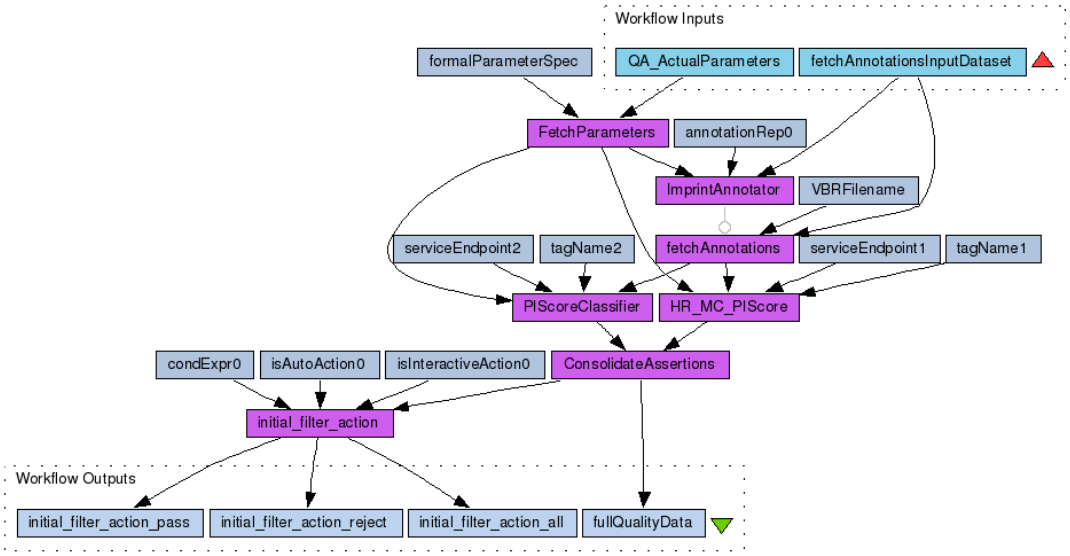


Figure 5.3: Quality workflow with ancillary configuration processors

adding ancillary processors to the structure described so far in this chapter. These additional processors simply supply constant values to the processors at execution time, through dedicated input ports. For completeness, we show an actual generated Quality workflow with the additional processors, in Figure 5.3.

5.5 Correctness of Quality workflows

Previously in this chapter, we have alluded to the connection between the semantics of Quality workflows, as an actual implementation of Quality Views, and the semantics of abstract QVs given as a functional program in the previous chapter. We are now finally in a position to make this connection formal, by showing that the QV-to-Taverna translator generates a workflow whose semantics is consistent with that of abstract QVs. Ideally, this amounts to showing that the following diagram is commutative:

$$\begin{array}{ccc} QV & \xrightarrow{\text{comp}} & WF_{QV} \\ \downarrow & & \downarrow exec(WF_{QV}, D) \\ f_{QV}() & \xrightarrow{eval(f,D)} & \mathbf{y} \end{array}$$

That is, for any input dataset D, the result $\mathbf{y}$ computed by compiling QV into a Quality workflow WF_{QV}, and executing it on input D, is the same as the result obtained by applying f_{QV} to D.

Note however that the semantics of f_{QV} and that of WF_{QV} are defined in different ways: we have a complete functional interpretation of QV, but only a structural definition for WF_{QV}. As a consequence, we will have to make the assumption that workflow processors correctly implement their corresponding quality functions. Under this assumption, we prove functional correctness by showing that WF_{QV} is structurally equivalent to the QV that it is compiled from. In order to do this, we must first describe QV using the same structural semantics that we have used for the workflow. This requires that we first define the structural operational semantics of the Haskell QV intepreter of Section 4.3, and then proceed to show inductively the structural consistency between workflow functions and QV functions.

5.5.1 Syntax and semantic rules for the QV interpreter

To recap, in Section 4.3 we began by introducing an *environment* data structure, designed to hold partial quality values during the computation, and then described the top-level interpreter function `QV`. This is a composition of elementary functions that apply Annotation, Assertion and Action functions to the data, i.e., `annotate`, `qassert`, and `act`. The main structural difference between the QV interpreter and the workflow is that describing the interpreter requires modelling both the function themselves, *and* the Haskell interpreter functions that apply those functions to the input data. In particular, the input variables in the interpreter, denoted by f, are now bound to quality functions. With this provision, we now give the syntax and semantics of these elements.

Environment

The environment is defined as the type

$$env = L(\mathbf{s} \times L(\mathbf{s}^3))$$

As we have seen in the previous section, in the workflow the environment is passed from processor to processor through the data links in the form of messages: the environment access functions that we have seen in the QV interpreter have no

equivalent in the workflow, because they are not exposed as explicit processors.

Annotation functions

The functional definition of a generic Annotation function A in the interpreter is identical to Eq. 5.5, and corresponds to the following axiom:

$$D : L(\mathbf{s}) \vdash A : L(\mathbf{s} \times L(\mathbf{s}^3)) \tag{5.29}$$

The interpreter function $\mathsf{annotate}_1$ applies A to an input dataset D. We can use simple composition to model function application, as follows:

$$\frac{D : \mathbf{s} \vdash A : L(\mathbf{s} \times L(\mathbf{s}^3)) \quad D : L(\mathbf{s}), f : L(\mathbf{s} \times L(\mathbf{s}^3)), e : env \vdash \mathsf{annotate}_1 : env}{D : L(\mathbf{s}), e : env \vdash \texttt{let } f \leftarrow A \texttt{ in } \mathsf{annotate}_1 : env}$$

where f is the input variable to $\mathsf{annotate}_1$ that binds to annotation function A.

Let $\vec{d} = [d_1 \dots d_m]$. The semantics is:

$$\frac{A[\vec{d}/D] \Downarrow ann \quad \mathsf{annotate}_1[e_0/e][ann/f] \Downarrow e'}{(\texttt{let } f \leftarrow A \texttt{ in } \mathsf{annotate}_1)[\vec{d}/D][e_0/e] \Downarrow e'}$$

where $ann = [\langle d_1, \vec{a}_1 \rangle, \langle d_2, \vec{a}_2 \rangle \dots]$ is the annotation structure computed by A, e_0 is the initial environment, and $e' = e_0 \cdot ann$ is the final environment after interpretation.

By extension, $\mathsf{annotate}$ applies a list of n annotation functions A_i to D:

$$\frac{\{D : \mathbf{s} \vdash A_i : L(\mathbf{s} \times L(\mathbf{s}^3))\}_{i:1..n} \quad D : L(\mathbf{s}), f_1 : L(\mathbf{s} \times L(\mathbf{s}^3)) \dots, e : env \vdash \mathsf{annotate} : env}{D : L(\mathbf{s}), e : env \vdash \texttt{let } f_1 \odot \dots \odot f_n \leftarrow A_1 \odot \dots \odot A_n \texttt{ in } \mathsf{annotate} : env} \tag{5.30}$$

with corresponding semantics:

$$\frac{\{A_i[\vec{d}/D] \Downarrow ann_i\}_{i:1\dots n} \quad \mathsf{annotate}[e_0/e]\{[ann_i/f_i]\}_{i:1\dots n} \Downarrow e'}{(\texttt{let } f_1 \odot \dots \odot f_n \leftarrow A_1 \odot \dots \odot A_n \texttt{ in } \mathsf{annotate})[\vec{d}/D][e_0/e] \Downarrow e'} \tag{5.31}$$

where now $ann_i = [\langle d_1, \vec{a}_{i1} \rangle, \langle d_2, \vec{a}_{i2} \rangle \dots]$ is the annotation structure computed by A_i, e_0 is the initial environment, and $e' = e_0 \cdot ann_1 \cdot \dots \cdot \dots ann_n$ is the final environment after interpretation.

Quality Assertions

The functional definition of a Quality Assertion function is similar to the one in Eq. 5.7, except for the new environment e:

$$D : L(\mathbf{s}), e : env \vdash QA : env \tag{5.32}$$

Similar to what we have seen for Annotation functions, the Qassert interpreter function applies QA to an input dataset D using environment env, which carries the annotated data. The result is a new version of the environment that includes the new annotation with the value computed by QA_f:

$$D : L(\mathbf{s}), f : L(\mathbf{s}^3), e : env \vdash \mathsf{Qassert} : env$$

This application is similar to the case for Annotations. For m QA functions, we have the rules:

$$\frac{\{D : L(\mathbf{s}), e : env \vdash QA_i : env\}_{i:1\ldots m} \quad D : L(\mathbf{s}), f_1 : env, \ldots f_m : env, e : env \vdash \mathsf{Qassert} : env}{D : L(\mathbf{s}), e : env \vdash \texttt{let } f_1 \odot \ldots \odot f_r \leftarrow QA_{f_1} \odot \ldots \odot QA_{f_m} \texttt{ in } \mathsf{Qassert} : env} \tag{5.33}$$

and

$$\frac{\{QA_i[\vec{d}/D][e_0/e] \Downarrow e_i\}_{i:1\ldots m} \quad \mathsf{Qassert}\{[e_i/f_i]\}_{i:1\ldots m} \Downarrow e'}{(\texttt{let } f_1 \odot \ldots \odot f_m \leftarrow QA_1 \odot \ldots \odot QA_m \texttt{ in } \mathsf{Qassert})[\vec{d}/D][e_0/e] \Downarrow e'} \tag{5.34}$$

Quality tests

Finally, a Quality test is represented by a function QT:

$$D : L(\mathbf{s}), e : env \vdash QT : L(\mathbf{b}) \tag{5.35}$$

where $\mathbf{b} = \texttt{True}|\texttt{False}$ is a boolean type. This function returns a list of boolean values, one for each input data element.

The QV interpreter function act applies a test QT to the annotated input data, and assigns a class label to each input data element depending on the outcome of the test:

$$D : L(\mathbf{s}), T : L(\mathbf{b}), e : env \vdash \mathsf{act} : L(\mathbf{s} \times \mathbf{s} \times L(s^3))$$

where the list of boolean variables $L(t)$ represents the outcome of the application

of QT to the input. Note that the test condition is part of the QT function and is not exposed to the interpreter. Each triple in the result type of act consists of a dataref with associated label and annotations. The label represents a quality class to which the data element is assigned. For a single condition, as illustrated here, we are going to use two quality classes, $\mathtt{c}_1$ and $\mathtt{c}_2$, corresponding to a true and false condition, respectively.

The interpreter's invocation of a set of r test functions on the data is defined by the following rules:

$$\frac{D : L(\mathbf{s}), e : env \vdash QT : L(\mathbf{b}) \quad D : L(\mathbf{s}), T : L(\mathbf{b}), e : env \vdash \mathsf{act} : L(\mathbf{s} \times \mathbf{s} \times L(s^3))}{D : L(\mathbf{s}), e : env \vdash \mathtt{let}\ t \leftarrow QT\ \mathtt{in}\ \mathsf{act} : L(\mathbf{s} \times \mathbf{s} \times L(s^3))} \tag{5.36}$$

and

$$\frac{QT[\vec{d}/D][e_0/e] \Downarrow \vec{b} \quad \mathsf{act}[\vec{b}/f] \Downarrow \vec{w}}{(\mathtt{let}\ t \leftarrow QT\ \mathtt{in}\ \mathsf{act})[\vec{d}/D][e_0/e] \Downarrow \vec{w}} \tag{5.37}$$

where $\vec{w} = \{\langle d_j, \mathtt{c}_1, e_j\rangle | b_j = \mathtt{True}\} \cup \{\langle d_j, \mathtt{c}_2, e_j\rangle | b_j = \mathtt{False}\}$, l and e_j is the set of annotations associated to d_j in the environment.

5.5.2 Correctness

As mentioned earlier in the section, there are two main structural differences between QVs and workflow functions. Firstly, QV functions define an interpreter, hence they take functions as parameters and apply them to data, while in the workflow, functions are first-class processors. And secondly, QV functions use an explicit environment, while workflow processors rely on data links. Despite these differences, however, we can still establish a structural correspondence among the functions. To establish functional correctness, we have to further assume that the result of executing a processor corresponding to a QV function on some input is the same as that of interpreting the function on the same input. For example, for Annotation functions we must postulate that if

$$\mathsf{A}_i[\vec{d}/D] \Downarrow [\langle d_1, ann_1\rangle, \langle d_2, ann_2\rangle \dots]$$

and

$$(\mathtt{let}\ f \leftarrow A_i\ \mathtt{in}\ \mathsf{annotate_1})[\vec{d}/D][e_0/e] \Downarrow e'$$

then in e' the annotations for each d_i are precisely ann_i.

Having made this assumption for the base case, we can show the structural

correspondance by induction. Observe that the annotate function is structurally equivalent to the workflow denoted with $\mathsf{WF_1}$ in Section 5.3 (Eq. 5.15), namely:

$$D : L(s) \vdash \texttt{let}\ x_1 \odot \ldots \odot x_n \leftarrow A_1 \odot \ldots \odot A_n\ \texttt{in}\ \mathsf{Merger} : L(\mathbf{s} \times L(\mathbf{s}^3))$$

Since at the beginning of the QV interpretation the environment is empty, we conclude that the semantics of $\mathsf{WF_1}$ is correct.

Moving on to QA processors: $y : L(\mathbf{s} \times L(\mathbf{s}^3)) \vdash \mathsf{QA_h} : L(\mathbf{s} \times L(\mathbf{s}^3))$ (5.7), we again observe that they are structurally identical to the functions QA_i (5.32), since $env = L(\mathbf{s} \times L(\mathbf{s}^3))$. As we did for Annotators, here we must again make an assumption of functional equivalence between each of the $\mathsf{QA_h}$ processors and corresponding functions QA_i —the details are similar to those shown earlier.

With this assumption, we may conclude that the result of merging m QA values, denoted earlier by $\mathsf{WF_2}$:

$$D : L(s) \vdash \texttt{let}\ x_1 \odot \ldots \odot x_n \leftarrow P_{QA_1} \odot \ldots \odot P_{QA_l} \qquad \texttt{in}\ \mathsf{Merger} : L(\mathbf{s} \times L(\mathbf{s}^3))$$

can be obtained by applying the QV function Qassert to m QA functions, starting from an environment computed by $\mathsf{WF_1}$. By structural induction, we conclude again that the resulting workflow up to this point, denoted with $\mathsf{WF_2}$ in Section 5.3, is correct.

A similar set of reasoning steps allows us to deal with the Actions layer of the Quality workflow. To recall, a single QT processor: $w : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{QT} : \mathbf{s} \times \mathbf{s}$ associates a quality class to a dataref: $\langle d, c \rangle$, and is applied to our current workflow $\mathsf{WF_2}$ as follows (5.22):

$$D : L(\mathbf{s}) \vdash \texttt{let}\ w \leftarrow \mathsf{WF_2}\ \texttt{in}\ \mathsf{QT_k} : L(\mathbf{s} \times \mathbf{s}) \tag{5.38}$$

The corresponding QV function is act, which, as expected, takes tests T as explicit input parameters (in addition to the environment):

$$D : L(\mathbf{s}), T : L(\mathbf{b}), e : env \vdash \mathsf{act} : L(\mathbf{s} \times \mathbf{s} \times L(s^3))$$

Note that the output is more general than the equivalent processor, in that it also includes all annotations. We are again in a situation where the semantics of the processor is less precise than that of the corresponding function. Recalling that a different class c is associated to d in the processor depending on the outcome of

the test, ultimately we conclude by induction that the entire workflow computes the set of quality classes that would be computed by the QV interpreter on the same input.

5.6 Embedded Quality workflows

We began the chapter with an illustration of a Taverna Quality workflow (Figure 5.2), and have now provided the formal machinery necessary to understand it in detail. The purpose of this workflow structure, however, only becomes clear when it becomes part of some larger workflow, providing "embedded" quality controls for it. Note that this is a natural operations to perform on workflows, because workflow models naturally support the notion of recursive sub-processes.

The problem that we address in this section is how to provide users with a simple mechanism to specify the integration. Let us begin, as an example, by analysing the steps required to integrate our example proteomics workflow, referred to as the *host workflow* with the quality workflow obtained by compiling our example QV.

- Locate the processors in the host workflow whose output is in the *scope* of the QV. To recall the definition given in Section 4.1, the scope of a QV is a Data Entity class in the ontology (specified in the <QV> element of the QV), and it indicates that the QV can be computed on all and only the collections of datasets that are individuals of the class. For example, if the scope is `HitEntry`, then any dataset that is semantically annotated with class `HitEntry` or any of its sub-classes, can be used as input to the QV. Thus, if the output from any of the processors is semantically annotated in this way, then these processors can be used to provide input to the QV. In our proteomics example, we assume that the output from `identifyProteins` is annotated with class `ImprintHitEntry`, and is therefore in the scope. Note that, ultimately, the indication of which data types are eligible to be in the scope comes from the users, who are responsible for annotating the data in the workflow[4]. We refer to the processors that produce these datasets as *quality insertion points*.

[4]If more than one dataset is eligible, for example, it is a users' choice where the quality workflow will be embedded.

- Identify whether there is a need for an adaptor processor (also known as a "shim service" [Hul06]) to convert the output data format of `identifyProteins` to the input format of the top-level Quality processor, namely `fetchAnnotations`. In this case, the `ImprintToQuratorAdapter` has been developed for the purpose. Adapters are linked so that they receive the output of insertion points in the host workflow.

- Add input links from the insertion points in the host workflow (or from the adapters, if they are present) to the top-level processors in the Quality workflow. These top-level processors are `fetchAnnotations` and each of the Annotators, i.e., `ImprintAnnotator`. Note that `fetchAnnotations` *does* require the input dataset: as we recall, this processor does not receive direct input from the Annotators, to account for the possibility that none exists. Therefore, it must take its input directly from the host processor.

- The outputs from the Action processors must be connected to some input in the host workflow. In the case of a simple Filter, one would typically connect the "accept" output to a processor that can carry out the appropriate action in the host workflow, while the "reject" output is simply not used. This is illustrated in the figure, where the "pass" port of `initial_filter_action` is connected to the input data port of `Uniprot2GO` in the host workflow[5]. We reiterate again that while filtering is a common type of action others may be specified. Using an n-way Splitter, for example, leaves the designer free to use any of the available n+1 output ports (including the pass-through "all" port).

The result of embedding in our example workflows, with the data ports, is shown in Figure 5.4.

This sequence of steps, which is common to all quality workflows, suggests that the integration can be carried out automatically, provided that users give explicit instructions for each of the steps above. To achieve this goal, we have proposed a simple *quality workflow deployment* language, described in the next section, that lets users specify how the integration should be performed. As part of Qurator, we have implemented a deployment component that interprets the language and carries out the instructions.[6]

[5]The figure only shows the ports that are connected through a data link. Therefore the

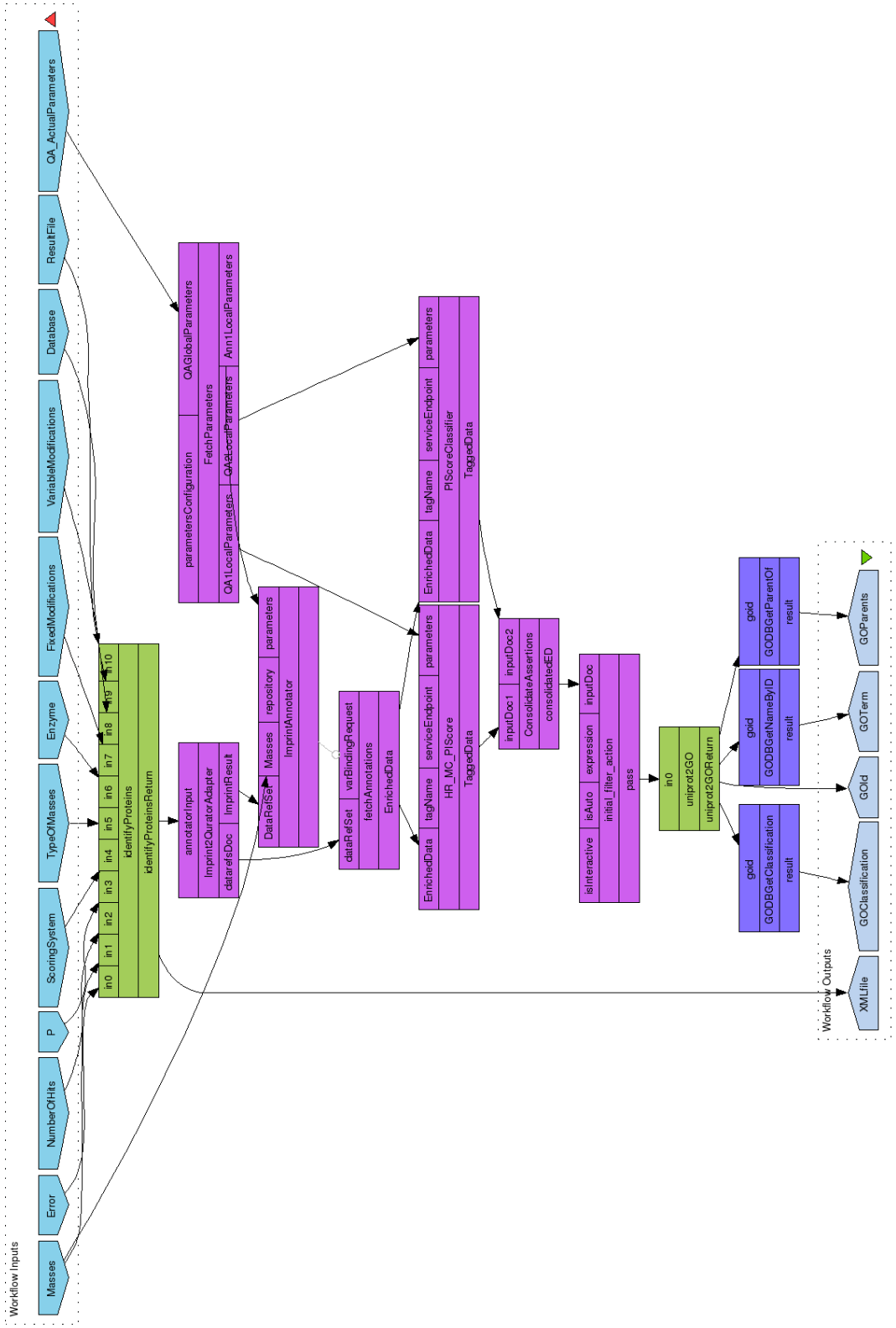


Figure 5.4: Proteomics workflow with embedded Quality workflow

5.6.1 Worklow deployment language

The language is based on only two graph-manipulation primitives, namely (i) adding a data or control link from processor A to processor B, and (ii) add a processor to the workflow. In addition, the user may specify whether adding a link from A results in previous outgoing links from A to be retained, or removed.

The latter feature lets users determine whether the quality workflow should be "intrusive", by intercepting the flow of the host workflow, or whether it should be deployed as an additional branch of the host workflow. In the former case the quality workflow is suitable for filtering data, as in the case shown in our example, while in the latter it provides new quality-aware output in addition to the output produced by the host workflow.

Using these simple primitives we can specify embeddings like the one shown in Figure 5.5. The deployment descriptor that produces the result in the figure, written in an XML-based syntax, is shown in Figure 5.6[7]. In its definition, note that a link is defined as a pair

$$[\langle \texttt{source processor}, \texttt{source port} \rangle, \langle \texttt{sink processor}, \texttt{sink port} \rangle]$$

for example $[\langle \texttt{initial\_filter\_action}, \texttt{Pass} \rangle, \langle \texttt{Uniprot2Go}, \texttt{in0} \rangle]$.

We believe the syntax to be largely self-explanatory. Let us now consider the effect of the `overriding` attribute in the first `<connector>` element.

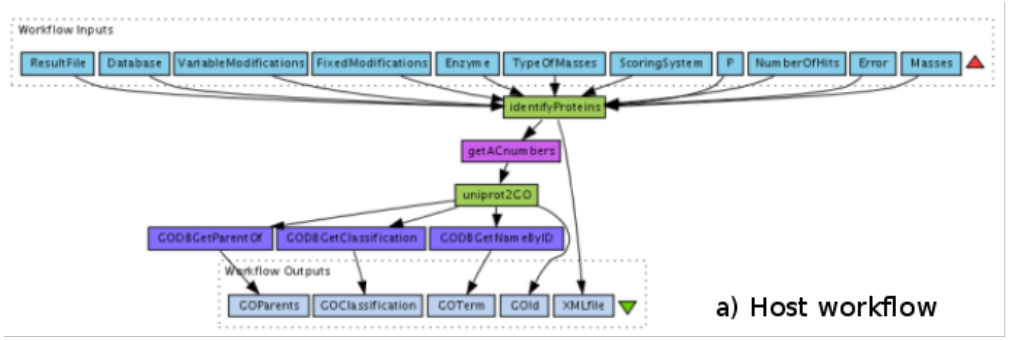
- The output from `identifyProteins` is redirected to the annotator, through the new adapter;
- Since `Masses`, one of the host workflow inputs, is required by `ImprintAnnotator` too, in the second `<connector>` element a link is added. This link is non-overriding because other processors need the value of `Masses`;
- The output from the adapter is routed both to the Annotator and to `fetchAnnotations`. The overriding setting is irrelevant here

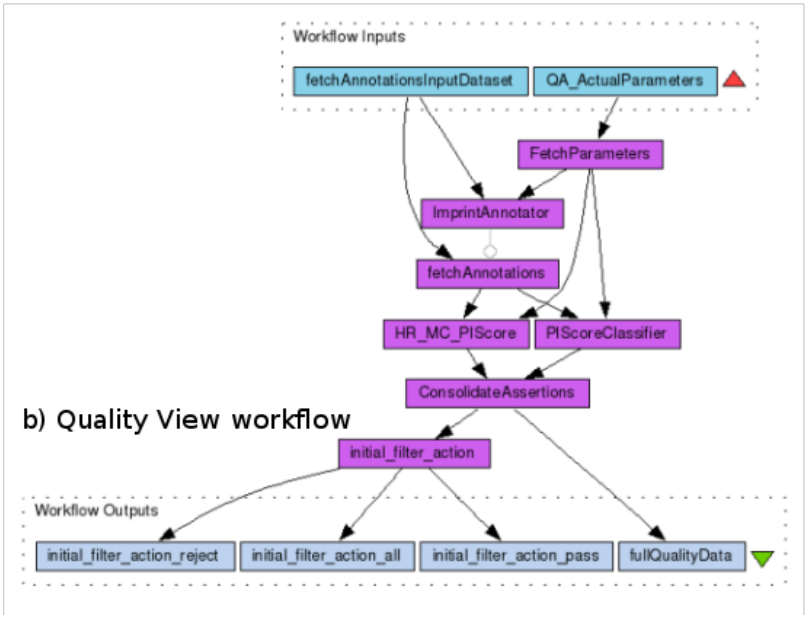
"reject" port, for example, is not shown.

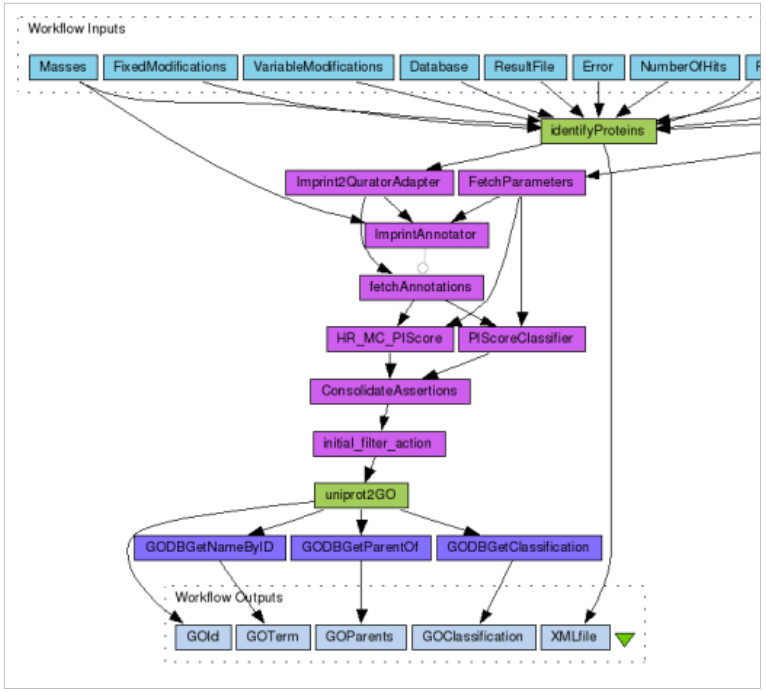
[6]Part of this implementation is due to our student, Paul Waring

[7]The initial `<scufl>` tag is a reference to the name, Scufl, of the Taverna language. It is used to indicate that this descriptor should be interpreted in the context of a Taverna workflow. This leaves open the option to add deployment descriptors for other workflow systems, which may require a different set of primitives.

Figure 5.5: Host and Quality workflows and the result of embedding







```
<deployment>
   <scufl hostflow="data/workflow/ImprintBaseWorkflow">
     <adapter name="Imprint2QuratorAdapter"
          scuflProcessor="ImprintOutputAdapter"/>

     <connector sourceProc="identifyProteins"
          sourcePort="identifyProteinsReturn"
          destProc="Imprint2QuratorAdapter" destPort="annotatorInput"
          linktype="data" overriding="true"/>

     <connector sourceInput="Masses"
          destProc="ImprintAnnotator" destPort="Masses"
          linktype="data" overriding="false"/>

     <connector sourceProc="Imprint2QuratorAdapter"
          sourcePort="datarefsDoc"
          destProc="fetchAnnotations" destPort="dataRefSet"
          linktype="data" overriding="true"/>

     <connector sourceProc="Imprint2QuratorAdapter"
          sourcePort="ImprintResult"
          destProc="ImprintAnnotator" destPort="DataRefSet"
          linktype="data" overriding="false"/>

     <connector sourceProc="initial_filter_action"
          sourcePort="pass"
          destProc="uniprot2GO" destPort="in0"
          linktype="data" overriding="false"/>
   </scufl>
</deployment>
```

Figure 5.6: Deployment descriptor for integrating the example Quality View within the Ispider workflow

since the adapter is a new processor. This is consistent with the workflow in Figure 5.2, where the dataset input is represented by `fetchAnnotationsInputDataset`. As mentioned earlier, `fetchAnnotations` uses this input to retrieve annotations from a repository, regardless of whether any Annotators are also present;

- Finally, the "pass" output from the action processor is wired to the input port that originally received the output from `identifyProteins`.

We note a final but important point regarding the semantics of the deployer. When links are removed, some processors may end up without inputs or outputs. The deployer automatically removes these processors, as they no longer contribute to the workflow. This accounts for the disappearance of processor `getACNumbers`, which is no longer needed, and of the original input `fetchAnnotationsInputDataset`.[8]

[8]In fact, the latter is auto-generated by the translator, to make it possible to use the Quality

5.7 Summary and Conclusions

In this chapter we have described a realization of abstract Quality Views as executable Taverna *quality workflows*, for which we have given a formal syntax and structural semantics, along with a formal definition of the translation process. The detailed illustration of the steps involved gives a calibration of the type of effort expected from scientists who want to deploy quality control features as part of data processing, when workflows are used to describe *in silico* experiments. In particular, by showing that the translation can be automated we support our claim that the IQ framework is close to the scientist's intuition when addressing quality issues.

workflow in "standalone mode", i.e., without embedding – mostly for testing purposes.

Chapter 6

The Qurator workbench

The lifecycle for Information Quality, proposed in Chapter 2, describes a set of user tasks designed to make data processing applications in e-science quality-aware. The research described so far has clarified the extent to which we can support the users in some of these tasks, namely the specification of consistent Quality Views and their compilation into executable processes, i.e., worklows, as well as in their integration as part of other scientific workflows. Having implemented these functionalities as software components, we now aim to design a more complete suite of components in order to support the users throughout the entire outer loop of the lifecycle. We call this suite of components, organized into a software architecture, the *Qurator workbench*, to emphasize that they provide an environment for scientists to experiment with personal definitions of information quality, and their application as part of data processing. The Quality View specification tool and compiler are two of the Qurator components.

In this chapter we investigate the challenges associated with the remaining tasks in the lifecycle, show how they translate into workbench components, and argue that the Qurator workbench provides support to the entire outer loop of the lifecycle. For clarity, we reproduce the IQ lifecycle diagram from page 59 here, as Figure 6.1. In this version of the figure we have numbered the tasks for which we provide support in the Qurator workbench. In Chapter 3 we have described the IQ ontology, and how users can contribute to it with domain-specific concepts (task 2 in the figure); Quality View specification (task 3) has been the topic of Chapter 4, and its compilation and integration (4 and 5) were addressed in the previous chapter.

As we have stated early in the introductory chapter, the entire inner loop is

out of the scope of the workbench. We also have an un-numbered task, namely the analysis of data acceptability criteria. In this task, scientists make the initial determination of whether quality controls are needed at all, and at the end of each iteration around the loop, decide whether the current quality process configuration is satisfactory. We regard this as a knowledge-intensive, distinctly human activity that is out of the scope of the workbench, as well.

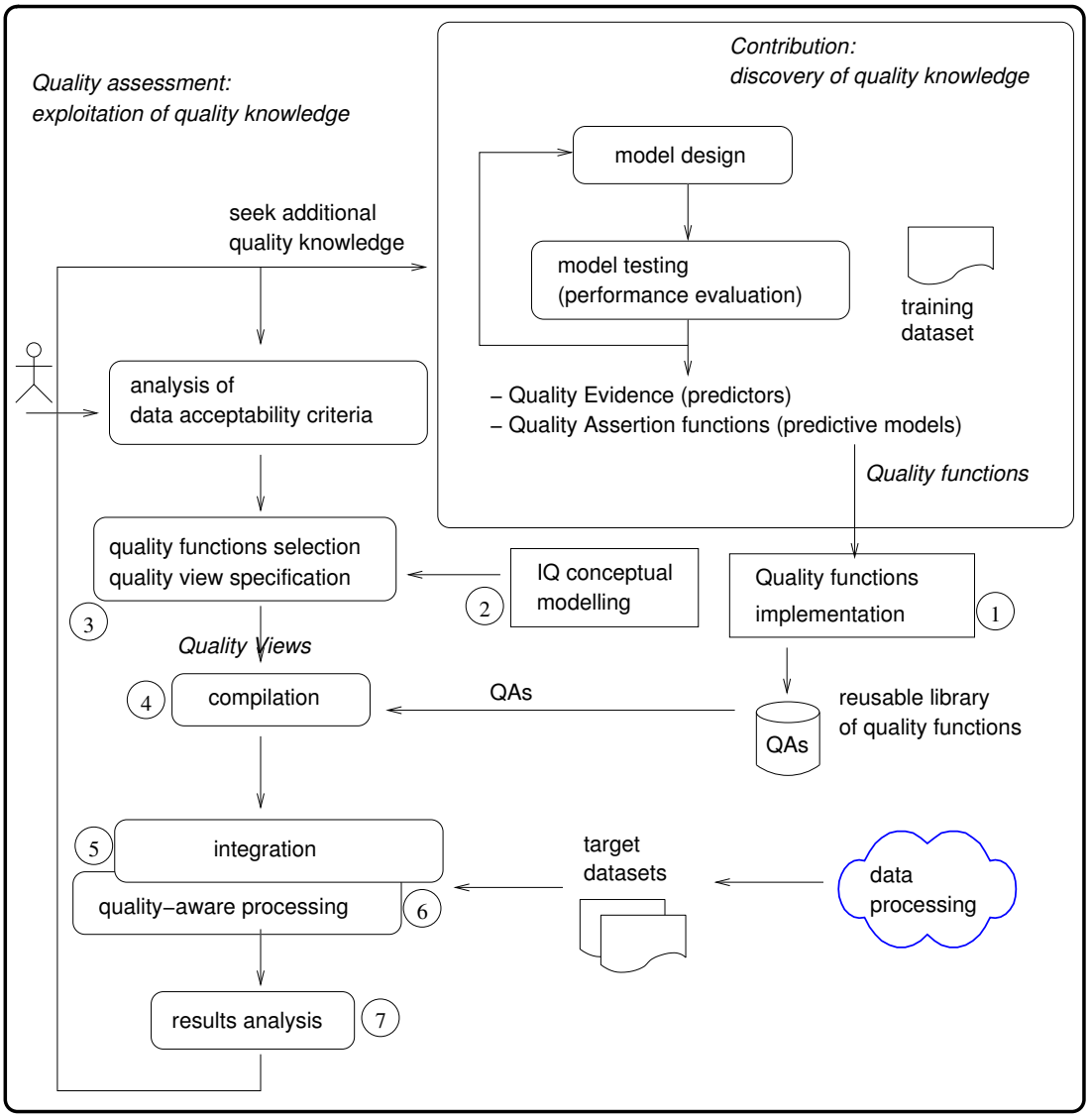


Figure 6.1: The IQ lifecycle as a workplan for the Qurator workbench

In the next three sections of this chapter we are going to address the remaining tasks that we have not yet discussed, as follows:

implementation of quality functions (task 1): here we address the problem

of facilitating the implementation of Quality Assertion functions as Web services, by partially automating the generation of the service code;

forms of quality-aware data processing (task 6): having described how Quality Views can be used to make scientific workflows quality-aware, we extend the result to database query processing, specifically for XML data; and

analysis of the execution results (task 7): here we argue that, as part of data and data quality analysis, users should be able to inspect the execution of a Quality View, in order to determine why a certain classification was made. We propose the new notion of *quality provenance*, or provenance about quality, and we address the problem of collecting and then using quality provenance.

Finally, in the last section we present our architectural solution for the entire Qurator workbench, highlighting the purpose of each of its components with respect to the lifecycle tasks.

6.1 Implementation of quality functions

Quality Assertion functions (quality functions, for short) are implemented as software components that can be composed into Quality Views. Since many different Quality Views may use the same quality functions, one important requirement for these components is their reusability. We facilitate reusability by stipulating that, in order to be used within the Qurator workbench, quality functions must be implemented as Web services, and furthermore, that all such services must implement the same interface. This facilitates the composition process by ensuring that all quality functions are invoked in the same way.

An important additional benefit of this uniformity is that it offers opportunities for the automated generation of some of the service implementation code. Of course, a number of Web service toolkits already provide utilities to generate implementation skeletons from a WSDL interface (the well-known Axis Apache project is one of them). These utilities, however, are generic in that they do not make any assumption on the type of messages that are exchanged over the interface, let alone the service implementation logic. As a result, the generated

implementation includes only to the code required to send and receive SOAP messages, regardless of their content and their intended use.

Quality functions, however, are likely to have a uniform structure, since, after all, by definition they all compute either a classification or a score model, using vectors of quality evidence values. Furthermore, the structure of their messages is known from the common definition of their interface. Based on these observations, in this section we investigate the opportunities and limits of generating code specifically for quality functions, when additional information is given regarding the structure of their messages, and to a limited extent, their internal logic.

Initially, let us consider the case of two quality functions with completely arbitrary logic. Despite their differences, we can assume that their implementation will have a recognizable common structure, consisting of (i) processing input messages to retrieve the input quality evidence values, (ii) computing the function value, and (iii) composing an output message.

Since the message structure is not only common to all these functions, but it is specified as part of their shared service interface, we can supply helper classes that transform the external message formats into a predictable internal object model. We observe that the *environment* data structure described in Chapter 4 (Section 4.3) is in fact a suitable representation of the function's input; to recall, it consists of a collection of attribute vectors, one for each input data element, with one element for each quality evidence value, plus one element to hold the function value, i.e., the classification. Thus, we can generate an implementation stub that includes calls to the helper classes to read and write elements of the environment, obviating the need for users to write the repetitive and less interesting parts of the code by hand.

This is already a step further from a generic stub that is completely oblivious to the message structure. If we do not know anything about the function logic, however, there is not much more that we can do to help the user. By contrast, suppose that the function implements a classifier using some kind of software framework, rather than in a bespoke fashion. For example, rather than implementing the code for a decision tree completely by hand, the quality knowledge engineer may leverage an existing framework from programming machine learning algorithms. The use of such framework imposes some regularity on the function code, hopefully giving it a recognizable structure; the idea is to exploit this regularity to generate as much of the implementation code as possible. In

practice, we hope to shift the users' task from that of programming the function, to that of providing a sufficiently specific, but higher-level declarative description of it, that can be used to drive code generation.

To make this idea more precise, let us consider one specific programming framework for knowledge discovery, namely Weka [WF05]. Weka is an open, community-contributed Java-based environment that includes an extensible collection of machine learning algorithms for data exploration and model generation. It caters to three classes of user. The first class includes users who need to analyse their data using pre-defined algorithms through a friendly, graphical interface. The second class includes developers who wish to incorporate the algorithms as part of their own application. Finally, the third class includes developers who contribute new, possibly experimental algorithms to the community as add-on components to the framework. Quality knowledge engineers belong for the most part to the first and second class: to these users, Weka provides the means to iterate around the inner loop of the IQ lifecycle, by training and testing their classification model without any programming effort. After training, Weka models can be used to classify a new dataset, either interactively or programmatically, and they can be saved for future use.

Using Weka to implement quality functions leads naturally to the predictable program structure that we were hoping for; in fact, executing a Weka model on a new dataset amounts simply to preparing the dataset to conform to the expected input format, loading the model into the Weka environment, launching the execution, and reading the result. Our hypothesis is that we can exploit this program structure to perform automatic code generation. We propose a three-step approach to achieve this goal. Firstly, we incorporate into the IQ ontology a symbolic description of known frameworks, or implementation patterns. Hopefully, there are only a limited number of these frameworks, like Weka for example, for the specific domain of knowledge discovery. Secondly, for each new quality function, quality engineers should specify that the function will be implemented using one the frameworks known to the ontology. We will see shortly how this can be achieved by a simple annotation of the common WSDL interface for quality functions. Thirdly, a code generator interprets the annotation in order to produce a stub of the function implementation. By following this strategy, we expect to find that the more constraining the framework chosen by the user, the more complete the code in the resulting stub will be.

6.1.1 Code generation example

To make our strategy more precise, let us return to the Weka example, where we assume that the quality engineer decides to implement the quality function using a well-known decision tree algorithm, C4.5 [Qui93]. At the end of the training phase, the classifier is available as a model file, say `myC4.5-QA`. The essential steps for invoking the model, fairly self-explanatory, are as follows:

```
// Weka classifier invocation pseudo-code
inputDataset = readDataset(inputDatafile); // input data in Weka format
Classifier model = readModel("myC4.5-QA");
Instances labelledDataset = classify(inputDataset, model);
writeDataset(labelledDataset, outputDatafile);
```

As it turns out, many Weka algorithms, and in particular all classification algorithms, can be invoked using exactly the sequence shown above, regardless of their type (furthermore, most of the established algorithms in the Weka collection support a common I/O data format, so that, in particular, we can define helper classes to adapt between the quality service's message formats and the Weka data format). These considerations suggest that we can capture the common code structure as part of a template, where certain parameters account for the differences amongst its different instances. The following template for the Weka model defines the implementation of the core quality service operation, denoted as `assertQuality()`. The underlined variables in the template are parameters that get instantiated at code-generation time, rather that at execution time.

```
// Weka template pseudo-code, for illustration purposes only
ServiceOutputMessage assertQuality(ServiceInputMessage inputMessage)
{
      inputDatafile = unpackInput(inputMessage,
                                  $evidenceVariables,
                                  $classificationModel );
      inputDataset = readDataset(inputDatafile);
      Classifier model = readModel($modelFile);
      Instances labelledDataset = classify(inputDataset, model);
      writeDataset(labelledDataset, outputDatafile);
      outputMessage = packOutput(outputDatafile, $outputVariable);
      return outputMessage;
}
```

The values of the underlined parameters become constants in the generated code. For example, the value for `$evidenceVariables` consist of a set of labels, like `Masses`, `Coverage` and so forth, that correspond to the input evidence expected by the classification model, plus the name of the (single) output variable that holds the class label. Since the quality engineer chooses these names when the model is defined on the training data, it is natural to have them as constants in the code. Similarly, the set of class labels is specified at model definition time, and used again when the model is applied. The second parameter in this template, `$classificationModel`, holds the set of class labels. The reference to the model file, `$modelFile`, is also a generation-time parameter. Finally, the `unpackInput` and `packOutput` methods encapsulate the Weka-specific I/O transformation.

This detailed example suggests that, if we define a template for Weka-based quality functions, then all that is needed to instantiate the template are the values for a few parameters. These values are easy to obtain: `$evidenceVariables` and `$classificationModel` are obtained from the ontology, while `$modelFile` is the trained model. A single template, as noted earlier, is sufficient to describe the code required by the broad set of quality functions that implement Weka-defined classifiers. In the next section we propose a generalization of this idea, which will result in the three-steps approach to code generation that we have anticipated earlier.

6.1.2 Annotating functions for code generation

The Weka template is only one of several that we can define as part of the Qurator workbench. Another is the "classifier QA" template, which we can use to generate implementation skeletons for the more generic case of arbitrary classification functions that have a shared WSDL interface as their only known common feature. The collection of available templates is extensible: should quality engineers decide to use a new framework, we can define one or more templates to capture the programming patterns that it requires (more than one template may be required, e.g. the invocation sequence for a Weka algorithm that is not a classifier may be different from the one shown in the previous example). Clearly, frameworks that can be captured using a small number of templates will be the most beneficial.

We refer to a collection of quality functions that share the same template as a *family*. The "Weka family" and the "classifier QA" family are two examples. Code generation for a new quality function involves two steps. Firstly, quality knowledge engineers associate the function to a family. Secondly, they *annotate* each function's WSDL interface with the required values for the corresponding template's parameters. The code generator instantiates the template using the annotations to bind the parameters to their values.

To define families, we use the IQ ontology and more specifically the `QA` hierarchy of function classes introduced in Section 3.2.3. In the upper ontology, we define the new class `QA-family`, and the new property `QA-in-family` having domain `QA` and range `QA-family`. Each class in the `QA-family` hierarchy has an associated code generation template. For example, the two families mentioned above are represented by the classes:

$$\texttt{WekaFamily} \sqsubseteq \texttt{QA-family}, \text{ and}$$

$$\texttt{ClassifierFamily} \sqsubseteq \texttt{QA-family}$$

We adopt the same axiomatic approach used in Chapter 3 to assert the association between a QA class and a family in the ontology. For example, for class $\texttt{ClassificationQA} \sqsubseteq \texttt{QA}$ (the "umbrella" class for all classification quality functions) the axiom

$$\texttt{ClassificationQA} \sqsubseteq \exists\, \texttt{QA-in-family}\,.\,\texttt{ClassifierFamily}$$

associates to `ClassificationQA` the family represented by `ClassifierFamily`, which therefore becomes the default family for all classification quality functions (in particular, `PISCoreClassifier` is now a member of the family). Crucially, the hierarchical nature of the QA functions makes it possible to override this default. Let us suppose that the engineer defines an alternate version of the PI classifier, let us call it `PIClassifier-1` $\sqsubseteq$ `ClassificationQA`, and decides to provide a Weka-based implementation for it. The following axiom:

$$\texttt{PIClassifier-1} \sqsubseteq \exists\ \texttt{QA-in-family}\ .\ \texttt{WekaFamily}$$

associates `PIClassifier-1` to the `WekaFamily`, rather than to the `ClassifierFamily`. Figure 6.2 summarizes this configuration, also adding the `RankingFamily` to represent code that computes score models rather than classifications[1].

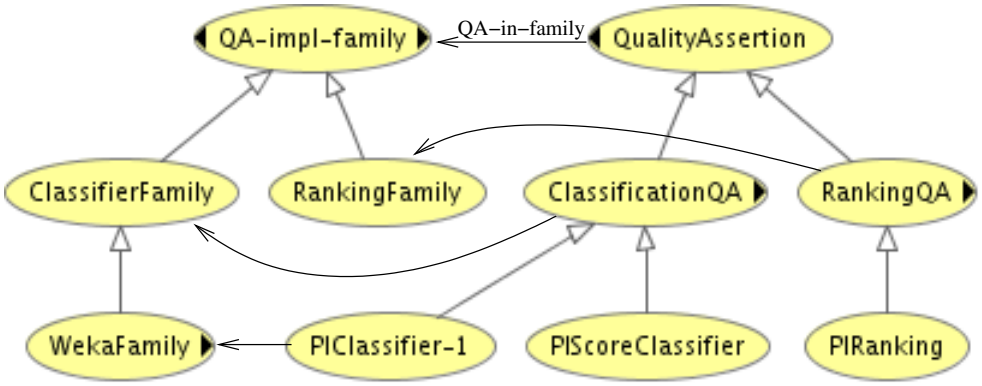


Figure 6.2: QA functions and related families

The quality knowledge engineer can use these new ontology classes to specify that a new quality service, say the service corresponding to `PIClassifier-1`, belongs to a certain family, and therefore it should be implemented using the corresponding template. There are several ways to record this information so that it can be used by the code generator. We adopt the approach, popular in the area of Semantic Web services, to annotate the WSDL interface of the service with references to classes in an ontology. We argue that this is a sensible approach to use in our case, since the IQ ontology includes a symbolic definition both of the function classes, and of the quality evidence that they require as input.

[1]The arrows on some of the class ovals indicate that they have sub-classes that are not shown.

As we have seen earlier, the latter is needed to provide values to the template parameters.

More specifically, we adopt the annotation conventions defined by SAWSDL [VS07a], an extension to WSDL that was recently proposed as a W3C standard (`www.w3.org/2002/ws/sawsdl/`). Stemming from the observation that WSDL describes Web service interfaces at a purely syntactic level, SAWSDL defines mechanisms by which service designers can add semantics to WSDL elements, in a principled way. Many of the concepts in SAWSDL are based on its precursor WSDL-S (`www.w3.org/Submission/WSDL-S/`), also a W3C submission. As explained in recent tutorials [She07, VS07b], SAWSDL extends the WSDL 2.0 standard [CMRSW06] by means of two new types of attributes. The first, called *modelReference*, is used to specify the association between a WSDL or XML Schema component and a concept in some semantic model. The second, called *schemaMapping*, is used to specify bi-directional mappings between the Type Definitions that appear in the WSDL specification, and ontology classes. Using *modelReference* to annotate a WSDL operation specification will suffice for our purposes. The following example, taken from [VS07b], illustrates its use:

```
<wsdl:operation name="order"
    sawsdl:modelReference=
         http://www.myontology.org#RequestPurchaseOrder">
    <wsdl:input element="OrderRequest"/>
    <wsdl:output element="OrderResponse"/>
</wsdl:operation>
```

The `modelReference` attribute represents an annotation that associates the semantic concept `http://www.myontology.org#RequestPurchaseOrder`, a class in the ontology defined by the namespace `http://www.myontology.org`, to the `order` operation.

In our case, the WSDL specification for quality functions exports a single operation, called `assertQuality`, as shown earlier in the Weka template example. Using the simple annotation mechanism just described, we can associate quality function classes to this operation:

```
<wsdl:operation name="assertQuality"
    sawsdl:modelReference=http://www.qurator.org:#PIClassifier-1">
    <wsdl:input element="assertionRequest"/>
    <wsdl:output element="assertionResponse"/>
</wsdl:operation>
```

Interestingly, the current literature on SAWSDL cited above suggests automatic service discovery, composition and integration as the main beneficiaries of semantic Web service annotation. Using annotations to automate the generation of Web service implementation code, as we do, seems to be a more novel application.

Once the quality engineer has produced a SAWSDL specification from the shared WSDL interface, the code generator performs the following operations, illustrated in Figure 6.3:

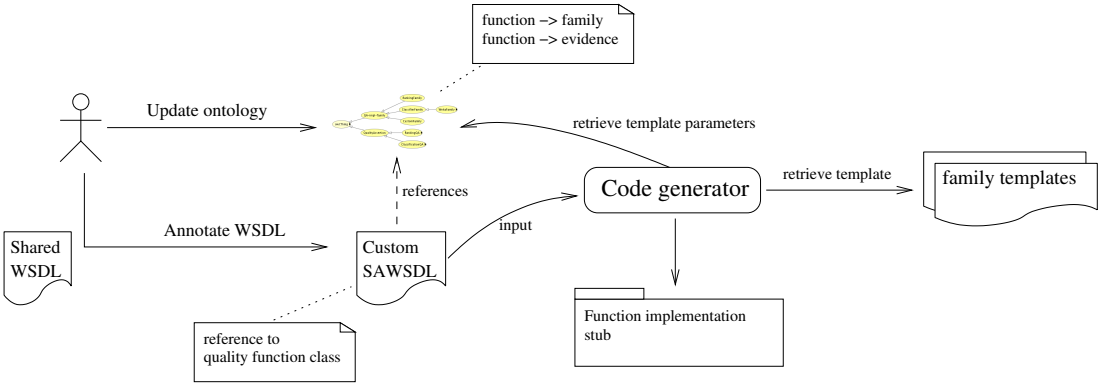


Figure 6.3: Semantic annotation of QA functions and code generation

- Parse the *modelReference* attribute associated to the `assertQuality` operation in the SAWSDL specification;
- Inspect the ontology in order to retrieve the additional semantic information associated with the quality function, i.e., (i) the function's family and (ii) the quality evidence types along with the classification model, if available (recall that the quality evidence types are associated to the quality function class in the ontology). These are used as values for the parameters[2];

[2]Note that some of the parameters, such as `$modelFile` in our example, are user-supplied rather than being automatically retrieved from the ontology.

- instantiate the code template associated to the family, by replacing the parameters with their values.

Note that it may be necessary to navigate through the ontology schema to retrieve the function's family. In the case shown in Figure 6.2, for example, `PIScoreClassifier`'s family is that of its parent, `ClassificationQA`.

6.1.3 Conclusions

In summary, we have proposed a simple algorithm to support users in task (1) of the IQ lifecycle, shown later in Figure 6.1. The algorithm is implemented as the "QV code generator" component of the Qurator workbench, shown in Section 6.4, Figure 6.12. The algorithm exploits the predictable structure of quality functions in order to automatically generate a code skeleton for the Web services that implement those functions. By doing this, we have shifted the user's focus from the service implementation to a higher-level specification from which the implementation can be generated. Having defined the IQ ontology earlier on, we have been able to achieve this goal by adding semantic annotations to the shared WSDL specification of the quality functions. The annotation is simple to perform (it amounts to a single new attribute), and yet it is sufficient, as we have seen, for the code generator to retrieve all the information required to instantiate a code template.

6.2 Quality-aware data processing

The second of the three remaining tasks identified at the beginning of the chapter concerns ways to make data processing quality-aware. In the previous chapter we have shown how we can add quality controls to workflows, using the Taverna workflow management system as an example. Here we argue that the applicability and usefulness of the Quality View concept extends beyond workflow processing, and specifically that it can also be applied to database query processing.

In this section we study this idea in detail for the case of XML data. Specifically, we are going to address the problem of specifying and executing XML queries that involve the computation of quality metadata and its use for the purpose of data selection. To motivate the choice of XML data as the object of this investigation, we observe that XML is commonly used to describe complex and

highly structured scientific experiments, for instance in transcriptomics and in proteomics (using the guidelines issued by the MGED society and by the HUPO initiative, respectively, as mentioned in our earlier discussion of Chapter 3). Indeed, a large number of these documents are stored in public repositories, such as PRIDE (`www.ebi.ac.uk/pride`) and PedroDB [KMGa04], and can be retrieved using XQuery.

We are going to show how Quality Views can be used to address the problem. Furthermore, we also propose a slight syntactic extension to the XQuery language, denoted *QXQuery*, which makes it easier for query designers to specify the invocation of Quality Views in the context of an XML query. The material presented here also appears in [EMS+07]. Before presenting our approach in detail, we review related work in this area.

6.2.1 Related work

The idea of enhancing traditional query processing with quality features is not entirely new. Most of the existing proposals, however, belong to the class of quality applications that we have called *provider-centric* in the introduction to this thesis, namely, those based on the assumption that the data provider is able to compute some form of quality metadata and associate it to the data. Once this is done, exposing the metadata at the level of the query language is then a relatively simple matter.

The literature offers both domain-specific and domain-agnostic versions of this approach. The work of Martinez and Hammer [MH05], for example, applies to biological data sources. The authors propose to extend a semi-structured data model in order to accommodate metadata regarding the quality of data stored in the sources. They define a fixed set of quality measures, such as *Stability*, *Density*, and a few more, which in their view are useful to assess the overall relevance of biological data sources. Once the provider defines procedures for computing these measures, these are exposed as part of the data model, hence they are available for querying. A similar idea, in the area of geo-spatial information systems, is developed by Mihaila *et al.* [MRE00]; here the goal is to expose quality metadata to the query language (a slight variant of SQL), in order to let users select appropriate sources in web information systems.

Domain-agnostic, quality-aware data architectures are more general, as expected. The DaQuincis system [SVM+04], for example, is based on the idea that

for a semi-structured data model, such as an XML schema, one can describe the quality of each element in a document, by matching it to a sibling "shadow" element. When this is done systematically over the entire schema, this approach results in a complete shadow quality document that has the same structure as the underlying data document (the resulting data model is called D^2Q, for "Data and Data Quality" model). One advantage of this idea, discussed in [MSC04], is that XML data and its metadata can be queried together. In the context of cooperative information systems, this model accommodates fine-grain quality information computed, for each source in the system, by a dedicated "Quality Factory". Under this assumption, users who wish to retrieve data from the cooperative system may express quality requirements, which a *Quality Broker* translates into queries on the D^2Q model. The result, ideally, is a selection of cooperative sources that conforms to the quality requirements.

Regarding the provider-centric approaches exemplified by these efforts, we question the main assumption that providers have the means, or the motivation, to compute sound and complete quality metadata. To repeat one of the main observations made in Chapter 2, we note that under this assumption users must understand and accept the quality metrics defined by the provider; and furthermore, quality-aware queries can only operate on those metrics. We have argued at the beginning of the thesis that this approach limits the usefulness of current information quality architectures. Indeed, in addition to lamenting its lack of flexibility, users may also find it difficult to trust the quality metadata advertised by the provider. This is because there is no incentive for a provider to acknowledge that some of its data is of low quality, much in the same way as any sellers would not easily recognize the limitations of their products; and at the same time, it is difficult for users to independently test the validity of the provider's quality claims, unless the origin of the metadata is properly and thoroughly documented.

A more promising approach, in our opinion, is represented by the XQual language [BE04]. XQual is based on user-defined quality dimensions (chosen, however, from amongst a fixed pool of dimensions by setting their relative priorities) and on user-specified *contracts*; these define constraints over the chosen dimensions, that the query evaluator should satisfy if possible. The IQ contracts of XQual are significant because they are an early attempt to allow consumers of information to define their IQ preferences in a declarative and machine-manipulable form. By contrast, we argue that, consistently with our user-centred

quality model, the users who issue the queries should have complete control over the type of quality assessment that is performed on the data. In the rest of this section we illustrate how we achieve this goal using Quality Views.

6.2.2 Technical approach

Our approach is based on the observation that, if a data element can be described using one of the semantic types known to the IQ ontology (i.e., as a sub-class of `Data Entity`), then we can potentially use Quality Views to associate quality values to that data element. As we know, this only requires the definition of suitable Annotation and Quality Assertion functions. Therefore, if such data elements are part of an XML document, and if we can use XQuery (specifically, XPath expressions) to refer to those elements, then it should be possible to compute a Quality View on that data. Furthermore, if we can express the association of quality values to data elements using XML, then we can use XQuery to seamlessly query the quality metadata along with the original XML data. Conveniently, this is indeed the case for our current implementation of Quality Views: they take an XML document containing a list of datarefs as input, and return an XML document containing the computed quality values (we can view this as an XML representation of the *environment* data structure that we have repeatedly described, for example in Section 6.1). Furthermore, note that quality workflows, i.e., the results of a Quality View compilation, can be viewed generically as services: they can be invoked through a well-defined interface, as long as a runtime environment (the Taverna workflow engine, in this case) is available. To the extent that we can interface an XQuery engine with a workflow engine, therefore, we can make use of Quality View services from within a query.

We argue therefore that our existing Qurator infrastructure is sufficient for our purposes. To make this intuition precise, let us consider an XML document taken from the PRIDE repository, mentioned earlier. Here is a fragment of the document:

```
<DBSearch>
    <username>David Stead</username>
    <id_date>2003-06-02</id_date>
    <DBSearchParameters>
    (...)
    </DBSearchParameters>
    <ProteinHit>
        <masses_matched>5</masses_matched>
        (...)
        <Protein>
            <accession_number>6957.1</accession_number>
            <gene_name>CDC48 (by homology to S. cerevisiae)</gene_name>
            <organism>Candida glabrata</organism>
            <description>microsomal protein of (...)</description>
        </Protein>
    </ProteinHit>
</DBSearch>
```

Our specific goal is to write an XQuery that return the accession numbers found in the document (exemplified by the underlined element), at the same time filtering the result according to some quality values associated to the accession numbers. These quality values are computed using a Quality View. As a starting point we use the following simple XQuery, that returns a list of triples of the form ⟨ accession number, experimenter's name, experiment date⟩:

```
<html>
    <ul> {
    let $inputDoc := "PSF1-ACE2-CG.xml"
    for $d in doc($inputDoc)//DBSearch
    return <li>
        accession number:  {data($d/ProteinHit/Protein/accession_number)},
        name:  {data($d/username)},
        date:  {data($d/id_date)} </li>
    } </ul>
</html>
```

Then, we define a QV to compute a PI score for each of the proteins indicated using their accession numbers. This QV, omitted for brevity, is similar to the one shown in Figure 4.1 on page 99. By incorporating this QV as part of

the XQuery above, we will then be able to select a relevant subset of the proteins based on their PI score, or alternatively, based on the discretization of the score into the three classes computed by `PIScoreClassifier`, i.e., {"low_PI_score", "close_to_avg_PI_Score", "high_PI_score"},

Let us suppose that we have compiled this QV into a Taverna workflow. As suggested earlier, the technical means by which we invoke this QV is by executing the workflow from the XQuery processor. We have implemented this mechanism using the Saxon XQuery engine (`saxon.sourceforge.net`) and its facility for invoking external functions. As shown in Figure 6.4, the Qurator workbench maintains a registry of executable Quality Views, that associates their symbolic name, for instance `uri:PedroQV`, to a workflow. The Saxon XQuery processor can request the execution by passing the symbolic name and the input dataref to the workbench, through an adapter. The workflow is submitted to the Taverna engine, and finally the results are returned to the query processor.

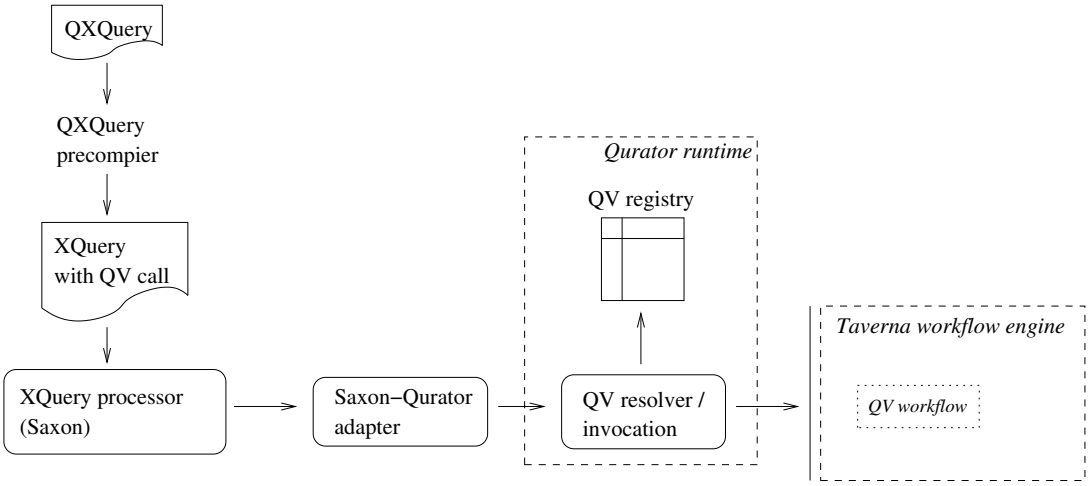


Figure 6.4: QXQuery execution model

In the figure we also show a preliminary step where a quality-extended XQuery, denoted *QXQuery*, is pre-compiled into a standard XQuery prior to being executed. This step allows users to specify the invocation of Quality Views using an intuitive syntax, making the mechanism more friendly to use. While we defer the description of this aspect until the next section, we observe that the standard XQuery language, however, is sufficiently expressive to support our invocation mechanism. The query in Figure 6.5, for example, extends the previous by adding a Quality View invocation (the main enhancements are underlined).

```
    declare namespace qvi="java:org.qurator.util.QVInvoker";
    <html><ul> {
1:   let $inputDoc := "PSF1-ACE2-CG.xml"
2:   let $classifiedData := qvi:QVInvoke(
3:        qvi:new(),
4:        "PedroQV",
5:        doc($inputDoc)//ProteinHit/Protein/accession_number,
6:        ($inputDoc))
7:   for $x in $qvResults/enrichedData/EDItem,
8:        $d in doc($inputDoc)//ProteinHit/Protein
9:   where $x/assertionValue/@AssertionTagValue ="high_PI_score"
10:       and fn:contains($d/accession_number, $x/@dataRef)
11:  return <li>
12:       accession number:  {data($d/accession_number)},
13:       description:  {data($d/description)},
14:       Hit Ratio:  {data($x/annotationValue
15:                          [@varName="HitRatio"]@varValue)}</li>
    }
    </ul> </html>
```

Figure 6.5: XQuery with Quality View invocation and quality-based selection

In the example, the external function invocation appears on line 2.[3] The input dataset required by the QV is constructed by the XPath expression on line 5, which extracts the proteins' accession numbers from the document.

The result of the QV execution, itself an XML document, is stored in the variable `$qvResults`. This new *quality document* carries the entire contents of the Quality View environment, where each dataref is associated to a vector of quality evidence and quality assertion attributes, as prescribed by Qurator. The `for` statement on line 7, along with the condition on line 10, effectively computes a "join" between the input document and the quality document, allowing the query to construct an output document where the IDs for the high-score proteins are listed along with any quality evidence values like the Hit Ratio, as shown on

[3]The namespace `qvi` is a URI that specifies the Java class to be loaded by the call. This, as well as the requirement to create a new instance of this class (`qvi:new()`), is part of Saxon's proprietary mechanism for invoking external functions.

lines 14-15. Thus, with this mechanism the users are now able both to filter data elements based on their quality metadata, and to include the metadata itself in the output, making it available to the invoking environment.

It is important to note that the Xpath expressions used to extract elements from the quality document are based on a known XML schema, defined as part of the Qurator framework. The complete schema is depicted graphically in Figure 4.2 (Chapter 4, page 100). Indeed, such predictability of the quality document structure is the key to providing a higher-level syntax for invoking Quality Views, resulting in simpler and more readable XQuery code as explained next.

6.2.3 QXQuery: a syntactic extension to XQuery

One problem with the type of query illustrated above is that their design requires knowledge both of the XML schema for quality documents, and of the specific mechanism needed to carry out the Quality View invocation. We note, however, that these details realize a regular pattern that is common to all quality-enhanced queries. Here we show how we can exploit this regularity to extend XQuery with a higher-level syntax that can be translated into plain XQuery prior to execution. Since the query above already contains all the functionality required to execute quality-enhanced queries, these extensions are merely syntactic sugar that can be handled by query pre-processing.

The new syntax is designed to allow query designers to express three main capabilities: specifying the dataset to be used as input to the QV, invoking the QV, and accessing its results. For the first two of these capabilities, we propose a new clause with the following syntax:

```
<QVClause> ::= using quality view <qualView>
               on <PathExpr> with key <PathExpr>
               as <VarName>

<qualView> ::= <URI> '(' ( <ExprSingle> ( ',' <ExprSingle> )* )? ')'
```

where the productions `<PathExpr>`, `<ExprSingle>` and `<VarName>` are all defined by the standard XQuery syntax. The example in Figure 6.6 illustrates its usage.

```
let $inputDoc := doc("PSF1-ACE2-CG.xml")
using quality view "uri://PedroQV"() on
$inputDoc/ProteinHit/Protein
     with key /ProteinHit/Protein/accession_number
as $qvResults
```

Figure 6.6: QXQuery fragment

In the example, the `<qualView>` construct specifies a reference to the quality view to be invoked by the query, for instance `"uri://PedroQV"`; this reference, in the form of a URI, will be resolved by the Qurator QV registry as described in the previous section. The `<PathExpr>` construct after the `on` keyword is used to indicate the input dataset to which the QV is applied, in this case the `/ProteinHit/Protein` document fragment. Within this fragment, we specify the datarefs with which the QV will associate the quality values, using the `key` keyword and a relative path expression, i.e., `/ProteinHit/Protein/accession_number`. Finally, we use the `as` clause to name a fresh XQuery variable that will hold the resulting quality document. The QXQuery pre-processor generates the code to produce an external function invocation by interpreting this new clause; the result is the invocation code shown earlier (lines 2-6).

Let us now see how this QV declaration is used in the context of a full query. Its effect, as observed, is to create a new variable that holds a reference to a quality document. A QV clause is therefore similar to one of the other XQuery clauses that bind values to new variables, namely `let` and `for`. When we expand the syntax of the standard XQuery FLWOR production that accounts for these clauses, we obtain:

```
<FLWORExpr> ::= (<ForClause> | <LetClause> | <QVClause>)+
                <WhereClause>? <OrderByClause>?
                return <ExprSingle>
```

Note that the only change here from the standard XQuery syntax is the addition of the `<QVClause>` non-terminal. A consequence of this addition is that a query can now contain zero or more quality view clauses, and that they can be interleaved between arbitrary numbers and combinations of `let` and `for` clauses, as necessary

to express the query requirements.

As a continuation of the example in Figure 6.6, here is a complete QXQuery:

```
<html> <ul> {
let $inputDoc := doc("PSF1-ACE2-CG.xml")
using quality view "uri://PedroQV"() on
    $inputDoc/ProteinHit/Protein
    with key /ProteinHit/Protein/accession_number
    as $qvResults
for $x in allResults($qvResults),
    $d in $inputDoc/ProteinHit/Protein/
where hasQuality($x, "PIScoreClassifier") = "high_PI_Score"
and fn:contains($d/accession_number, getDataRef($x))
return <li>
    accession number:  { data($d/accession_number) },
    description:  { data($d/description) },
    Hit Ratio:  { data(getAnnotationValue($x, "HitRatio") </li>
} </ul> </html>
```

The user-defined XQuery functions, underlined, provide access to the XML elements of the quality document, resulting in a query that is functionally equivalent to the XQuery of Figure 6.5. For example, `allResults` returns a list of document fragments, one for each input dataref, containing its associated quality evidence and assertions; it is defined simply as:

```
define function allResults($x) {
  return $x/enrichedData/EDItem
}
```

We can observe its effect by inspecting the fragment of a result of executing a QV, given in Figure 6.7. Similarly, `hasQuality` takes one of these document fragments and returns its quality assertion values:[4]

```
define function hasQuality($item, $tagName) {
  return $item/AssertionValue@[AssertionTagValue = $tagName]
}
```

[4]Note that this can be easily generalized to a function that returns the value for some input key.

```
<enrichedData>
    <EDItem dataRef="9413.1">
         <assertionValue AssertionTagValue="low_PI_Score"
                         AssertionTagName="PIScoreClassifier" />
         <annotationValue varName="Mass"
                         varOntType="q:Mass" />
         <annotationValue varName="Masses"
                         varOntType="q:Masses" />
         <annotationValue varName="ELDP"
                         varOntType="q:ELDP" />
         <annotationValue varName="Coverage"
                         varOntType="q:MassCoverage" />
         <annotationValue varName="HitRatio"
                         varOntType="q:HitRatio" />
    </EDItem>
    (...)
</enrichedData>
```

Figure 6.7: Example of a quality document fragment

6.2.4 Conclusions

To summarize, in this section we have addressed the problem of specifying and executing XML queries that involve quality metadata. We have shown that we can achieve this goal by executing Quality Views as part of XQuery processing, using standard XQuery functionality. In addition, we have presented a simple syntactic extension to XQuery that makes it easier for users to specify how Quality Views should be invoked, and how to use their result for data selection.

With this, we have extended our support to the IQ lifecycle (specifically to task 6 in reference Figure 6.1) to include a new form of quality-aware processing, namely XML query, in addition to the workflow processing described in the previous chapter. The component denoted *QXQuery pre-compiler* implements this functionality in the workbench, as shown in Figure 6.12 of Section 6.4. In the next section we address the remaining issue, represented by task 7 in the same figure, namely the analysis of Quality View results.

6.3 Quality provenance

Analysing the results of a Quality View is the final step in the outer loop of the IQ lifecycle (task 7 in the reference lifecycle figure). This is where users collect the results of their experiment, knowing that those results are now viewed through their chosen "quality lens". Our goal at this stage is to help users understand the impact that the additional quality features have had on the data, specifically by requesting an explanation of why a certain data element has been placed in a certain quality class (in particular, why it has been accepted / rejected) during the execution of a certain experiment. Furthermore, users may want to analyse the quality classification of their data not just on a single experiment, but for variations of the experiment, obtained for example using different quality parameters each time (for instance, different action expressions): is a certain data element consistently rejected, for example, or is its acceptance particularly sensitive to a quality configuration? These become legitimate questions when one considers that the point of the outer loop in the lifecycle is to allow users to rapidly modify and re-deploy Quality Views, leading to the inexpensive generation of experiment variants.

To address these questions we propose the new notion of *quality provenance*, or "provenance of Quality Views". The term *provenance*, and more specifically *data*

provenance, has become popular over the past few years to indicate, in a broad sense, "information that helps determine the derivation history of a data product, starting from its original sources" [SPG05]. The term has acquired a more specific meaning in e-science, where provenance is interpreted as a detailed record of an experiment, defined in such a way that scientists can use it to analyse, validate and verify the results of the experiment, or to replicate it [GGS+03, ZWG+04]. In this sense, provenance information is effectively metadata that describes the process by which information, in this case an experimental result, has been obtained.

Here we address the problem of how to best describe, collect, store and exploit quality provenance. To set our discussion in the context of current research and solutions in provenance modelling we begin, in the next section, by introducing the framework proposed by Simmhan *et al.*, in their recent survey on data provenance in e-science [SPG05].

6.3.1 Characteristics of provenance

The following characteristics are used in [SPG05]:

Intended use of provenance: Common uses for provenance information include estimating the quality and reliability of the data (a need highlighted at the NSF Workshop on Data Management for Molecular and Cell Biology [JO04]), and explaining its derivation, for example to justify decisions that are based on the outcome of a process. We find examples of this type of applications in specific domains, notably healthcare [KVVS+06]. Other uses include keeping an audit trail of the process for validation purposes [WMF+05], tracking resource usage, and establishing ownership of data (as well as determining liability in case of data errors);

Types of resources for which provenance is collected: While a complex process may involve many elementary process components and many types of data, it is often the case that provenance metadata is only relevant for some subset of them. A number of workflow processes, for example, only serve as adapters, to change the format of the data without contributing to its content. Likewise, some of the data may be less central to the outcome of an experiment than other. It is therefore important for a provenance architecture to let users focus on the processes and data types of interest, and at the right level of detail.

Conceptual model for provenance: Recent proposals, notably by the Provenance EU-sponsored project (`www.gridprovenance.org`), include a domain-independent conceptual model for provenance that is based on the notion of *p-assertion*, an abstract representation of tasks and of the messages exchanged by the tasks. An alternative approach, embraced mainly by the ^{my}Grid project, advocates the use of Semantic Web technology to represent e-science provenance [ZWG+04]. In this approach, the elements of the provenance model are viewed as instances of ontology classes; this opens the way to the interpretation of provenance metadata using knowledge management techniques, notably Description Logics and automated reasoning.

Collection and storage models for provenance: Depending on the type of process, provenance information can be collected manually, i.e., in the form of an "electronic log book" maintained by scientists, or when the process is automated, from within the process execution environment itself, as is the case for Taverna;

Provenance dissemination and analysis: This is the last of the characteristics used in [SPG05]; it describes the means available to the users to access and analyse provenance information. Here, interesting recent research has been focusing on how complex workflow provenance models can be presented to the user. In *Zoom*UserViews* [CBD06, BBD07], for example, users may choose to "zoom" in and out of a workflow, i.e., to reveal or hide its sub-workflows; the system then automatically modifies the provenance information presented to the user, to be consistent with the chosen level of abstraction.

We are now going to describe the quality provenance model in terms of these characteristics.

6.3.2 The Qurator quality provenance model

The goal of quality provenance (its intended use) is to provide users with the ability to trace quality-based decisions to the quality evidence and functions that compute the elements used in the decision. It follows that quality provenance is naturally associated with the input dataset of a Quality View, at the granularity of

the individual data element. In our example, we associate provenance metadata to individual protein identifier. In terms of the IQ ontology, this means that provenance metadata is associated to classes in the `DataEntity` hierarchy.

Representing and collecting quality provenance

Although the *p-assertion* model mentioned earlier may be emerging as a standard for provenance representation, we have instead chosen a semantic approach for the quality provenance model, i.e., one where the model elements can be interpreted using the IQ ontology. This choice makes quality provenance consistent with the overall Qurator quality model: since the elements of a Quality View are defined in the ontology, it should be possible, we argue, to use the ontology to describe the effect of their execution as well. As a consequence of this choice, the model is expressed using an RDF graph. This has the additional benefit that we can perform provenance analysis using a declarative query language (i.e., SPARQL), as discussed in the next section.

Following the semantic approach, we propose a quality provenance model consisting of two parts. We describe it using an example. The first part, called the *static* model, partially describes the structure of a Quality View. The model, generated by the Quality View compiler, is an RDF graph with two resources, i.e., the two nodes on the left in the graph of Figure 6.8.[5] In this example, the QV consists of a single Quality Assertion, `PIScoreClassifier`, and a single action, called `initial_filter_action`. The action resource, at the top, carries the definition of the action expression; while the QV resource at the bottom is the root of a graph that describes its input and output variables. Note that each input variable is itself an RDF resource, consisting of a name (the literal) and a type, identified by the property `rdf:type`. As expected, the type is an ontology class, e.g. `q:Coverage`, `q:Masses`, etc. (the QA resource itself has a semantic type, namely the resource `q:PIScoreClassifier`).

A *dynamic model* for quality provenance is an RDF graph that is populated during each workflow execution, and contains references to the static model. Its purpose is to capture the values of the variables involved in the workflow, i.e., those that appear in the static model, as well as the effect of the actions –also defined in the static model. Each new execution of the same quality workflow

[5]These are anonymous nodes, or *b-nodes* in RDF terms. The ovals in the graph are RDF resources, while the square boxes are *literals*, i.e., constant values.

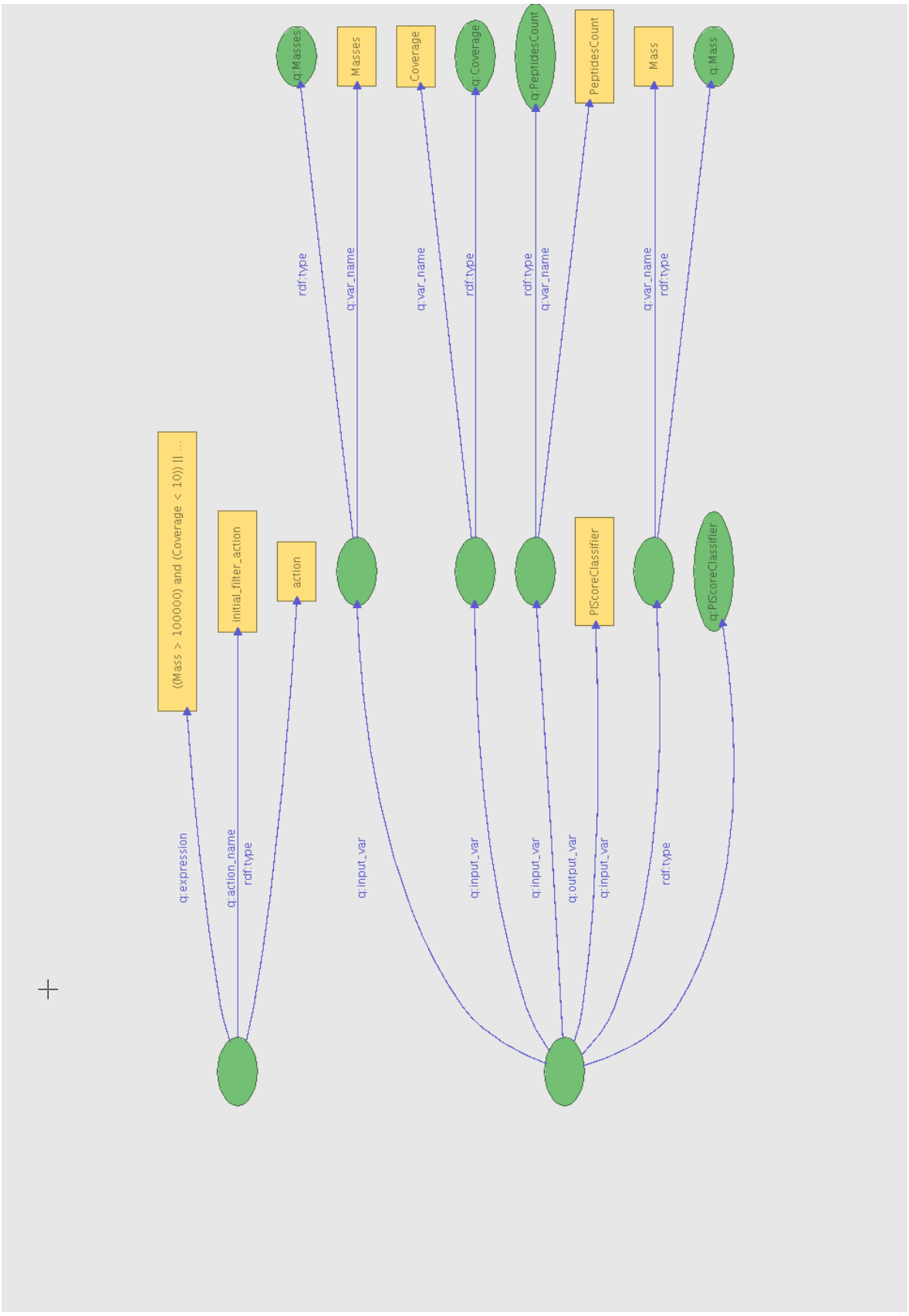


Figure 6.8: Static quality provenance model (example)

results in the generation of a new dynamic model (for the same static model).

From a technical standpoint, the mechanism for collecting provenance information into the dynamic model exploits Taverna's ability to accept third party monitoring components and to send notifications to them for a variety of events that occur during workflow execution. Using this notification pattern, we have developed a quality provenance module that monitors the activity of individual processors in the quality workflow, as well as the content of the messages they exchange. The result is the quality provenance component of the workbench, shown in Figure 6.12.

Let us describe the dynamic model through an example, considering the RDF graph shown in Figure 6.9 (we are showing a fragment of the complete model obtained at the end of one single workflow execution), and in particular its leftmost resources. The resource at the bottom left identifies a workflow execution; each execution is given a unique identifier (i.e., the resource `PP6...`) that serves as a reference for the other nodes, which are related to it through the `q:workflow` property. This common reference defines the scope for all the resources associated with a single execution. It ensures, for example, that we can retrieve the entire quality provenance graph for one execution independently from that of other executions (using a query with the constraint that the workflow be the same for all resources returned), while at the same time allowing for queries over multiple executions, for example "all protein datarefs in class *fail*", simply by ignoring the workflow identifier.

The subgraph rooted at the next resource above the workflow identifier describes the binding of a variable (`Masses`, of type `q:Masses`) to a value (20), within the context of a workflow execution and for the dataref indicated by the property `q:data_item`, in this case protein `P33B97` (the bindings for other variables and for other datarefs are omitted for clarity). The graph associated to the top resource has a similar structure; the binding is relative to the same dataref, but this time for the output variable of `PIScoreClassifier`, i.e., the value of the Quality Assertion (`close_to_avg_score`). Finally, the second resource from the top accounts for the action that was taken during the same execution, on the same dataref (`fail`, in this case).

In summary, we use a combination of a single static provenance model, created at QV compilation time, and multiple dynamic models, each generated during workflow execution, to capture detailed information regarding the execution.

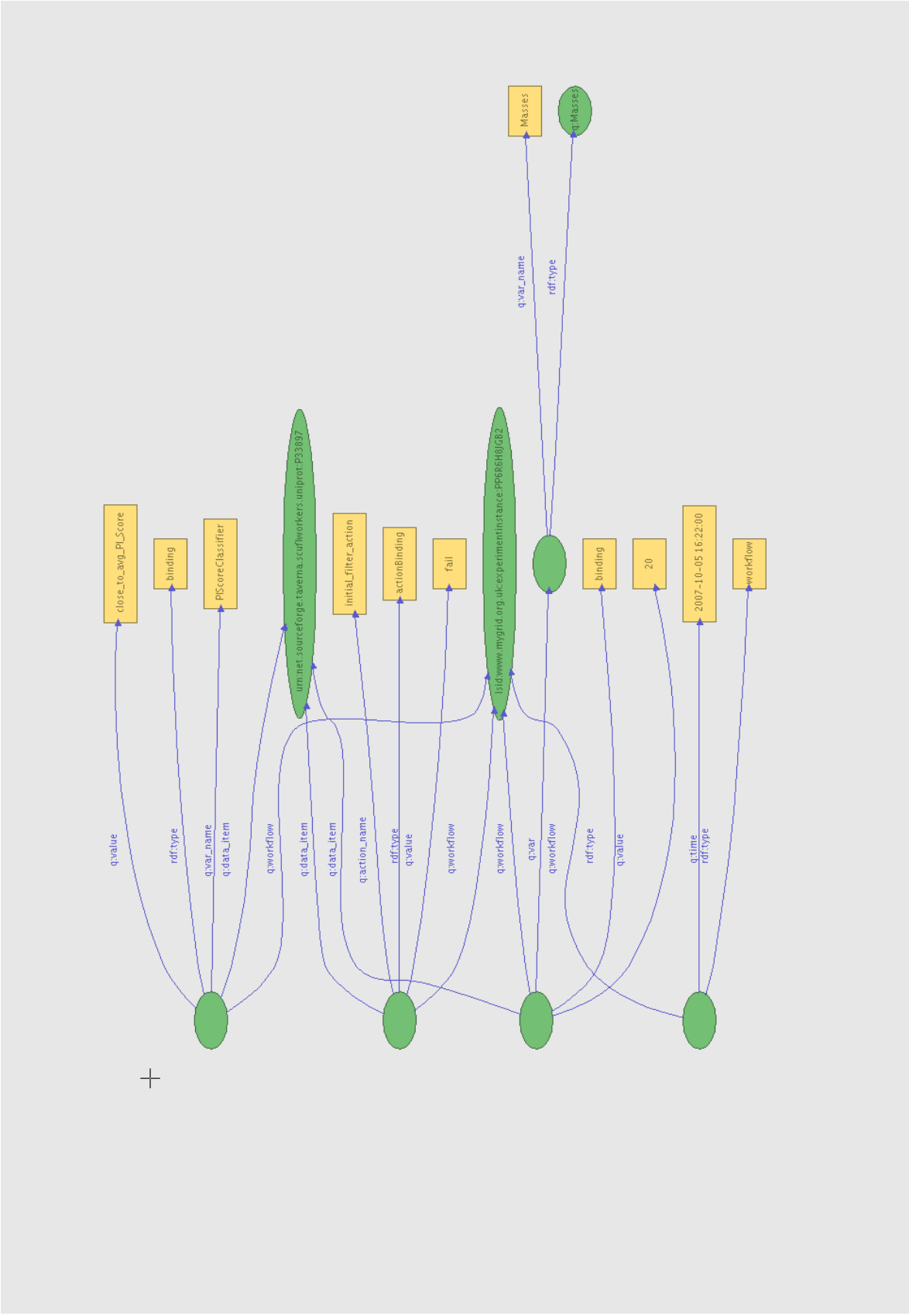


Figure 6.9: Dynamic quality provenance model (example)

Next, we see how we can use a variety of queries to exploit the model for provenance analysis.

Exploiting quality provenance

As part of the Qurator workbench we offer a programmatic interface for querying the model, using the SPARQL query language (the emerging W3C standard RDF query language[6]). In addition, however, we have also defined a graphical user interface for common types of provenance analysis, based on a set of pre-defined queries. Using the interface, users can: focus on a single dataref, retrieving the variable bindings and its quality classification, as we have seen in the previous section; visualize the entire set of quality values computed by a QA function over the entire dataset; visualize the quality classifications of a dataref across a series of workflow executions; and more.

Figures 6.10 and 6.11 show examples of the interface being used to visualize the provenance model described in the previous section. In the first figure we use the interface to trace the assertion values for a single data element, while in the second, users may select the quality assertion at the top right (Figure 6.11) to reveal the evidence types that it depends on, along with the assertion value for each input data element.

6.3.3 Conclusions

In this section we have addressed the problem of analysing the results of Quality View executions, as defined in task 7 of the IQ lifecycle (Figure 6.1). Our approach is based on the definition of a narrow-scope, dedicated provenance model for quality workflows, that we have called *quality provenance*.

Several broad-scope provenance models have been developed recently (the Taverna provenance module, for example). Oblivious of the workflow semantics, these models capture a generic form of provenance, i.e., a trace of the messsages exchanged by processes. By contrast, the quality provenance model is dedicated entirely to describing the execution of Quality Views. We have shown in this section that this limited scope makes the model suitable to answer specific users questions regarding quality-based decisions taken during the execution of a scientific workflow; the resulting workbench component, denoted *quality provenance*

[6]http://www.w3.org/TR/rdf-sparql-query/

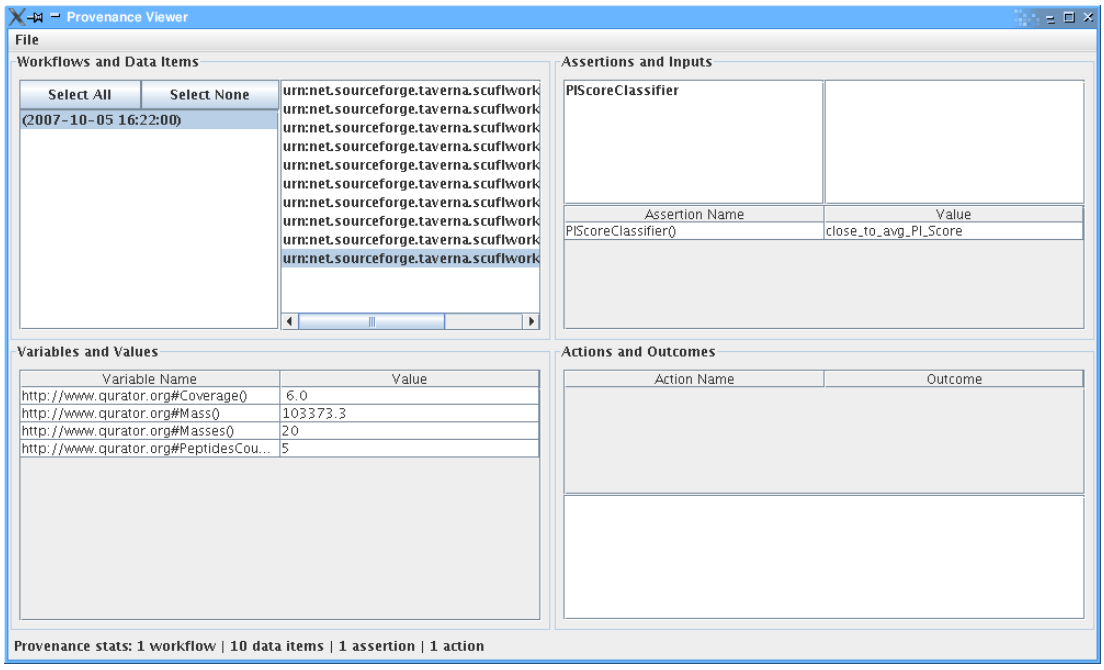


Figure 6.10: Qurator Provenance GUI - example one

in Figure 6.12 below, fulfills the goal of the "results analysis" task of the lifecycle.

6.4 Summary: the Qurator workbench

At the beginning of the thesis we argued that quality-awareness in data processing is the result of an experimental process, and we have proposed the IQ lifecycle as a means to describe such process. Having now completed the presentation of our technical solutions to support the lifecycle tasks in the outer loop, we conclude the chapter by presenting the overall Qurator workbench, depicted in Figure 6.12, and clarify how each of its components support the lifecycle.

As shown in the figure, we make a distinction between *Qurator core components*, on the left side, and *user-supplied services* on the right. The latter correspond to Quality Assertion and Annotation functions; their invocation is coordinated by the Taverna workflow engine. Each of the core components serves a purpose in the context of the lifecycle, as follows.

- The implementation of Quality Assertion functions, task 1, is supported by the QV code generator, described earlier in this chapter;
- The exploration and update of the IQ ontology, task 2, is supported by the

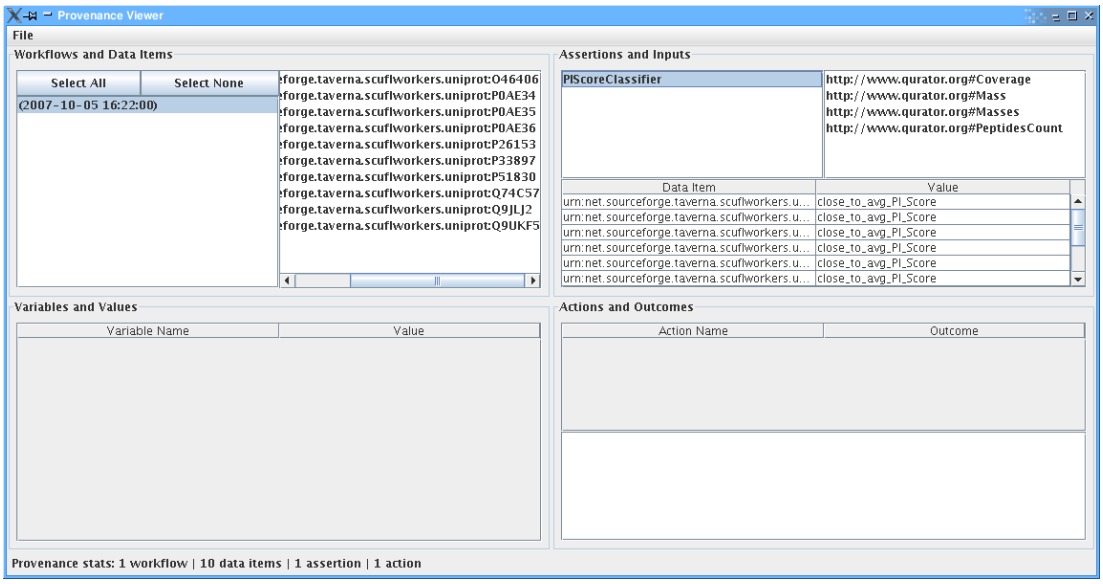


Figure 6.11: Qurator Provenance GUI - example two

ontology browser and consistency checker, introduced in Chapter 3;

- Quality view specification (task 3) is supported by the homonymous component, through a graphical user interface, as described in Chapter 4;
- QV compilation (task 4) is the responsibility of the QV compiler, which uses the QA registry to resolve the logical names of functions and map them to user-supplied services (shown on the right). The compiler is discussed in Chapter 5;
- QV integration (task 5) is again realized by the component with the same name (see Section 5.6), using user-defined deployment descriptors;
- Regarding quality-aware data processing, the workbench provides both runtime support for the execution of quality workflows, and compilation support for the QXQuery extension for XML query, as discussed earlier in Section 6.2. Several components contribute to the runtime support: in addition to the QV registry and the QV invoker, described earlier, the *Data Enrichment* service is responsible for retrieving quality evidence values from one or more of the QE repositories, shown at the bottom, according to the asyncronous process pattern described in the preceding chapter (Section 5.4). To recall, the pattern prescribes that annotation functions, shown in the user-defined services space, write quality evidence values to the repository,

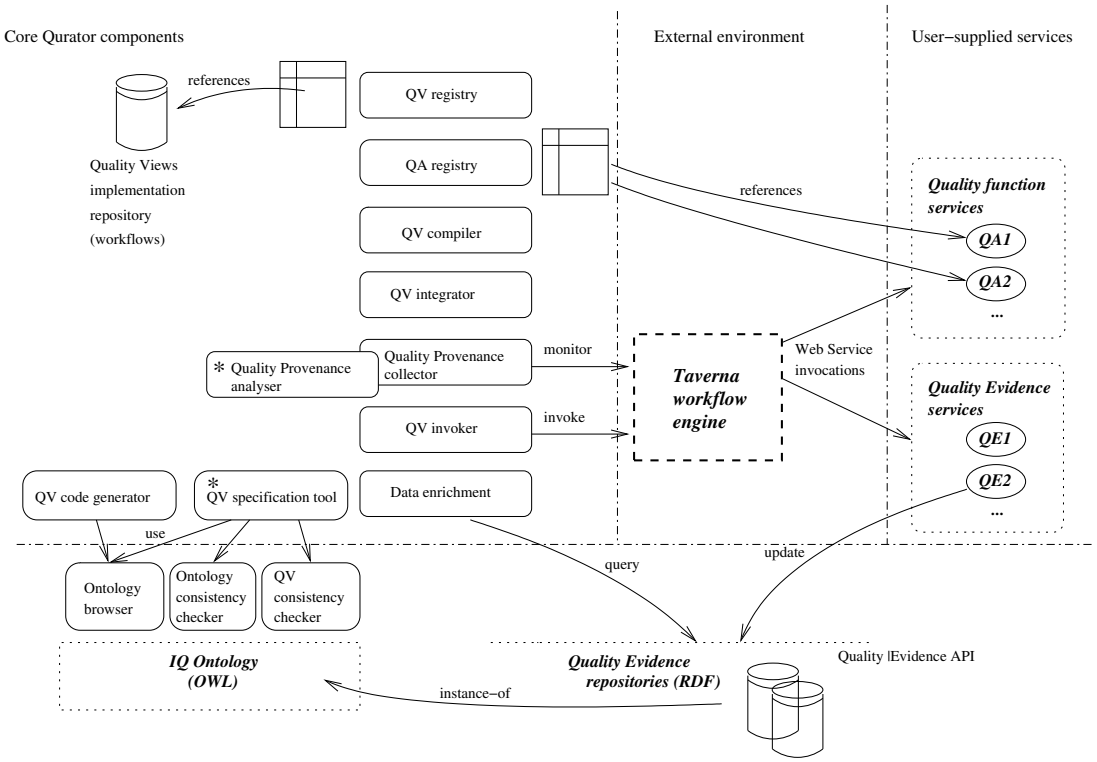


Figure 6.12: Qurator workbench architecture. Components with a (*) include a user interface

to make them available later to the QA functions. The Qurator programming interface makes multiple RDF repositories, used to store the quality evidence values, accessible to the Data Enrichment service;

- Finally, result analysis (task 7) is supported by the quality provenance collector and analyser components described in the previous section.

These relationships are summarized in Table 6.13. With the implementation of the Qurator workbench we have precisely defined the extent to which our user-centric approach to information quality lends itself to the automated processing of personalised quality definitions in the context of e-science. In the next, concluding chapter we identify specific limitations of the model and implementation, and outline a plan for further progress in this area of research.

Lifecycle task	**Workbench component**
Implementation of Quality Assertion functions as services	QV code generator
Exploration and update of the IQ ontology	Ontology browser and Consistency checker
Quality view specification	Quality view specification tool
QV compilation	QV compiler, QA registry
QV integration	QV integrator
Quality-aware data processing	QXQuery pre-compiler QV registry, QV invoker Data Enrichment Interface to Quality Evidence repositories
Result analysis	Quality provenance collector and analyser

Figure 6.13: Summary of Qurator workbench support to lifecycle tasks

Chapter 7

Conclusions

Not enough evidence God! Not enough evidence! [†]

In this chapter we summarize the resarch contributions presented in this thesis and discuss lines of further research necessary to help overcome the current limitations of our work.

7.1 Summary of research contributions

The research described in this thesis has been motivated by the observation that the assessment of information quality in e-science is an important and largely open problem. As we noted in the introduction, what makes the e-science proposition attractive to the scientific community is the large-scale availability of third-party, public scientific results, and their reusability in further data-intensive scientific experiments. However, there is a risk that inadequate quality control for these experiments may lead to erroneus data being released into the public domain, with potentially damaging consequences to further experiments.

In this thesis we have focused on the role of the user scientists in assessing the quality of third-party information used in their experiments. Users, we have argued, view quality assessment as the problem of establishing whether the information they need is fit for use in the context of their application. Two elements

[†]B. Russell, upon being asked what he would reply if, after dying, he were brought into the presence of God and asked why he had not been a believer. Quote found in the Stanford Encyclopedia of Philosophy, in the entry for "Evidence", 2006.

contribute in making the users' determination. Firstly, a model for estimating the likelihood of errors in the data. Such a model, although objective, is typically based on some indirect evidence that may be indicative of the presence of errors; therefore it is predictive, in the sense that the presence of errors in the data cannot be established with certainty. The second element, distinctly subjective, is a criterion for data acceptability given the model. Considering the uncertainty associated with quality prediction, this second element reflects the users' personal tolerance to errors, i.e., their propensity to the risk of using faulty data.

These two aspects rarely emerge explicitly as part of e-science experiments, although some elements of quality control may be present in latent form, for example as part of scientific workflows. We have coined the term *quality knowledge* to denote the scientists' latent knowledge about information quality. The research hypothesis that we have pursued in this thesis is that it is possible, and beneficial, to make quality knowledge explicit, by exposing it as a "first-class citizen" as part of scientific data processing. The benefit of making quality knowledge explicit is mainly in its reusability: we envision an incrementally growing library of user-defined quality functions that can be added with little effort to the users' data processing evironments as commodity components, making the experiments "quality-aware".

In our research we have been exploring the feasibility of this idea. In particular, we have postulated that the management of information quality follows a particular *IQ lifecycle*. We have used the lifecycle both to describe our specific research objectives, in the introductory chapter, and to summarize our achievements, in the previous chapter. The lifecycle, illustrated in Figure 6.1, consists of two interconnected loops. The inner loops involves the discovery of new quality knowledge. Here *quality knowledge engineers* define quality functions, the objective component of the users' quality model. The loop accounts for the need to incrementally refine the definition of quality functions, and reflects a typical iterative process of knowledge discovery. The outer loop involves the use of such quality knowledge; at its core is the original notion of *Quality Views*, i.e., processes defined as a composition of quality functions, which capture the subjective aspect of quality by including (i) a particular user choice of functions, and (ii) a decision process that is based on the function values.

As a result of our research, we have found that we can facilitate the creation and dissemination of quality knowledge, by defining (i) a conceptual model for

capturing user-defined information quality (IQ) concepts in an incremental way, and (ii) a software architecture, called the Qurator workbench, for the definition and execution of Quality Views, and for their integration as part of e-science experiments. In the previous chapter we have shown how the combination of these two elements can be used to provide support to various phases of the IQ lifecycle. In the following we summarize our main contributions in support of this claim.

Data classification framework and IQ ontology

Regarding the IQ model, in Chapter 2 we have advocated the use of a data classification framework for representing user-defined quality knowledge. Then, in Chapter 3 we have explored the use of ontologies to accommodate the symbolic definition of quality functions; this is aimed at facilitating the task of sharing and reusing quality functions among members of the community. We have also found that the logic features of the ontology make it possible to define a rich axiomatization of IQ concepts. As a result, we have been able to use the ontology to provide a formal definition of *consistency* of a Quality View, and we have given an algorithm for testing consistency.

Quality Views and quality workflows

Regarding the software architecture, in Chapter 4 we have proposed a simple language for the specification of Quality Views as abstract processes, we have defined a formal semantics for the language, and we have shown that these Quality View specifications can be automatically compiled into composite services, provided that their component quality functions are themselves implemented as services. To demonstrate this idea, we have chosen to compile Quality Views into workflows (called "quality workflows" in Chapter 5). Workflows have the advantage over other types of service, that they can be easily integrated with e-science experiments, which are often themselves specified as workflows. We have shown examples of how workflow experiments can be made quality-aware with little human effort.

In Section 6.2 we have also shown how we can use Quality Views to add quality-based data selection to query processing, namely to XQuery, exploiting the implementation of Quality Views as services.

Support for quality function implementation

The specification of Quality Views, their compilation and execution as quality processes are only three of the phases of the IQ lifecycle that we have demonstrated support for. In addition, in Chapter 6 (Section 6.1) we have also addressed the problem of how to turn quality knowledge into an implementation of quality functions: this is the last phase of the inner loop in the IQ lifecycle, where quality knowledge engineers release their quality functions to the community. Observing that these functions follow a regular design pattern, we have shown how the effort required to generate implementation code for those functions can be partly reduced through automated code generation; furthermore, we have made the point that the amount of automation that can be achieved depends on the users' reliance on particular software frameworks for the implementation of the function logic.

Definition and support for quality provenance

After the execution of a Quality View process, the last phase of the lifecycle outer loop involves analysing its results. To model this phase, in Chapter 6 (Section 6.3) we have proposed the concept of *quality provenance*, i.e., provenance of a quality process. Since quality processes make decisions as to whether data elements should be accepted or rejected, we have argued that those decisions should be explained and made clear to users after execution. Our quality provenance model demonstrates how users can track the accept/reject decision made during the execution of a Quality View, and across a history of executions.

7.2 Limitations and further research

We can improve upon and extend our results in several directions. In this section we discuss current limitations and propose further research ideas to overcome those limitations. For this, we are going to use our IQ lifecycle one last time as a reference framework (please see Figure 6.1 on page 160).

We begin with two complementary issues related to our main body of research, namely support for the outer loop, and then address more exploratory issues concerning the inner loop.

7.2.1 Managing uncertainty in quality

Probabilistic classification

The first issue concerns the type of classification computed by our quality functions. So far we have assumed that these functions associate either a class label or a score with data elements. This, however, may not be sufficient to capture the inherent uncertainty associated to any predictive model of quality. Indeed, some classifiers, like Naive Bayes, return the probability that a data element belongs to a certain class, in addition to the class label. We argue that maintaining an explicit representation of uncertainty in the quality characterization of the data can be useful, because it provides additional input to the decision models that are based on quality: if we interpret the probability of class membership as a measure of strengh of the classification, we can then use it for example to determine whether a given quality characterization is significant enough to influence a decision.

Data management support for uncertain data

The second issue concerns the long-lived association between the data and its quality, and the further uses that we can make of the quality information beyond the execution of a Quality View. For example, at the end of a protein identification workflow we may decide to save both the results, i.e., the list of identified proteins, and their score or quality class. To see how maintaining this information may be useful, let us recall an observation we made during the course of our survey on data quality research, in the first part of Chapter 2, regarding certain approaches to data integration in the presence of errors in the sources. In some of this work, notably by Naumann *et al.* [NUJ99, Nau02b], the authors postulate that integration can be driven by vectors of quality values that are associated to the data in the various sources, but they seem to underestimate the problem of how those vectors can be generated in the first place. A similar problem of maintaining an explicit association between data and quality metadata is also found in the DaQuincis architecture [SVM+04], as mentioned in the same survey (on page 36). We can see here how the values computed using quality views may contribute to solving the problem, by providing values for the vectors.

Further work is required, however, to design an adequate data engineering solution to store quality metadata. Our recent proposal for a generic metadata

management architecture [MAC+07] can be used as a starting point. One feature that makes it interesting is that it associates explicit *lifetime* information to metadata; using this feature, users may specify that certain events invalidate the metadata. This may be useful in the case of quality metadata, for example to indicate that a quality value should no longer be used when the underlying quality evidence changes.

By combining the two issues discussed here, namely explicit representation of uncertain quality metadata, and long-lived metadata values, we are faced with the new problem of a metadata repository with support for uncertainty. The recent research results mentioned in Chapter 2 (Section 2.2.5) on this topic, namely the Trio system [Wid05, BSHW06], the MayBMS system [AKO07b] and its related theoretical results [AKO07a, AKO07c] can be used as a starting point for research in this direction.

7.2.2 Problems in quality knowledge discovery

We now move on to issues related to the inner loop. As we have stated in our introductory chapter, in our work we have focused for the most part on the outer loop of the lifecycle, and have simply assumed that quality knowledge engineers can create new quality functions when needed. Our only current contribution to the inner loop consists, as mentioned above, of a scheme for the partial automation of Quality Assertion implementation code. This leaves open a number of interesting research problems regarding the creation of new quality knowledge.

Types of quality evidence

The first issue concerns the choice of metadata types that are good candidates to be used as quality evidence. In the Introduction, and again in Chapter 2, we have cited the availability of rich metadata as one of the main pre-requisites for providing a meaningful quality characterisation to complex data types, such as the results of a scientific experiment. Although we have cited data and process provenance as possible sources of evidence, we have yet to investigate this intuition in detail. In this realm, one could for example attempt to demonstrate the effectiveness of pre-defined score models, such as those provided natively by protein identification algorithms (which are different from the new score model that we have used in our example, proposed by Stead *et al.* [SPB06]), or some

combination thereof. Furthermore, anecdotal evidence may suggest, for instance, a possible correlation between the accuracy of an experiment description and the accuracy of its actual results, on the grounds that a precise and detailed description of the experiment is indicative of a reliable process control used by the experimenters. Also, one may devise measures of overall reputation of a laboratory, that could be used as indirect predictors of the accuracy of their results. More generally, we ask whether we can find interesting predictors of information quality in process provenance, as acquired during the execution of workflows, or in the rich description of e-science experiments as part of the submission of the results to public sites (such as in the case of MIAME and MIAPE).

As we can see, this is a broad and largely uncharted area where the search for valuable quality knowledge promises to be challenging. We propose to investigate ways of extending our Qurator workbench in order to support this search.

Cost of quality evidence

Assuming that we have demonstrated the predictive power of certain types of metadata for quality estimation, we are still faced with the potential problem that such metadata may not always be available, or that it may be expensive to acquire. The availability and cost of quality evidence largely determine the type of quality estimation that we can afford to associate with our experiments, and the overhead incurred in its computation. While in the use cases presented in the Introduction, the evidence necessary to compute the score models are promptly available (from the output of a workflow processor and from a database, respectively), this may not always be the case. Suppose for instance that we decide to use some details of an experimental process, such as the parameters used to configure some equipment, as evidence. Unless the collection of this information is mandatory according to the experiment description guidelines, it may well be missing, and it may be costly, or even impossible, to retrieve it from other sources. These considerations lead to several new research questions. Firstly, how is the performance of a quality decision model affected by partially missing input metadata? Current results in the area of knowledge discovery indicate that some algorithms are more robust than others in this situation (see for example [Li06, BM03]). In the same vein, under what circumstances can we use "cheaper" types of evidence as surrogates for other, more expensive ones? And finally, how can we define a cost model for quality estimation? Would a

formal characterization of the Annotation functions in terms of their cost provide a valid starting point?

Discovery and implementation of new quality knowledge

Having established what types of potential quality evidence are available, the next step within the inner lifecycle loop is to design an accurate predictive model. This problem, as we have suggested early in the thesis, falls in the realm of data mining and knowledge discovery (KD), a prolific area involving statistic and data management [HK06]. Since our current Qurator workbench does not include specific support for the discovery phase, a natural question to ask is to what extent we can exploit, and possibly improve upon, the body of theoretical and technological results available from the KD community. We plan to investigate how these results can be integrated into the workbench, in order to streamline the quality discovery process as much as possible.

We have described our preliminary investigation into this possibility while presenting our proposal for the semi-automated generation of quality service implementation code, in Section 6.1. As a concrete use case, we have studied ways to automatically turn Weka models [WF05] into Quality Assertion services. Our early results indicate that the semantic information regarding the QV components recorded in the Qurator ontology, combined with the semantic annotation of a generic quality service interface, are sufficient to automate the generation of an executable quality service that implements a Weka classifier. However, further work is needed on this topic to provide a general generation mechanism across a wider range of potential quality classification and score models.

Axiomatization of new quality knowledge

The last responsibility for quality knowledge engineers, after having defined and implemented a new quality function, is to provide a symbolic representation for it in the ontology. As discussed in Chapter 3, this involves creating new ontology classes that extend those defined in the upper ontology, and defining a number of logical axioms to establish relationships among these classes. The complete example that we presented in Table 3.3 (page 84) for the proteomics use case indicates that this may be a non-trivial task that requires specific expertise in the management of semantic models. Regarding this, in Chapter 3 we have already made two points. Firstly, that we can exploit automated reasoning on the

ontology to ensure that the result is not an inconsistent collection of axioms; we have described the potential and the limitations of this approach. And secondly, that the specification of the set of axioms may proceed incrementally: if any of the expected axioms are missing, this will typically limit the types of support provided by the reasoner, but will not result in inconsistencies. In future work we plan to explore this issue further, to see whether we can support the task of creating the axioms in any useful way. We propose to consider ontology-aware user interfaces, specifically looking at open platforms for ontology design, such as Protege (`protege.stanford.edu`), as a starting point to achieve this goal.

Appendix A

BNF grammar for Quality Views Action expressions

(Token definitions omitted)

<expression> : <term> [**OR** <expression>]
<term> : <fact> [**AND** <term>]
<fact> : [**NOT**] <primary>
<primary> : <relComparison> | <idTest> | "(" <expression> ")"
<idTest> : <setMembership> | <nullValueTest>
<relComparison> : (<ID>
| <INTEGER_LITERAL>
| <FLOATING_POINT_LITERAL>
| <STRING_LITERAL>)
("<" | "<=" | ">" | ">=" | "=" | "=" | "<>" | "~" "=")
(<ID>
| <INTEGER_LITERAL>
| <FLOATING_POINT_LITERAL>
| <STRING_LITERAL>
| <PATTERN>)
<nullValueTest> : <ID> **IS** [**NOT**] **NULL**
<setMembership> : <ID> [**NOT**] **IN** <setExpr>
<setExpr> : "{" <STRING_LITERAL> ["," <STRING_LITERAL>]* "}"

Appendix B

Haskell code for the Quality View interpreter

```
------------------------------
-- A QV interpreter in Haskell
------------------------------

------------------------------
--- some names for simple types
------------------------------
type URI = String   -- should really be a URI...
type Dataref = String
type ActualParamValue = String
type FormalParamName = String
type LocalFormalParam = (FormalParamName, FormalParamName)
type BoundActualParam = (FormalParamName , ActualParamValue)

------------------------------
--- Environment
------------------------------
data QTriple = QTriple   { _Name :: String, _Class :: URI, _Value :: String }
type QTripleSet = [ QTriple ]

-- one row in the env matrix
data EnvRow     = EnvRow { d :: URI, qSet :: [ QTriple] } deriving Show
type Env        = [ EnvRow ]  -- matrix == list of rows
```

```
---------------
------ data model for Annotators
---------------

-- Annotator Functions Af map a dataref to a list of QTriples
-- the second arg is a list of actual parameters
type Af = Dataref -> [BoundActualParam] -> [QTriple]

-- an Annotator specification includes an Annotation function
-- and a list of (variable, class) pairs
data AnnSpec    = AnnSpec { _Af :: Af,
                            _outputVars :: [String],
                            _AnnParameters :: [LocalFormalParam] }

---------------
------ data model for Quality Assertions
---------------
-- QA functions take a dataref and a list of QTriples and compute a new QTriple
type AnnotatedData  = (Dataref, [QTriple])
type QAf  = [AnnotatedData] -> [BoundActualParam] -> [(Dataref, [QTriple])]

-- a QA specification includes the QA function and a list of
--  (variable, class) pairs that indicate the evidence
--  that it depends on
data QASpec  = QA { _QAf :: QAf,
                    _inputVars :: [String],
                    _outputVar :: String,
                    _QAParameters :: [LocalFormalParam] }
```

```
------------------------------
--- Quality View specification
------------------------------
-- a Quality View is a list of Annotators, QA,
-- and QTest plus global formal parameters
data QVSpec  = QVSpec { _formalParams :: [FormalParamName],
                        _ann :: [AnnSpec],
                        _QA :: [QASpec],
                        _QT :: [QTest] }

-- QV output
type DatarefQualityClass = (Dataref, String, [QTriple])

---------------
------ data model for Actions
---------------
type CondExpr = Dataref -> Env -> Bool

-- cond: the conditional expression on the input vars
data QTest = QTest { channelName :: String,
                     channelAnnotation :: String,
                     cond :: CondExpr }

type ActionSpec = [QTest]
```

```
-- a QV is a function of input data and a QVSpec,
-- which returns a list of lists of datarefs,
-- each list representing one quality class
qv :: [Dataref] -> [ActualParamValue] -> QVSpec -> [DatarefQualityClass]
qv _D  _actualParams _QVSpec   =
           let env = initEnv _D (collectVars _QVSpec)
              _boundParams = paramBinding (_formalParams _QVSpec) _actualParams
            in  act _D (_QT _QVSpec)
                    (qAssert _D  (_QA _QVSpec) _boundParams
                        (annotate _D (map (\x -> ((_Af x),
                                    (globalToLocalParams
                                           _boundParams
                                          (_AnnParameters x))))
                                          (_ann _QVSpec))  env))


--------------
--- Annotations
--------------
-- apply Af to each dataref and update the Env with the annotation result
annotate1 :: [Dataref] -> (Af, [BoundActualParam]) -> Env -> Env
annotate1 _D (_Af, _parms) e = multiUpdateEnv [ (d, ( _Af d _parms )) |
                                                d <- _D ] e

-- [String] is a list of actual parameters
annotate :: [Dataref] -> [(Af, [BoundActualParam])] -> Env -> Env
annotate _ [] e  = e
annotate _D (h:rest) e = annotate _D rest (annotate1 _D h e)

-- collects all var names from each annotator
collectVars :: QVSpec -> [ String ]
collectVars _QVSpec   = concat [ (_outputVars _annSpec)   |
                                          _annSpec <- (_ann _QVSpec)] ++
                                [ (_outputVar _QASpec)   |
                                          _QASpec <- (_QA _QVSpec)]
```

```
--------------
--- Quality Assertions
----------------
-- one QAssertion takes in data and its annotations and applies a QAf,
-- updating the env
-- to reflect the new QTriple

qAssert1 :: [Dataref] -> QASpec -> [BoundActualParam] -> Env -> Env
qAssert1 _D _QASpec _actualParams e  = multiUpdateEnv
                                       ((_QAf _QASpec)
                                        [ (d,
                                          (fetchAnnotations
                                                  d
                                                  (_inputVars _QASpec) e)) |
                                                   d <- _D ]
                                          (globalToLocalParams
                                                  _actualParams
                                                  (_QAParameters _QASpec)))
                                          e


-- multiple QAssertions
qAssert :: [Dataref] -> [QASpec] -> [BoundActualParam] -> Env -> Env
qAssert _ [] _ e = e
qAssert _D (h:rest) _actualParams e = qAssert _D rest _actualParams
                                            (qAssert1 _D h _actualParams e)


--------------
--- Actions
--------------
-- perform action tests on the enriched data and associate
-- a channel name to each dataref
act :: [Dataref] -> [QTest] -> Env -> [DatarefQualityClass]
act _D []  e     = [ (d, "all data", (allQTriples d e)) | d <- _D ]
act _D _Tests e  = [ (d, (channelName aTest), (allQTriples d e)) |
                             d <- _D, aTest <- _Tests,
                             (cond aTest) d e  == True ]
```

```
--------------
--- Parameters binding
--------------
-- binds each global formal parameter name to an actual parameter value,
-- by position
paramBinding :: [FormalParamName] -> [ActualParamValue]-> [BoundActualParam]
paramBinding (x:xs) (y:ys) = (x,y) : paramBinding xs ys
paramBinding [] [] = []

-- maps global actual parameters of the form (globalName, value)
-- to local formal parameters of the form (globalName, localName)
-- to yield (localName, value)
globalToLocalParams :: [BoundActualParam] ->
                               [LocalFormalParam] ->
                                       [BoundActualParam]
globalToLocalParams g l = [ (localName, value) |
                            (globalName, value) <- g,
                            (globalName1, localName) <- l,
                            globalName == globalName1 ]

--------------
--- Environment
--------------
-- adds a row to env to account for one data identifier
addRow :: URI -> Env -> Env
addRow x []  = [EnvRow {d = x, qSet = [] }]
addRow x (r:rest) = EnvRow {d = x, qSet = [] } : r : rest

addRows :: [URI] -> Env -> Env
addRows [] e = e
addRows (d:rest) e = addRows rest (addRow d e)

-- add one evidence tuple to one row of the env with the varname but no value
addQTriple :: String -> QTripleSet -> QTripleSet
addQTriple    v qSet = QTriple { _Name = v, _Class = "", _Value = ""} : qSet

-- add one evidence tuple to each env row
```

```
addQVar :: String -> Env -> Env
addQVar v e =  [ EnvRow {d = (d row), qSet = (addQTriple v (qSet row)) } |
                                                    row <- e ]
-- add one evidence tuple to each env row, for each evidence variable name

addQVars :: [String] -> Env -> Env
addQVars [] e = e
addQVars (v:rest) e = addQVars rest (addQVar v e)

-- update each EnvRow identified by a d in Dataref
-- with the corresponding values in the QTriple
multiUpdateEnv :: [(Dataref, [QTriple])] -> Env -> Env
multiUpdateEnv [] e = e
multiUpdateEnv ((d, qTriples) :rest) e = multiUpdateEnv rest
                                            (updateEnv1 d qTriples e)

-- updates the row identified by d with all Qtriples in the second arg
updateEnv1 :: String -> [QTriple] -> Env -> Env
updateEnv1 d (h:rest) e = updateEnv1 d rest (updateEnv d h e)
updateEnv1 _ [] e = e

---------
--- updating the env with a new value
--- eg (updateEnv dataref varname value e)
--- ex updateEnv d QTriple {"ev", "evClass", "value"}
updateEnv :: String -> QTriple -> Env -> Env
updateEnv _ _ [] = []
updateEnv dataref q (h:rest) |
                  (d h) == dataref  =
                            EnvRow {d = (d h),
                            qSet = (updateCellSet q (qSet h)) } : rest

updateEnv dataref q (h:rest) |
                  (d h) /= dataref  = h : (updateEnv dataref q rest)

updateCellSet :: QTriple ->  [QTriple] -> [QTriple]
updateCellSet _ [] = []
updateCellSet q (cell:rest) |
```

```
                  (_Name cell) == (_Name q)  =
                           QTriple {_Name = (_Name cell),
                                    _Class = (_Class q),
                                    _Value = (_Value q)} : rest

updateCellSet q (cell:rest) | (_Name cell) /= (_Name q) =
                           cell : (updateCellSet q rest)

-- fetch the QTriples for a list of variables v and for a specific dataref
fetchAnnotations :: Dataref -> [String] -> Env -> [QTriple]
fetchAnnotations d _V e = concat [ (getQTriple d v e) | v <- _V ]

-- fetch entire row of annotations + quality values for d
allQTriples :: Dataref -> Env ->  [ QTriple ]
allQTriples dataref e = concat [ (qSet row) | row <- e, (d row) == dataref ]


-- [QTriple] is a singleton. an empty QTriple list indicates
-- no QTriple matching the (data, var) pair
-- getQTriple d v e = [_Name = "v", _Class = "c", _Value = value"]
getQTriple :: String -> String -> Env -> [QTriple]
getQTriple _ _  [] = []
getQTriple dataref var (h:rest) | (d h) == dataref   =
                                [ qtriple  |
                                     qtriple <- (qSet h) ,
                                     (_Name qtriple) == var]
getQTriple dataref var (h:rest) | (d h) /= dataref = getQTriple dataref var rest


--- env initialization
-- takes list of dataref and a list of variable names and creates the env matrix
--- by the way the env is setup, rows must be added first
initEnv ::  [String] -> [String] -> Env
initEnv  datarefs vars   =  addQVars vars (addRows datarefs [])
```

```
---------------------
---------------------
-- example QVSpec
---------------------
---------------------

-- annotations
af1  = \x -> \plist -> [QTriple { _Name = "e1",
                                 _Class = "e1Class",
                                 _Value = (x ++ " e1 annot")},
                        QTriple { _Name = "e2",
                                 _Class = "e2Class",
                                 _Value = (x ++ " e2 annot")} ]

af2  = \x -> \plist -> [QTriple { _Name = "e3",
                                 _Class = "e3Class",
                                 _Value = (x ++ " e3 annot")}]

ann1 = AnnSpec { _Af = af1,
                 _outputVars = ["e1", "e2"],
                 _AnnParameters = [("p1","p1Local")] }

ann2 = AnnSpec { _Af = af2,
                 _outputVars = ["e3"],
                 _AnnParameters = [("p1","p1"), ("p2","p2Local")] }

-- QA
qa1  = \ad -> \plist -> [ (d, [QTriple { _Name = "q1",
                                         _Class = "q1Class",
                                         _Value = "QA1 for "++" d " }])  |
                          (d, _) <- ad ]

qa2  = \ad -> \plist -> [ (d, [QTriple { _Name = "q2",
                                         _Class = "q2Class",
                                         _Value = "QA2 for "++" d " }])  |
                          (d, _) <- ad ]
```

```
_QASpec1 = QA { _QAf = qa1,
                _inputVars = ["e1", "e2"],
                _outputVar = "q1",
                _QAParameters = [("p3","p3")] }
_QASpec2 = QA { _QAf = qa2,
                _inputVars = ["e2", "e3"],
                _outputVar = "q2",
                _QAParameters = [("p2","p2Local")] }

-- QTest (action)
_channel1Condition = \d -> \e -> True
_Qtest1 = QTest { channelName = "ch1",
                  channelAnnotation = "red",
                  cond = _channel1Condition}

_channel2Condition = \d ->
                     \e ->
                     (_Value (head (getQTriple d "q2" e)) == "QA2 for "++" d ")

_Qtest2 = QTest { channelName = "ch2",
                  channelAnnotation = "white",
                  cond = _channel2Condition}

-- Quality View
testQV = QVSpec { _formalParams = ["p1", "p2", "p3"],
                  _ann = [ann1, ann2],
                  _QA = [_QASpec1, _QASpec2],
                  _QT = [_Qtest1, _Qtest2] }
```

Bibliography

[ABBMed] A. Avenali, P. Bertolazzi, C. Batini, and P. Missier. Brokering infrastructure for minimum cost data procurement based on quality - quantity models. *Decision Support Systems*, 2007, Accepted.

[ABC99] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.

[ABC+03] M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar aggregation in inconsistent databases. *Theoretical Computer Science*, 296(3):405–434, March 2003.

[Abr90] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.

[AKO07a] L. Antova, C. Koch, and D. Olteanu. From complete to incomplete information and back. In *SIGMOD Conference*, pages 713–724, 2007.

[AKO07b] L. Antova, C. Koch, and D. Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, pages 1479–1480, 2007.

[AKO07c] L. Antova, C. Koch, and D. Olteanu. World-set decompositions: Expressiveness and efficient algorithms. In *ICDT*, pages 194–208, 2007.

[AM03] R. Aebersold and M. Mann. Mass spectrometry-based proteomics. *Nature*, 422:198–207, March 2003.

[AMMH07] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[APS03] T.K. Attwood and D.J. Parry-Smith. *Introduction to Bioinformatics*. Pearson Education Taiwan Ltd, 2003.

[BBD07] O. Biton, S. Cohen Boulakia, and S. B. Davidson. Zoom*userviews: Querying relevant provenance in workflow systems. In *VLDB*, pages 1366–1369, 2007.

[BBS05] M. Bilenko, S. Basu, and M. Sahami. Adaptive product normalization: Using online learning for record linkage in comparison shopping. In *ICDM*, pages 58–65, 2005.

[BBWG03] M. Buechi, A. Borthwick, A. Winkel, and A. Goldberg. ClueMaker: A language for approximate record matching. In *Procs. 8th International Conference on Information Quality, ICIQ 2003, Cambridge, Ma*, 2003.

[BC07] C. Baker and H. Cheung, editors. *Semantic Web – Revolutionizing Knowledge Discovery in the Life Sciences*, chapter Knowledge Discovery for Biology with Taverna. Biomedical and Life Sciences. Springer, 2007.

[BCM+03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[BDM02] S. Bocs, A. Danchin, and C. Medigue. Re-annotation of genome microbial coding-sequences: finding new genes and inaccurately annotated genes. *BMC Bioinformatics*, 2002.

[BE04] L. Berti-Equille. Quality-Adaptive Query Processing over Distributed Sources. In *Proceedings of the 9th International Conference on Information Quality (IQ'04)*, pages 285–296, Boston MA, USA, 2004.

[BEBS05] L. Berti-Equille, C. Batini, and D. Srivastava, editors. *IQIS 2005, International Workshop on Information Quality in Information Systems, 17 June 2005, Baltimore, Maryland, USA (SIGMOD 2005 Workshop)*. ACM, 2005.

[BEF+05] K. Belhajjame, S.M. Embury, H. Fan, C. Goble, and al. Proteome data integration: Characteristics and challenges. In *Proceedings of UK e-ScienceAll Hands Meeting*, 2005.

[BEPG+05] L.D. Burgoon, J.E. Eckel-Passow, C. Gennings, D.R. Boverhof, J.W. Burt, C.J. Fong, and T.R.Zacharewski. Protocols for the assurance of microarray data quality and process control. *Nucleic Acids Research*, 33(19), 2005.

[BFG+07] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.

[BG05] J. Barateiro and H. Galhardas. A survey of data quality tools. *Datenbank-Spektrum*, 14:15–21, 2005.

[BGMY04] M. Bern, D. Goldberg, W.H. McDonald, and J.R. III Yates. Automatic quality assessment of peptide tandem mass spectra. *Bioinformatics*, 20(Suppl. 1):i49–i54, 2004.

[BHS05a] F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg Siekmann on the Occasion of His 60th Birthday*, number 2605 in Lecture Notes in Artificial Intelligence, pages 228–248. Springer, 2005.

[BHS05b] L. E. Bertossi, A. Hunter, and T. Schaub, editors. *Inconsistency Tolerance [result from a Dagstuhl seminar]*, volume 3300 of *Lecture Notes in Computer Science*. Springer, 2005.

[BKM06] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM*, pages 87–96, 2006.

[BM03] G. Batista and M.C. Monard. An analysis of four missing data treatment methods for supervised learning. *Applied Artificial Intelligence*, 17(5-6):519–533, 2003.

[Bre99] S. Brenner. Errors in genome annotation. *Trends in Genetics*, 15(4):132–133, 1999. Short Communication.

[BS06] C. Batini and M. Scannapieco. *Data Quality – Concepts, Methodologies and Techniques*, volume XX of *Data-Centric Systems and Applications*. Springer, 2006.

[BSHW06] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: databases with uncertainty and lineage. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 953–964. VLDB Endowment, 2006.

[BT07] S. Cohen Boulakia and V. Tannen, editors. *Data Integration in the Life Sciences, 4th International Workshop, DILS 2007, Philadelphia, PA,*

USA, June 27-29, 2007, Proceedings, volume 4544 of *Lecture Notes in Computer Science*. Springer, 2007.

[CAB+04] S. Carr, R. Aebersold, M. Baldwin, A. Burlingame, et al. The need for guidelines in publication of peptide and protein identification data. *Molecular & Cellular Proteomics*, 3(6):531–533, 2004.

[CBD06] S. Cohen, S. Cohen Boulakia, and S. B. Davidson. Towards a model of provenance and user views in scientific workflows. In Leser et al. [LNE06], pages 264–279.

[CCG+00] F. Caruso, M. Cochinwala, U. Ganapathy, G. Lalk, and P. Missier. Demonstration of telcordia's database reconciliation and data quality analysis tool. In *VLDB 2000, September 10-14, 2000, Cairo, Egypt*, pages 615–618. Morgan Kaufmann, 2000.

[CCGK07] S. Chaudhuri, B. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *Proceedings VLDB*, Vienna, Austria, Sept. 2007.

[CFG+07] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *Procs. VLDB*, Vienna, Austria, Sept. 2007.

[CGH+06] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming Scientific and Distributed Workflow with Triana Services. *Concurrency and Computation: Practice and Experience (Special Issue: Workflow in Grid Systems)*, 18(10):1021–1037, 2006.

[CMM03] J. Colinge, A. Masselot, and J. Magnin. A systematic analysis of ion trap tandem mass spectra in view of peptide scoring. In *Proceedings of the Third International Workshop on Algorithms in Bioinformatics (WADI)*, Lecture Notes in Computer Science, Budapest, September 2003.

[CMRSW06] R. Chinnici, J-J. Moreau, A. Ryman, and eds. S. Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. World Wide Web Consortium, March 2006. available at http://www.w3.org/TR/wsdl20.

[CPH+03] M. Cornell, N. Paton, C. Hedeler, P. Kirby, D. Delneri, A. Hayes, and S. Oliver. GIMS: an integrated data storage and analysis environment for genomic and functional data. *Yeast*, 20(15):1291–306, Nov 2003.

[CZS+04] K.K. Challapalli, C. Zabel, J. Schuchhardt, A.M. Kaindl, et al. High reproducibility of large-gel two-dimensional electrophoresis. *Electrophoresis*, 25:3040–3047, 2004.

[DS04] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875, 2004.

[DS07] N. N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *PODS*, pages 1–12, 2007.

[DV01] D. Devos and A. Valencia. Intrinsic errors in genome annotation. *TRENDS in Genetics*, 17(8), 2001.

[EBC+05] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (bpel). *Journal of Grid Computing*, 3(3-4):283–304, 2005.

[EEV02] M.G. Elfeky, A.K. Elmagarmid, and V.S. Verykios. Tailor: a record linkage tool box. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, San Jose, CA, Feb. 2002. IEEE Computer Society.

[EIV07] A. K. Elmagarmid, P. G. Ipeirotis, and V.S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, Jan 2007.

[EMS+07] S. Embury, P. Missier, S. Sampaio, M. Greenwood, and A. Preece. Incorporating domain-specific information quality constraints into database queries. Journal of Data and Information Quality, submitted, Sept. 2007.

[Faw06] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 28(8):861–874, 2006.

[FB02] D. Fenyö and R.C. Beavis. Informatics and data management in proteomics. *Trends in Biotechnology*, 20(12 (Suppl.)):S35–S38, 2002.

[FGMA02] M.R. Flory, T.J. Griffin, D. Martin, and R. Aebersold. Advances in quantitative proteomics using stable isotope tags. *Trends in Biotechnology*, 20(12 (Suppl.)):S23–S29, 2002.

[FS69] I.P. Fellegi and A.B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64, 1969.

[FvHH+01a] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P. F. Patel-Schneider. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[FvHH+01b] D. Fensel, F. van Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45, Mar/Apr 2001.

[GFS+01] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.A. Saita. Declarative data cleaning: Language, model, and algorithms. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 371–380. Morgan Kaufmann, 2001.

[GGS+03] M. Greenwood, C. Goble, R. Stevens, J. Zhao, M. Addis, D. Marvin, L. Moreau, and T. Oinn. Provenance of e-science experiments - experience from bioinformatics. In Simon Cox, editor, *OST e-Science Second All Hands Meeting 2003 (AHM'03)*, Nottingham, UK, September 2003.

[GPFLC04] A. Gómez-Pérez, M. Fernández-López, and O. Corcho. *Ontological Engineering with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web*. Advanced Information and Knowledge Processing. Springer, 2004. ISBN: 978-1-85233-551-9.

[HHJW07] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, San Diego, California, June 2007.

[HK06] J. Han and M. Kamber. *Data Mining –Concepts and Techniques*. Morgan Kauffman, 2nd edition, 2006. ISBN 10:1-55860-901-6.

[HM07] C. Hedeler and P. Missier. *Database Modeling in Biology: Practices and Challenges*, chapter Quality management challenges in the post-genomic era. Artech House, 2007. In print.

[Hor02] Ian Horrocks. DAML+OIL: a description logic for the semantic web. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1):4–9, March 2002.

[HPSvH03] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26, 2003.

[Hul06] D. Hull. Description and classification of shims in ^{my}grid. Technical report, University of Manchester, 2006.

[HWS+06] D. Hull, K. Wolstencroft, R. Stevens, C. A. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web-Server-Issue):729–732, 2006.

[HWSG02] W.S. Hancock, S.L. Wu, R.R. Stanley, and E.A. Gombocz. Publishing large proteome datasets: scientific policy meets emerging technologies. *Trends in Biotechnology*, 20(12 (Suppl.)):S39–S44, 2002.

[HZB+] D. Hull, E. Zolin, A. Bovykin, I. Horrocks, U. Sattler, and R. Stevens. Deciding semantic matching of stateless services. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06) and Eighteenth Innovative Applications of Artificial Intelligence (IAAI-06) Conference*, pages 1319–1324, Boston, MA, USA, July.

[IGH01] T. Ideker, T. Galitski, and L. Hood. A new approach to decoding life: Systems biology. *Annu. Rev. Genomics Hum. Genet.*, 2:343–372, 2001.

[JHH+93] S.L. Peyton Jones, C.V. Hall, K. Hammond, W. D. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, March 1993.

[JO04] H. V. Jagadish and F. Olken. Database management for life sciences research. *SIGMOD Record*, 33(2):15–20, 2004.

[KMGa04] K. Garwood K, T. McLaughlin, C. Garwood, and al. PEDRo: a database for storing, searching and disseminating experimental proteomics data. *BMC Genomics*, 5(1), Sep 2004.

[KVVS+06] T. Kifor, L. Varga, J. Vàzquez-Salceda, S.Sergio lvarez, and S. Willmott. EHCR: An EU provenance case study. Technical report, SZTAKI, 2006.

[LA03] B. Ludscher and I. Altintas. On providing declarative design and programming constructs for scientific workflows based on process networks. Technical Report SciDAC-SPA-TN-2003-01, San Diego Supercomputer Center, 2003.

[LABe05] B. Ludscher, I. Altintas, C. Berkley, and el. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, Special Issue on Scientific Workflows, 2005.

[LE05] J. Listgarten and A. Emili. Statistical and computational methods for comparative proteomic profiling using liquid chromatography-tandem mass spectrometry. *Molecular & Cellular Proteomics*, 4(4):419–434, 2005.

[LH03] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 331–339, New York, NY, USA, 2003. ACM Press.

[Li06] J. Li. Robust rule-based prediction. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1043–1054, 2006.

[LNE06] U. Leser, F. Naumann, and B. A. Eckman, editors. *Data Integration in the Life Sciences, Third International Workshop, DILS 2006, Hinxton, UK, July 20-22, 2006, Proceedings*, volume 4075 of *Lecture Notes in Computer Science*. Springer, 2006.

[LR05] B. Ludäscher and L. Raschid, editors. *Data Integration in the Life Sciences, Second InternationalWorkshop, DILS 2005, San Diego, CA, USA, July 20-22, 2005, Proceedings*, volume 3615 of *Lecture Notes in Computer Science*. Springer, 2005.

[LSBG03] P.W. Lord, R.D. Stevens, A. Brass, and C.A. Goble. Investigating semantic similarity measures across the Gene Ontology: the relationship between sequence and annotation. *Bioinformatics*, 19(10):1275–83, 2003.

[LZRA03] X. Li, H. Zhang, J.A. Ranish, and R. Aebersold. Automated statistical analysis of protein abundance ratios from data generatted by stable-isotope dilution and tandem mass spectrometry. *Analytical Chemistry*, 75(23):6648–6657, 2003.

[MAA04] A. Motro, P. Anokhin, and A.C. Acar. Utility-based resolution of data inconsistencies. In Felix Naumann and Monica Scannapieco, editors, *International Workshop on Information Quality in Information Systems 2004 (IQIS'04)*, Paris, France, June 2004. ACM.

[MAC+07] P. Missier, P. Alper, O. Corcho, I. Dunlop, and C. Goble. Requirements and services for metadata management. *IEEE internet Computing*, Special issue on Semantic-Based Knowledge Management, Sept. / Oct. 2007.

[MBL06] T. McPhillips, S. Bowers, and B. Ludscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *Proceedings*

3rd International Conference on Data Integration for the Life Sciences (DILS), LNCS/LNBI. Springer, 2006.

[MDBL07] D. Martin, J. Domingue, M. L. Brodie, and F. Leymann. Semantic web services, part 1. *IEEE Intelligent Systems*, 22(5):12–17, 2007.

[ME05] P. Missier and S. M. Embury. Provider issues in quality-constrained data provisioning. In *IQIS 2005, International Workshop on Information Quality in Information Systems, 17 June 2005, Baltimore, Maryland, USA (SIGMOD 2005 Workshop)*, pages 5–15, 2005.

[MEG+06] P. Missier, S. M. Embury, M. Greenwood, A. D. Preece, and B. Jin. Quality views: Capturing and exploiting the user perspective on data quality. In *VLDB*, pages 977–988, Seoul, Korea, September 2006.

[MEG+07] P. Missier, S. M. Embury, M. Greenwood, A. D. Preece, and B. Jin. Managing information quality in e-science: the qurator workbench. In *SIGMOD Conference*, pages 1150–1152, 2007.

[MEH+07] P. Missier, S. Embury, C. Hedeler, M. Greenwood, J. Pennock, and A. Brass. Accellerating disease gene identification through integrated SNP data analysis. In *Procs. Data Integration in the Life Sciences 2007 (DILS 2007), June 2007, Philadelphia, USA*, LNBI. Springer, 2007.

[MGM+07] S. Miles, P. Groth, S. Munroe, S. Jiang, T. Assandri, and L. Moreau. Extracting causal graphs from an open provenance data model. *Concurrency and Computation: Practice and Experience*, 2007.

[MH05] A. Martinez and J. Hammer. Making quality count in biological data sources. In Berti-Equille et al. [BEBS05], pages 16–27.

[Mit97] T. M. Mitchell. *Machine Learning.* Mc Graw-Hill, 1997. ISBN 0-07-115467-1.

[MLV+03] P. Missier, G. Lalk, V. S. Verykios, F. Grillo, T. Lorusso, and P. Angeletti. Improving data quality in practice: A case study in the italian public administration. *Distributed and Parallel Databases*, 13(2):135–160, 2003.

[Mog91] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Mor90] J. M. Morrissey. Imprecise information and uncertainty in information systems. *ACM Trans. Inf. Syst.*, 8(2):159–180, 1990.

[MPE+05] P. Missier, A. D. Preece, S. M. Embury, B. Jin, M. Greenwood, D. Stead, and A. Brown. Managing information quality in e-science: A case study in proteomics. In *ER (Workshops)*, pages 423–432, 2005.

[MRE00] G. A. Mihaila, L. Raschid, and M. E.Vidal. Using quality of data metadata for source selection and ranking. In *WebDB (Informal Proceedings)*, pages 93–98, 2000.

[MSC04] D. Milano, M. Scannapieco, and T. Catarci. Quality-driven query processing of Xquery queries. In *CAiSE Workshops (2)*, pages 78–89, 2004.

[NA04] A.I. Nesvizhskii and R. Aebersold. Analysis, statistical validation and dissemination of large-scale proteomics datasets generated by tandem MS. *Drug Discovery Today*, 9(4):173–181, 2004.

[Nau75] J. Naus. *Data quality control and editing.* M. Dekker, 1975.

[Nau02a] F. Naumann. *Quality-Driven Query Answering for Integrated Information Systems*, volume 2261. Springer Berlin / Heidelberg, 2002. ISSN 0302-9743 (Print) 1611-3349 (Online).

[Nau02b] F. Naumann. *Quality-Driven Query Planning*, volume 2261 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.

[New67] H.B. Newcombe. Record linking: The design of efficient systems for linking records into individual and family histories. *Am. J. Human Genetics*, 19(3), 1967.

[NFL04] F. Naumann, J.C. Freytag, and U. Leser. Completeness of integrated information sources. *Information Systems*, 29(7):583–615, 2004.

[NKAJ59] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130:954–959, October 1959.

[NKKA03] A.I. Nesvizhskii, A. Keller, E. Koller, and R. Aebersold. A statistical model for identifying proteins by tandem mass spectrometry. *Analytical Chemistry*, 75(17):4646–4658, 2003.

[NUJ99] F. Naumann, U.Leser, and J.C.Freytag. Quality-driven integration of heterogenous information systems. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 447–458, Edinburgh, Scotland, UK, September 1999. Morgan Kaufmann.

[OAF+04] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, pages 3045 – 3054, November 2004.

[PJM+06] A. Preece, B. Jin, P. Missier, S. Embury, D. Stead, and Al Brown. Towards the management of information quality in proteomics. In *19th IEEE International Simposium on computer-based medical systems (IEEE CBMS 2006)*, Utah, USA, 2006.

[PJP+06] A. D. Preece, B. Jin, E. Pignotti, P. Missier, S. M. Embury, D. Stead, and A. Brown. Managing information quality in e-science using semantic web technology. In York Sure and John Domingue, editors, *ESWC*, volume 4011 of *Lecture Notes in Computer Science*, pages 472–486. Springer, 2006.

[PKPS02] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of International Semantic Web Conference (ISWC)*, Sardinia, Italy, 2002.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[PME+06] A. Preece, P. Missier, S. Embury, B. Jin, and M. Greenwood. An ontology-based approach to handling information quality in e-science. *Concurrency and Computation: Practice and Experience*, 2006. submitted.

[PPCC99] D.N. Perkins, D.J.C. Pappin, D.M. Creasy, and J.S. Cottrell. Probability-based protein identification by searching sequence databases using mass spectrometry data. *Electrophoresis*, 20:3551–3567, 1999.

[Qui93] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[RA05] K.A. Resing and N.G. Ahn. Proteomics strategies for protein identification. *FEBS Letters*, 579:885–889, 2005.

[Rah04] E. Rahm, editor. *Data Integration in the Life Sciences, First International Workshop, DILS 2004, Leipzig, Germany, March 25-26, 2004, Proceedings*, volume 2994 of *Lecture Notes in Computer Science*. Springer, 2004.

[RDH+04] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. OWL pizzas: Practical experience of teaching OWL-DL: Common errors and common patterns. In E Motta and et al N Shadbolt, editors, *Proceedings of the European Conference on Knowledge Acquistion*, volume LNAI3257 of *Lecture Notes on Computer Science*, pages 63–81, Northampton, England, 2004. Springer-Verlag.

[Red96] T.C. Redman. *Data quality for the information age*. Artech House, 1996.

[RH01] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[RS05] V. Ravichandran and R.D. Sriram. Toward data standards for proteomics. *Nature Biotechnology*, 23(3):373–376, 2005.

[SB04] M. Scannapieco and C. Batini. Completeness in the relational model: a comprehensive framework. In *Procs. 9th International Conference on Information Quality, ICIQ 2004, Cambridge, Ma*, 2004.

[SBB+00] R. Stevens, P.G. Baker, S. Bechhofer, G. Ng, A. Jacoby, N. W. Paton, C. A. Goble, and A. Brass. TAMBIS: Transparent access to multiple bioinformatics information sources. *Bioinformatics*, 16(2):184–186, 2000.

[SCY04] R.G. Sadygov, D. Cociorva, and J.R. III Yates. Large-scale database searching using tandem mass spectra: Looking up the answers in the back of the book. *Nature Methods*, 1(3):195–201, 2004.

[She07] A. Sheth. SAWSDL: Tools and applications. W3C track of WWW2007 Conference, May 2007. Banff, Canada.

[SMB05] M. Scannapieco, P. Missier, and C. Batini. Data quality at a glance. *Datenbank-Spektrum*, 14:6–14, 2005.

[Smi02] R.D. Smith. Trends in mass spectrometry instrumentation for proteomics. *Trends in Biotechnology*, 20(12 (Suppl.)):S3–S7, 2002.

[SPB06] D. A. Stead, A. Preece, and A. J.P. Brown. Universal metrics for quality assessment of protein identifications by mass spectrometry. *Molecular & Cellular Proteomics*, 5(7):1205–1211, 2006. Also available at http://www.mcponline.org/papbyrecent.shtml.

[SPG05] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[SVM+04] M. Scannapieco, A. Virgillito, C. Marchetti, M. Mecella, and R. Baldoni. The DaQuincis architecture: a platform for exchanging and improving data quality in cooperative information systems. *Inf. Syst.*, 29(7):551–582, 2004.

[SZB+04] I.I. Stewart, L. Zhao, T. Le Bihan, B. Larsen, et al. The reproducible acquisition of comparative liquid chromatography/tandem mass spectrometry data from complex biological samples. *Rapid Communications in Mass Spectrometry*, 18:1697–1710, 2004.

[TMR+07] D. Turi, P. Missier, D. De Roure, C. Goble, and T. Oinn. Taverna Workflows: Syntax and Semantics. In *Proceedings of the 3rd e-Science conference*, Bangalore, India, December 2007.

[TPG+03] C.F. Taylor, N.W. Paton, K.L. Garwood, P.D. Kirby, et al. A systematic approach to modeling, capturing, and disseminating proteomics experimental data. *Nature Biotechnology*, 21, 2003.

[TSWH07] I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana Workflow Environment: Architecture and Applications. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 320–339. Springer, New York, Secaucus, NJ, USA, 2007.

[TSWR03] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana applications within grid computing and peer to peer environments. *Journal of Grid Computing*, 1(2), June 2003.

[Var96] H. R. Varian. *Intermediate microeconomics : a modern approach*. W.W. Norton, New York ; London, 4 edition, 1996.

[VM04] V.S. Verykios and G.V. Moustakides. A generalized cost optimal decision model for record matching. In *Proc. 2004 Intl Workshop Information Quality in Information Systems (IQIS)*, 2004.

[VS07a] K. Verma and A. Sheth. Semantically annotating a web service. *IEEE Internet Computing*, 11(2), MarchApril 2007.

[VS07b] K. Verma and A. Sheth. Using SAWSDL for semantic service interoperability. Tutorial at Semantic Technology Conference, May 2007. San Jose, CA.

[Wad90] P. Wadler. Comprehending monads. In *LISP and Functional Programming*, pages 61–78, 1990.

[Wad95] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.

[WAH+07] K. Wolstencroft, P. Alper, D. Hull, C. Wroe, P. Lord, R. Stevens, and C. Goble. The ^{my}Grid ontology: Bioinformatics service discovery. *International Journal of Bioinformatics Research and Applications (IJBRA)*, 2007.

[WF05] I.H. Witten and E. Frank. *Data Mining – Practical Machine Learning Tools and Techniques*. Data Management. Morgan Kauffman, 2nd edition, 2005. ISBN-10: 0-12-088407-0.

[Wid05] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[Win93] W.E. Winkler. Improved decision rules in the felligi-sunter model of record linkage. Statistical Research Report Series RR93/12, US Bureau of the Census, Washington, D.C., 1993.

[Win02] W. E. Winkler. Methods for record linkage and bayesian networks. Technical report, U.S. Census Bureau, Statistical Research Division, 2002.

[Win06] W.E. Winkler. Overview of record linkage and current research directions. Statistical Research Report Series RRS2006/02, US Bureau of the Census, Washington, D.C., 2006.

[WKA04] D. Wieser, E. Kretschmann, and R. Apweiler. Filtering erroneous protein annotation. *Bioinformatics*, 20(Suppl. 1):i342–i347, 2004.

[WMF+05] S. C. Wong, S. Miles, W. Fang, P. T. Groth, and L. Moreau. Provenance-based validation of e-science experiments. In *International Semantic Web Conference*, pages 801–815, 2005.

[WPCea06] P.L. Whetzel, H. Parkinson, F.C. Causton, and L. Fan et al. The MGED Ontology: a resource for semantics-based description of microarray experiments. *Bioinformatics*, Jan 2006. Pubmed PMID: 16428806.

[WSG+03a] C. Wroe, R. Stevens, C. Goble, A. Roberts, and M. Greenwood. A suite of DAML+OIL Ontologies to describe bioinformatics web services and data. *International Journal of Cooperative Information Systems special issue on Bioinformatics*, March 2003. ISSN: 0218-8430.

[WSG+03b] C. Wroe, R. Stevens, C. Goble, A. Roberts, and M. Greenwood. A suite of DAML+OIL ontologies to describe bioinformatics web services and data. *International Journal of Cooperative Information Systems, special issue on Bioinformatics*, March 2003.

[ZAS02] N. Zhang, R. Aebersold, and B. Schwikowski. Probid: A probabilistic algorithm to identify peptides through sequence database searching using tandem mass spectral data. *Proteomics*, 2:1406–1412, 2002.

[ZWG+04] J. Zhao, C. Wroe, C. Goble, R. Stevens, D. Quan, and M. Greenwood. Using semantic web technologies for representing e-science provenance. In *Third International Semantic Web Conference (ISWC2004)*, number 3298 in LNCS, pages 92–106, Hiroshima, Japan, November 2004. Springer-Verlag.