HyperFlow: a Visual, Ontology-Based
Query and Data-Flow Language
for End-User Information Analysis

Dolev Dotan
HyperFlow: a Visual, Ontology-Based Query and Data-Flow Language for End-User Information Analysis

RESEARCH THESIS

Submitted In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Dolev Dotan

Submitted to the Senate of the Technion - Israel Institute of Technology

ADAR, 5766 HAIFA March 2006
This research thesis was done under the supervision of Assoc. Prof. Ron Pinter in the Department of Computer Science.

I wish to thank Assoc. Prof Ron Pinter for his counsel and guidance, and especially for his backing me up on all the various endeavors I took during this research. Without this atmosphere of extensive academic freedom this work would not have been possible.

I would like to thank my parents Yosi and Hana Dotan for their advice and support.

A special thanks to my wife Revital – for her love and encouragement as well as for her help in graphic design.

The generous financial help of the Technion and the IBM company is gratefully acknowledged.
1. Introduction ................................................................................................................3
   1.1 Motivation ..............................................................................................................3
   1.2 Contribution .........................................................................................................4
   1.3 Outline ..................................................................................................................5

2. Case Study: In-Silico Experiments and the Bioinformatics Assay Environment ....7
   2.1 Introduction ...........................................................................................................7
   2.2 Background: Current Technologies for Enabling In-Silico Experiments .................8
      2.2.1 The Centralized Approach ...........................................................................8
      2.2.2 The Distributed Approach ...........................................................................9
   2.3 Using HyperFlow to Define an In-Silico Experiment ............................................13
   2.4 Towards the Bioinformatics Assay Environment ................................................15

3. The Modeling Language ..............................................................................................16
   3.1 Background: OWL – The Semantic Web's Modeling Language .........................16
      3.1.1 Introduction ..................................................................................................16
      3.1.2 Universal Resource Identification and its Implications ..............................17
      3.1.3 OWL Subspecies .......................................................................................18
      3.1.4 OWL-DL Syntax .......................................................................................19
      3.1.5 OWL DL Semantics and Reasoning ..........................................................23
   3.2 The Semantics of using OWL in HyperFlow .......................................................24
   3.3 Custom Datatypes ...............................................................................................25
      3.3.1 Related work – OWL-E ..............................................................................26
      3.3.2 The Proposed Approach – Datatype Objects .............................................27
      3.3.3 Semantics of Datatype Objects ...................................................................30
      3.3.4 Units of Measurement ..............................................................................32
      3.3.5 Reasoning with Datatype Objects ..............................................................33
      3.3.6 Using Existing Ontologies ..........................................................................33
   3.4 Collections ..........................................................................................................34
   3.5 User-Defined Classes ..........................................................................................34
4. The HyperFlow Language ................................................................. 35
   4.1 Visual Representation of Ontology Entities .................................. 35
   4.2 Data-Flow Constructs ............................................................... 36
      4.2.1 Modules ................................................................. 36
      4.2.2 Result Sets ............................................................ 37
      4.2.3 Implicit Iteration ....................................................... 38
   4.3 HyperFlow Queries ............................................................... 40
      4.3.1 Query Constraints ......................................................... 40
      4.3.2 And, Or and Not .......................................................... 44
      4.3.3 Joins ........................................................................... 46
      4.3.4 Modifiers for All, Some, Transitively, and Optionally .............. 47
      4.3.5 Down Casting ............................................................... 49
      4.3.6 Selecting Groups ........................................................... 50
      4.3.7 Grouping .................................................................... 51
      4.3.8 Ordering .................................................................... 53
      4.3.9 Removing Duplications ................................................... 54
      4.3.10 Construct and Construct Graph ....................................... 54
      4.3.11 Functions and Methods ............................................... 57
      4.3.12 Undirected Edges ........................................................ 59
      4.3.13 Subqueries ................................................................. 60
      4.3.14 Query Naming and Recursion ....................................... 61
   4.4 Language Analysis ................................................................. 62
   4.5 Related Work and Comparison .................................................. 63
      4.5.1 Visual Query Lanugages ................................................... 63
      4.5.2 Dataflow Languages ...................................................... 70

5. Implementation: The HyperFlow Environment .................................. 74
   5.1 Features ............................................................................... 74
      5.1.1 Windows ....................................................................... 74
      5.1.2 Interaction Mechanisms ................................................. 75
   5.2 The Semantic Framework ......................................................... 78
      5.2.1 Technologies Used ......................................................... 78
         (i) Jena – A Semantic Web Framework for Java ....................... 78
         (ii) OWL-S – An Ontology for Describing Services ................. 79
5.2.2 Ontology Management .................................................................80
5.2.3 The Semantic Utilities ..............................................................80
5.2.4 Service Matchmaking ...............................................................82
5.3 The Graphical Framework ...........................................................83
  5.3.1 Technologies Used .................................................................83
    (i) Eclipse ..................................................................................... 83
    (ii) GEF – The Graphical Editing Framework ...............................85
    (iii) EMF – The Eclipse Modeling Framework ...............................88
5.3.2 GefFace – A Framework for Building Graphical Editors with GEF 89
5.4 Tying it All Together – the Implementation of the HyperFlow Editor ....92
  5.4.1 Model ......................................................................................92
  5.4.2 View .......................................................................................94
  5.4.3 Controller ...............................................................................94
  5.4.4 Actions ..................................................................................95
6. Summary and Further Work ............................................................97
  6.1 Summary ...................................................................................97
  6.2 Further Work ............................................................................97
    6.2.1 Language .............................................................................98
    6.2.2 Implementation ..................................................................98

Appendix 1. ModelTranslator ............................................................ 101
Appendix 2. HyperFlow's Expressive Power ..................................... 103
Appendix 3. Finding The Properties of a Class and their Ranges ......... 110
References ......................................................................................... 114
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A simple bioinformatics analysis process</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>The XSD datatype hierarchy</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>The built-in datatype object hierarchy</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>Built-in predicate restrictions</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>Collections in HyperFlow</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>Iteration over two input collections</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>Iteration over one input collection</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>Passing input through modules</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>Usage modes of various constraints</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>Using a constraint between two schema nodes</td>
<td>42</td>
</tr>
<tr>
<td>11</td>
<td>AND, OR and NOT</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>Specifying execution order</td>
<td>45</td>
</tr>
<tr>
<td>13</td>
<td>Join examples</td>
<td>46</td>
</tr>
<tr>
<td>14</td>
<td>All, transitively</td>
<td>47</td>
</tr>
<tr>
<td>15</td>
<td>Outer joins using optionally</td>
<td>47</td>
</tr>
<tr>
<td>16</td>
<td>Down casting</td>
<td>49</td>
</tr>
<tr>
<td>17</td>
<td>Selecting groups</td>
<td>50</td>
</tr>
<tr>
<td>18</td>
<td>Collect</td>
<td>51</td>
</tr>
<tr>
<td>19</td>
<td>Group by</td>
<td>51</td>
</tr>
<tr>
<td>20</td>
<td>Group by and having</td>
<td>52</td>
</tr>
<tr>
<td>21</td>
<td>Order by</td>
<td>53</td>
</tr>
<tr>
<td>22</td>
<td>Removing duplicates from the result</td>
<td>54</td>
</tr>
<tr>
<td>23</td>
<td>Creating new result objects using construct</td>
<td>55</td>
</tr>
<tr>
<td>24</td>
<td>Creating new ontology objects using construct</td>
<td>55</td>
</tr>
<tr>
<td>25</td>
<td>Constructing a new graph</td>
<td>56</td>
</tr>
<tr>
<td>26</td>
<td>Collection and set operators</td>
<td>57</td>
</tr>
<tr>
<td>27</td>
<td>Boolean constraints as functions</td>
<td>57</td>
</tr>
<tr>
<td>28</td>
<td>Uncorrelated nested query</td>
<td>60</td>
</tr>
<tr>
<td>29</td>
<td>Correlated nested query</td>
<td>60</td>
</tr>
<tr>
<td>30</td>
<td>A recursive subquery</td>
<td>61</td>
</tr>
<tr>
<td>31</td>
<td>Gql queries</td>
<td>63</td>
</tr>
<tr>
<td>32</td>
<td>A MDDQL query</td>
<td>64</td>
</tr>
<tr>
<td>33</td>
<td>A QGraph query</td>
<td>64</td>
</tr>
<tr>
<td>34</td>
<td>A VOQL query (two modes of depicting projection and conditions)</td>
<td>65</td>
</tr>
<tr>
<td>35</td>
<td>A HVQS query</td>
<td>65</td>
</tr>
<tr>
<td>36</td>
<td>QBE queries</td>
<td>65</td>
</tr>
<tr>
<td>37</td>
<td>A VOODOO query</td>
<td>66</td>
</tr>
<tr>
<td>38</td>
<td>A Kaleidoquery query</td>
<td>66</td>
</tr>
<tr>
<td>39</td>
<td>A DFQL query</td>
<td>67</td>
</tr>
<tr>
<td>40</td>
<td>A QUIVER query</td>
<td>67</td>
</tr>
<tr>
<td>41</td>
<td>An Iconic SQL query</td>
<td>68</td>
</tr>
<tr>
<td>42</td>
<td>A Kepler workflow</td>
<td>71</td>
</tr>
<tr>
<td>43</td>
<td>A Taverna workflow</td>
<td>71</td>
</tr>
<tr>
<td>44</td>
<td>A Triana workflow</td>
<td>72</td>
</tr>
<tr>
<td>45</td>
<td>A PipelinePilot workflow</td>
<td>72</td>
</tr>
<tr>
<td>46</td>
<td>An übertool workflow</td>
<td>72</td>
</tr>
<tr>
<td>47</td>
<td>A KDE (Knowledge Discovery Environment) workflow</td>
<td>73</td>
</tr>
</tbody>
</table>
List of Tables

Table 1. OWL-DL Constructs ................................................................. 20
Table 2. Entity icons in HyperFlow ................................................... 35
Table 3. Feature comparison between several VQLs and HyperFlow .......... 69
Abstract

As life-science research becomes increasingly information-centric, bioinformatics databases and analysis-services become increasingly abundant. However, currently the burden of finding the available databases and services, learning how to use each of them, and integrating their functions and formats into unified analysis processes typically falls on the slender shoulders of end-users with practically no programming skills. Thus, one of the major challenges of the bioinformatics community is to successfully harness data-integration, service-oriented architecture, ontologies and advanced user-interfaces towards the goal of enabling users to easily create intricate in-silico experiments as part of their research.

With this goal in mind, we have developed HyperFlow, a novel visual language for information analysis that combines into a unified framework features from visual dataflow and visual query languages. HyperFlow is designed to make it easier for users to retrieve, filter, and manipulate information, using databases alongside e.g. web services, in a transparent, intuitive, reproducible and traceable manner. It allows users to visually design and execute information analysis processes in a single diagram. It lends itself both to the common programming mode of "design and execute" as well as to the ad-hoc mode of "interactive exploration", which is usually used in research. As a visual query language, its expressive power is at the top of its class. In addition, it may be used with relational and object oriented databases, as well as ontologies described in the highly-expressive Web Ontology Language (OWL). Finally, although it was designed with a particular application in mind, HyperFlow is completely domain-neutral.

This thesis describes HyperFlow's features as well as the characteristics and design of the prototype interface we have implemented. This interface is "ontology-aware": it uses the information in the ontology in order to guide the users through graphically composing queries; in addition, it makes use of Semantic-Web-style matchmaking in order to find applicable services to launch. Finally, the interface is being built on top of the Eclipse platform, which provides an extensible architecture that we intend to use towards further development of an extensive, open environment for bioinformatics analysis – the Bioinformatics Assay Environment.
List of Abbreviations

GEF – Graphical Editing Framework  
GI – Graphical Instance  
MVC – Model-View-Controller  
OWL – Web Ontology Language  
SWT – Standard Widget Toolkit  
VQL – Visual Query Language
Chapter 1.

Introduction

1.1 Motivation

Recent years have brought on an explosion of information in many domains. Considerable progress has been made in generic development of database and information retrieval technologies to handle new scales of data diversity and extent. Moreover, new fields – such as bioinformatics and business intelligence – emerged in an effort to create specialized domain-specific tools that help retrieve this information, treat it in various ways, and mine it for interesting conclusions.

Still, quite often the user interface offered by the databases and analysis services is basic, under-expressive, and rigid. This limits the extent to which the information can be analyzed by end-users. For example, the interfaces provided by many public-domain databases are based on web forms. Such an interface typically allows its users to retrieve entries of a single kind in each query. Using the fields in the provided forms, users can specify simple constraints. However, they cannot use even simple mechanisms such as a relational join, not to mention the more complex operations of retrieving a projection over a join, grouping, or applying aggregate functions.

Another major issue that inhibits information analysis is that of integration. There are many databases and analysis services in each domain, and hardly any question can be answered by using just one of them. Thus, users often need to perform complex information handling processes that span multiple resources. These processes can be seen as workflows comprising queries and service executions. Currently, no existing technology allows non-programmer end-users to define such processes easily. Thus, the only way most users can achieve this integration is manually, in a laborious, error prone process which involves finding the useful databases and services, learning how to use each of them, and manually executing them in order. HyperFlow, the language described in this thesis, is designed to allow such end-users to define information-handling processes, including arbitrarily complex queries, in an easy manner.
1.2 Contribution

In order to ease the orchestration of complex database queries and service executions into integrated information handling processes, we suggest leveraging the power and agility of visual languages. To this end, we present HyperFlow [1], a new visual language that represents the "marriage" of two well-known types of visual languages: visual query languages and visual dataflow languages. The whole – the result of this merger – is greater than the sum of its parts, as HyperFlow not only allows the simple combination of queries into workflows, but also provides increased expressive power, intuition, and simplicity, both in posing queries and in specifying workflows.

HyperFlow is designed to support almost all constructs from the SQL, OQL and SPARQL query languages, thus giving the user a unified language to access most common databases available today. Queries are depicted in an intuitive manner, using a combination of declarative and imperative constructs, which mirrors the way end-users think of a complex query. In addition, HyperFlow facilitates easy usage of services. These can be distributed services such as web services, as well as local services such as programs and scripts, and even built-in database capabilities such as GROUP BY and COUNT. Furthermore, workflows can be nested in other workflows, allowing reuse, scoping and encapsulation. Finally, advanced dataflow techniques such as implicit iteration allow users to easily deal with collections of items in a single instruction.

To further support the HyperFlow language I have developed a prototype user interface which provides an easy way to design HyperFlow workflows and queries. By using a special-purpose, ontology-aware editor, users are able to create queries which may be quite complex by simply using a limited set of point-and-click gestures. In addition, the editor prevents users from creating queries which are erroneous in syntax or semantics. Finally, the editor is also aware of the various available services and can suggest, through semantic matchmaking, those that are applicable for usage in the next step of the analysis process. This way, users can find the service they need among the myriad of available services, even if they did not know about the service in advance.

In order to perform the matchmaking, we use Semantic Web technologies: In HyperFlow, the domain is modeled using OWL [2] – the W3C's Web Ontology
Language. OWL's expressive power allows the domain model to be enriched with information that is usually kept outside the domain's schema. Such an enriched schema is called an ontology. In addition, services are described using the Semantic Web service description language [3]. Finally, to further enhance the modeling, I have extended OWL with support for collections and custom datatypes.

Currently, the prototype implementation provides an editor for HyperFlow. Adding execution capabilities will be the subject of future work. When these capabilities are added, the editor will support two modes of operation. In the first mode the user designs the workflow first and then executes it; in the second, the user performs interactive exploration of information (also called ad-hoc analysis). In the latter mode, a user can pose a query, review its results, and then decide on the next step of the workflow. As an added benefit, the workflow that is generated by a session can be stored for traceability purposes and can be re-used at a later point, even on different inputs.

1.3 Outline

The thesis is organized as follows:

Chapter 2 presents our main case study, which is in the field of bioinformatics. Section 2.1 introduces the need of users in this field to perform complex "in-silico" experiments, and what are some of the obstacles they are facing. Then, Section 2.2 describes some of the research and development achievements that can help users to perform these analyses today. Subsequently (Section 2.3), we discuss how HyperFlow can enable (more) users to do so in an easy, integrative manner, and give a possible usage example. We conclude this chapter with an outlook into our goal of using HyperFlow (and other technologies) to create an extensive, open environment for bioinformatics analyses – the Bioinformatics Assay Environment (BAE).

Chapter 3 describes the OWL-based data modeling language used in HyperFlow. In Section 3.1, a short description of OWL is given. Section 3.2 explains the changes in semantics that were needed to allow OWL ontologies to represent information in databases. Then, my extensions to OWL are described: Section 3.3 discusses the support for custom datatypes, and Section 3.3.6 discusses collections.
Chapter 4 features a comprehensive presentation of the HyperFlow language. It starts by describing the graphical representation of ontology entities (Section 4.1). This is followed by a description of the various workflow (Section 4.2) and query (Section 4.3) constructs. Finally, a comparison with other visual languages is given in Section 4.4.

Chapter 5 describes the prototype user interface that I implemented, called the HyperFlow Environment. First (Section 5.1), the implementation's main features are described. The semantic framework that was developed to manage and handle ontologies is described in Section 5.2. Section 5.3 describes the underlying technologies used for the development of the graphical features of the environment. Finally, Section 5.4 details the architecture of the environment and how the graphical and ontology-related components are used together.

Chapter 6 summarizes this work (Section 6.1) and outlines potential further work on the language (Section 6.2.1) and the implementation (Section 6.2.2).
Chapter 2.

Case Study: *In-Silico* Experiments and the Bioinformatics Assay Environment

2.1 Introduction

Life science researchers make increasing use of bioinformatics resources. The increase is not only in quantity (*i.e.* number of requests), but also in the quality of the usage (*i.e.* complexity of the analyses). Many contemporary studies involve novel bioinformatics analyses, also called "*in-silico*" experiments. Scientists need a way to easily define such analyses, in a manner that is traceable and repeatable. Thus, this field presents an acute need for the features that HyperFlow provides. For these reasons, bioinformatics was our main motivation and case study.

The domain of bioinformatics is uniquely complex in the amount and diversity of available resources. As of this writing (August 2005), there are more than 1100 databases [4] and hundreds of services in the public domain alone. In addition, resources are constantly being added, removed and changed. Often, different resources use different formats for the same kind of data, and users need to convert between them. The various schemas of bioinformatics databases are typically intricate, semi-structured and highly-linked among themselves. Thus, integration of more than one resource is usually needed. All of these factors make the task of manually finding the correct resources and performing the analysis processes practically impossible for most end-users. The flow and query mechanisms of the HyperFlow language, as well as the matchmaking capabilities of the interface, could be very useful in defining such experiments.
2.2 Background: Current Technologies for Enabling In-Silico Experiments

In order to allow users to define and execute their complex in-silico experiments, three issues must be addressed: resource access, integration, and user interfaces. These three fields have been getting much attention in bioinformatics. In this section, we will look at some of the state-of-the-art technologies developed in these areas. These technologies can be divided into two groups: those following the centralized approach and those following the distributed approach. We will now describe each of these approaches in turn.

2.2.1 The Centralized Approach

Access and Integration: Many of the most important publicly-available bioinformatics databases can currently be accessed through warehouses such as Entrez [5] at the National Center for Biotechnology Information (NCBI) and SRS [6] at the European Bioinformatics Institute (EBI). Aggregating several databases in a warehouse has several benefits, such as a single access mechanism, a single data format, fast responses for cross-database queries, and the ability to calculate links between databases in advance.

Bioinformatics warehouses use various back-end technologies with various query capabilities. For example, SRS uses a proprietary query mechanism which provides a query language with low expressive power. An even weaker language is provided by Entrez, despite the fact that most of its data is internally stored in a relational database. Other projects (e.g. SRI’s BioWarehouse [7] and UBC’s Atlas [8]) provide a purely relational database schema which can be freely queried, thus increasing the available expressive power. However, they are currently not as comprehensive as SRS and Entrez.

User interface: SRS and Entrez each provide a built-in user interface, which allows users to access in a uniform way the various databases comprising the warehouse. However, these interfaces provide only minimal query expressive power and deliver limited service support. The stronger warehouses mentioned above supply only an SQL interface, thus limiting their usage to users who are proficient in SQL and who have studied the warehouse's complex schema.
2.2.2 The Distributed Approach

In addition to the limitations stated above, bioinformatics warehouses suffer from classical problems of warehouses, such as having to keep information freshness (by constantly updating the databases) and having to maintain comprehensiveness (by constantly adding more resources). A distributed approach to solving the integration problem could, in theory, solve these problems: Given an open system that can accept registration from different resources and provide an illusion of a single unified database, there would be no need to maintain local copies of the information. On the other hand, new problems will be created, such as slower response times due to higher network traffic and lower optimization capabilities. In any case, such an open federated system does not exist yet. However, there is much research and development activity in this direction, as will be discussed below.

Access: The first step towards distributed integration is creating a way to access the resources. Several technologies are being used to achieve this. Perhaps the most successful one (owing its success to its simplicity and limited scope) is the Distributed Annotation System (DAS) [9], which defines a standard for the creation of web services that provide sequence annotations (i.e., chromosomal-location-based information). The DAS standard uses Representational State Transfer (REST), i.e., query in the URL and response as XML. REST is also being used in other projects, for example to enable programmatic access to Entrez.

Other, newer initiatives are based on the industry-standard web services stack, namely SOAP, WSDL, and (less often) UDDI. For example, the BioMOBY [10] project aims to define standards for SOAP-based bioinformatics web services. BioMOBY services are registered using WSDL documents that contain terms from central ontologies of service and object types. The registry (which is not based on UDDI) is called MOBY Central. Currently, more than 140 services are registered in MOBY Central. Another similar initiative is Soaplab [11], which focuses on enabling a SOAP-based interface to command-line bioinformatics analysis programs. Developed at EMBL, it is primarily used to provide access to many of EMBL's services, including the tens of programs comprising EMBOSS [12], a package of high-quality open source software for sequence analysis.
Another technology which is considered promising for service access is grid computing. This technology has the potential to also solve several scalability problems by distributing computation and pertinent data. Pilot projects using grid technology are the European Data-Grid [13], and Bio-GRID [14] projects, North Carolina BioGrid [15], Japanese BioGrid [16], Asia Pacific BioGrid [17], the National Cancer Institute's Cancer Biomedical Informatics Grid (caBIG) [18], and the UK's e-Science Initiative's myGrid [19]. The latter two are ambitious projects which aim to provide not only access to services, but also semantic integration, workflow enactment and user interface capabilities.

After making a service available for easy programmatic access, it is also useful to provide a service description, which allows the service to be discovered by users. There are various ways to describe a service. One such solution, which is gaining increasing interest from the bioinformatics community, makes use of Semantic Web standards. Among these standards are the Web Ontology Language OWL [2] (see Chapter 2) and the service description ontology OWL-S [3] (see Section 5.2.1(ii)). Several notable projects (e.g. myGrid, caBIG, Semantic MOBY [20]) provide semantic descriptions of their services. Services described in this way can be discovered in a process which is often called semantic matchmaking (see Section 5.2.4 for our implementation).

Semantic Web technologies are being used in the bioinformatics community for several other purposes, such as describing schemas of databases. For more information, see the position paper archive of the W3C Workshop on the Semantic Web for Life Sciences [21], which includes our own paper [22].

Integration: In order to create a useful distributed architecture which supports features such as complex queries and transparent service execution, more than programmatic access to resources is needed. To this end, several systems exist that act as mediators, presenting a virtual common schema to the user. A mediator knows how to process complex queries written over this schema, by generating a distributed query plan which consists of dispatching sub-queries to the individual resources and integrating the results. In the context of life sciences, some of the notable mediator systems are BioKleisli/K2 [23], DiscoveryLink [24], BioMediator [25], BACIIS [26], TAMBIS [27], and myGrid's OGSA-DQP [28].
Another important type of mediation is workflow enactment. In recent years, the adoption of service oriented architecture had led to a myriad of workflow description languages and enactment systems (e.g. BPEL4WS [29]). However, as we discuss in Section 4.5.2, scientific workflows often have requirements that are not met by these business-oriented standards. Two notable middleware systems that enable such scientific workflows are myGrid's FreeFluo [30] component and Scitegic's Pipeline-Pilot [31] server.

User Interface: After creating a way to access, find and integrate disparate databases and services, one challenge still remains: making them accessible to end-users. One possible method to do so is thorough the means of programming. Perl [32] is a very popular language in bioinformatics. Many reusable Perl modules are available for accessing resources, parsing information from flat-files and handling information in various ways. Many of these modules are contained in the well-known open-source bioinformatics library BioPerl [33]. Similar libraries exist for other languages, such as BioJava [34] and BioPython [35]. Database query languages and modules are also widely used. However, most bioinformatics end-users are not programmers and do not know how to query databases without some specialized user interface. Thus, unfortunately, most end-users today do not perform complex analyses by themselves.

To help end-users perform complex analysis processes, graphical user interfaces are needed. For instance, several systems provide a method for defining workflows using a graphical editor. Among those are: myGrid's Taverna [36] component, PipelinePilot's desktop client (both work with their respective enactment mediators), Kepler [37], BioSPICE [38], Science Factory's überTool [39], InforSense' KDE [40] (Knowledge Discovery Environment), and Incogen's Visual Integrated Bioinformatics Environment (VIBE) [41].

None of the workflow projects supply visual means for defining queries. As discussed above, bioinformatics queries may be quite complex and almost always involve more than one database. This challenge was the focus of the Transparent Access to Multiple Bioinformatics Information Sources (TAMBIS) [27] project. TAMBIS provided a user interface that allowed scientists to graphically define queries over a virtual schema. The queries were then sent to the TAMBIS mediator, and the results were displayed in a table. However, TAMBIS suffered from the low
expressive power of its queries, the inability to directly invoke services or compose workflows, and the lack of visualization capabilities for the different bioinformatics datatypes.

Bioinformatics features many kinds of data objects which require specialized visualization capabilities. For example, several applications exist for displaying 3D structures, and they typically include advanced features such as rotating the structure, switching between alternate presentation forms (e.g. balls and sticks, space-filling), and more. As another example, sequence-based annotations are often displayed over an axis that represents the sequence, thus showing their location in the sequence. Finally, life-science research (as any empirical research) often requires statistical analysis and visualization.

Many of the workflow-based solutions mentioned above also provide visualization to some extent. Several additional noteworthy applications are: GeneMine [42] (providing search for and visualization of various sequence, structure, and function related information), OmniView [43] (providing retrieval from various DAS resources and visualization of the annotation on a single sequence axis), and ISYS [44].

ISYS is a framework for enabling client-side interoperability of loosely-coupled components, each providing either: resource access, some analysis capability or visualization. ISYS gave users much more power than using any component alone, as it allowed giving one component an input from a different component, getting suggestions for applicable services and viewers in a manner that resembles semantic matchmaking, and even filtering of objects in a manner which is coordinated across visualization components. The latter feature, which is a form of direct manipulation of information, allows users to achieve complex joins in an ad-hoc manner, without formulating a query and without needing a mediator. Finally, the simplistic, loosely coupled and open nature of ISYS was supposed to enable the development of many components, increasing its usefulness. In reality, too few components were written, and the project was discontinued. In addition, the project had several drawbacks: It did not intend to give a consistent user interface to different services, not even to recurring tasks such as querying databases. Furthermore, it did not allow users to pose complex cross-database queries. Lastly, although ISYS allowed scientists to use resources in a complex manner – thus creating ad-hoc information-handling processes
it had no support for depicting them as workflows, nor any support for advanced workflow features such as iteration.

2.3 Using HyperFlow to Define an In-Silico Experiment

We will now see how HyperFlow can help users define in-silico analyses easily by visually composing service executions and complex queries into a unified analysis workflow. A sample HyperFlow diagram, depicting a very simple bioinformatics analysis process, is shown in Figure 1. A researcher who has isolated a DNA molecule in the lab wishes to study it by comparing – in various ways – its sequence to sequences in public databanks.

The user starts her analysis session with an execution of the BLAST [45] service (in Figure 1a), which finds alignments to similar sequences in the databank. The alignments are depicted using a result-set box. Using these alignments, the scientist poses a follow-up query that requests the actual sequences that these alignments represent. She further constrains the query to return only mammalian sequences whose similarity of alignment is more than 98%. This query is depicted by a query box (Figure 1b) – a visual construct that provides a clear encapsulation of the query itself and is naturally integrated within the overall dataflow diagram.

With the relevant sequences from the databank at hand, the scientist now proceeds to analyze them in three different ways. First, she decides to compare the original sequence she isolated in the lab against each of the sequences she retrieved. She does so by running (Figure 1c) a global alignment algorithm, once for each pair of the original and retrieved sequences. Next, she launches (Figure 1d) a service that gives a multiple alignment of the retrieved sequences. Finally, to understand how the genes encoded by the database sequences are controlled, the scientist reuses a workflow (Figure 1e) she once prepared that finds which transcription factors are common to all the sequences in the group.
Expressing this kind of an analysis would be rather difficult by using queries alone or dataflow alone. It is the combination of these two paradigms, the reliance on views, and the introduction of constructs such as implicit iteration, that make it possible to construct this type of a session. Moreover, the pictorial rendering of the whole process makes it both clear as well as well-defined at the same time.
Towards the Bioinformatics Assay Environment

While HyperFlow can be very useful to bioinformatics end-users, by itself it is not enough. Our goal to allow end-users to easily perform complex analyses would not be complete without incorporating comprehensive resource access and integration mechanisms, visualization capabilities, statistics and more. Thus, the long-term goal of my research is to create an integrative environment for bioinformatics analyses – the Bioinformatics Assay Environment (BAE).

The BAE will enable users to easily access information, analyze it and visualize it in various ways. Via HyperFlow, it will support creating and executing pre-designed and ad-hoc workflows, including visually creating complex queries which are conveniently located in the same diagram as the workflow. The system will be able to use one or more of the federated mediators or warehouses (see Section 2.2) to answer these queries. It will also provide access to services which use standard remote interfaces such as DAS [9], BioMOBY [10], Soaplab [11] and myGrid [19], and local libraries such as BioPerl [33], BioJava [34] and EMBOSS [12], thus allowing usage of the hundreds of resources which use those protocols.

The system will maintain provenance information (i.e. where did each piece of information come from, what version of the database or service was used, etc.) to increase traceability. It will include both generic as well as domain-specific viewers that will provide visualizations both via charts and tables as well as domain concepts such as 3D molecular structures, sequences, alignments, and interaction networks. Ultimately, these viewers will also support analysis via direct-manipulation. Finally, viewers and services that support statistical analyses should also be included, perhaps by integration with the statistical language R [46].
Chapter 3.

The Modeling Language

At the base of HyperFlow lies an expressive object-oriented data model, based on W3C's Semantic Web Ontology Language (OWL) [2]. OWL (described in detail in Section 3.1) provides increased expressive power in comparison with the data models of object-oriented and relational databases, adding features such as individuals, enumerations, local class restrictions, Boolean class expressions, a property hierarchy and transitive properties (see Section 3.1.4). Its reliance on a globally-unique naming mechanism (Section 3.1.2) allows a class or property defined in any ontology to reference and even extend classes and properties from other ontologies – thus creating a sort of "global" ontology. In order to use OWL to represent database schemas, its semantics must be slightly modified. This is discussed in Section 3.2.

Describing service inputs and outputs in OWL facilitates advanced semantic matchmaking, which enables the HyperFlow environment to help users choose among the numerous available resources. In order to enhance the description of the domain and the services, I further extended OWL by introducing custom datatypes and collection classes. These are discussed in Sections 3.2 and 3.3.6, respectively.

3.1 Background: OWL – The Semantic Web's Modeling Language

3.1.1 Introduction

The vision of the Semantic Web [47] is to enable automatic interoperation among entities on the Web. Such interoperability can be achieved through annotation of the content on the Web with machine-processable ontological terms. Ontologies [48] are formal and explicit specifications of certain domains. An ontology typically consists of a number of classes, a number of relations (usually called properties) between these classes, a number of instances and a number of axioms. These elements are all expressed using some logical language.
In order to allow sharing and reuse of ontologies on the Semantic Web, the World Wide Web Consortium (W3C) has developed two ontology languages. The first is RDFS [49], which was developed as a lightweight ontology language; the second language is OWL [2], which is a more expressive ontology language based on Description Logics [50]. Users of OWL thus have the benefit of being able to reason over the ontology to check consistency and determine logical consequences (inferences), as will be described below.

3.1.2 Universal Resource Identification and its Implications

OWL features several characteristics that make it more than just a syntactic variant of Description Logics. One of them is its identification mechanism: In order to support distribution of ontologies over the Web, a unique identifier is needed for each resource (class, instance, property). For this purpose OWL uses Universal Resource Identifiers (URIs). Many times, ontology designers actually use Uniform Resource Locators (URLs), which are a kind of URIs. In these cases, the name also describes the access method. Other types of URIs, such as the Life Sciences Identifiers (LSIDs) [51], are Universal Resource Names (URNs) and require resolution for accessing the resource. An example of an LSID is:

\texttt{urn:lsid:ncbi.nlm.nih.gov:GenBank:T48601:2} which represents the \textit{individual} (\textit{i.e.} instance) whose id is T48601:2 in the GenBank database. Whatever identification method is used, having URIs as resource names allows the creator of one ontology to reference and use resources from other ontologies in a seamless way, creating a sort of "global ontology".

A URI can always be divided into two parts: the namespace and the local name. For example, when using URLs, the URL of the document defines the namespace and the name of the element inside the document is the local name. They are separated by \#. For legibility reasons, namespaces are usually replaced with short names (prefixes), separated from the local names by a ":". For example, to reference the top-level OWL concept \texttt{Thing} (\url{http://www.w3.org/2002/07/owl#Thing}) we will write \texttt{owl:Thing}.

As mentioned above, URIs are used in OWL in order to create "global" ontologies. That is, any ontology can reference classes or properties from any other ontology by using its URI. For example, to demonstrate the representation of sequence alignments, I have created a sample alignments ontology, which includes
classes like the `NucleotideSequenceAlignment` class. This class represents an alignment found in services like BLAST [45] – i.e. an alignment of an input sequence with a sequence found in the database. These sequences are referenced using the properties `inputSequence` and `foundDatabaseSequence`, whose ranges are the classes `NucleotideSequenceString` and `NucleotideSequenceEntry`, respectively. Both classes are not defined in the `alignments` ontology, but rather in the `sequences` ontology. This allows the different ontologies to be developed independently, and even in different locations and organizations, as long as the URIs of the classes remain stable.

The usage of URIs also allows defining class hierarchies that span multiple ontologies. Each class can be defined as extending (using `subClassOf`) any other class, including classes from other ontologies. For example, the `NucleotideSequenceEntry` class is a subclass of the `DBEntry` class from the `informatics` ontology. This class has properties that are appropriate for entries of many bioinformatics databases, such as `creationDate`, `version`, and `description`. This allows reuse and enhances modeling, without sacrificing the distribution and independence of the different ontologies.

### 3.1.3 OWL Subspecies

The OWL language provides three increasingly expressive sublanguages, OWL Lite, OWL DL and OWL Full. Each of these sublanguages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded. OWL Lite is a notational variant of the Description Logic $\text{SHIF}(D)$. OWL DL is a notational variant of the Description Logic $\text{SHOIN}(D)$ [52]. It is called DL since it is actually the largest part of OWL that fits in the Description Logic framework, and thus known to have decidable reasoning. The most expressive species of OWL, OWL Full, has no corresponding Description Logic. This allows OWL Full ontologies to remove several restrictions inherent in Description Logics. For example, it foregoes the distinction between instances and classes (ABox and TBox in DL terminology) and allows resources to be treated as both classes and instances in different contexts. While this provides increased expressive power for specifying ontologies, there is no known decidable algorithm for reasoning over OWL Full.
3.1.4 OWL-DL Syntax

We are only concerned here with the most well-known and most investigated species of OWL, namely OWL DL. OWL ontologies can be written in various syntaxes, most notably the RDF/XML syntax. For reasons of legibility, in the remainder of this section I will describe OWL DL using Description Logics syntax and OWL Abstract Syntax.

A summary of OWL's constructs is given in Table 1. In the next pages, each OWL construct will be further explained. Examples from my sample bioinformatics ontology are given for select constructs.

The basic building blocks of OWL ontologies are:
1. **Classes** are either:
   a. A named class (declared using `owl:Class` with a URI)
   b. A class expression, as described below.
   
   • Named classes may be related to class expressions (and thus described – hence the name description logics) by using the two class axioms: `owl:subClassOf (⊑)` and `owl:equivalentClass (≡)`, both described below.

2. **Datatypes** that OWL supports are:
   a. XML Schema datatypes [53] (e.g. `xsd:Integer`)
   b. Enumerations of datatype values (constructed using `owl:oneOf(...)`)  

3. **Properties** (`owl:ObjectProperty` or `owl:DatatypeProperty`). Note that unlike typical properties in object-oriented programming languages and frame-based ontology languages, description logic properties are not defined as a part of the class. Hence, different classes can share the same (uniquely named) property. The association of properties to classes is defined in the property's universally-defined domain (`rdfs:domain`). Similarly, a property may have a universally-defined range (`rdfs:range`).

4. **Object instances** (`owl:Individual`): any instance can be a member of more than one class.

5. **Datatype values** (e.g. `10^^xsd:integer`)
Table 1. OWL-DL Constructs

<table>
<thead>
<tr>
<th>OWL Abstract Syntax</th>
<th>DL syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class axioms</strong></td>
<td></td>
</tr>
<tr>
<td>Class(A partial C₁ ... Cₙ)</td>
<td>A ∈ Cᵢ</td>
</tr>
<tr>
<td>Class(A complete C₁ ... Cₙ)</td>
<td>A ⊑ Cᵢ ⋱ Cₙ</td>
</tr>
<tr>
<td>EnumeratedClass(A a₁ ... aₙ)</td>
<td>A ⊑ {a₁, ..., aₙ}</td>
</tr>
<tr>
<td>SubClassOf(C₁ C₂)</td>
<td>C₁ ⊑ C₂</td>
</tr>
<tr>
<td>EquivalentClasses(C₁ ... Cₙ)</td>
<td>C₁ ⊑ ... ⊑ Cₙ</td>
</tr>
<tr>
<td>DisjointClasses(C₁ ... Cₙ)</td>
<td>C₁ ∩ ... ∩ Cₙ ⊑ ⊥</td>
</tr>
<tr>
<td><strong>Property axioms</strong></td>
<td></td>
</tr>
<tr>
<td>ObjectProperty(R)</td>
<td>R ⊑ Rᵢ</td>
</tr>
<tr>
<td>super(R₁) ... super(Rₙ)</td>
<td>R ⊑ Rᵢ</td>
</tr>
<tr>
<td>domain(C₁) ... domain(Cₙ)</td>
<td>T ⊑ ∨R⁻ Cᵢ</td>
</tr>
<tr>
<td>range(C₁) ... range(Cₙ)</td>
<td>T ⊑ ∨R Cᵢ</td>
</tr>
<tr>
<td>[InverseOf(Rₚ)]</td>
<td>R ⊑ R⁻</td>
</tr>
<tr>
<td>[Symmetric]</td>
<td>R ⊑ R⁻</td>
</tr>
<tr>
<td>[Functional]</td>
<td>T ⊑ ϵ 1R</td>
</tr>
<tr>
<td>[InverseFunctional]</td>
<td>T ⊑ ϵ 1R⁻</td>
</tr>
<tr>
<td>[Transitive]</td>
<td>Trans(R)</td>
</tr>
<tr>
<td>DatatypeProperty(U)</td>
<td>U ⊑ Uᵢ</td>
</tr>
<tr>
<td>super(U₁) ... super(Uₙ)</td>
<td>U ⊑ Uᵢ</td>
</tr>
<tr>
<td>domain(C₁) ... domain(Cₙ)</td>
<td>T ⊑ ∨U⁻ Cᵢ</td>
</tr>
<tr>
<td>range(T₁) ... range(Tₙ)</td>
<td>T ⊑ ∨U Tᵢ</td>
</tr>
<tr>
<td>[Functional]</td>
<td>T ⊑ ϵ U</td>
</tr>
<tr>
<td>SubPropertyOf(Q₁ Q₂)</td>
<td>Q₁ ⊑ Q₂</td>
</tr>
<tr>
<td>EquivalentProperties(Q₁ ... Qₙ)</td>
<td>Q₁ ⊑ ... ⊑ Qₙ</td>
</tr>
<tr>
<td><strong>Individual assertions</strong></td>
<td></td>
</tr>
<tr>
<td>Individual(o)</td>
<td>o ∈ Cᵢ</td>
</tr>
<tr>
<td>type(C₁) ... type(Cₙ)</td>
<td></td>
</tr>
<tr>
<td>value(R₁ o₁) ... value(Rₙ oₙ)</td>
<td>(ω₀, o₀) ∈ Qᵢ</td>
</tr>
<tr>
<td>value(U₁ t₁) ... value(Uₙ tₙ)</td>
<td>(ωᵢ, tᵢ) ∈ Uᵢ</td>
</tr>
<tr>
<td>SameIndividual(o₁ ... oₙ)</td>
<td>o₁ = ... = oₙ</td>
</tr>
<tr>
<td>DifferentIndividuals(o₁ ... oₙ)</td>
<td>oᵢ ≠ oⱼ, i ≠ j</td>
</tr>
</tbody>
</table>

(b) Class Expressions

<table>
<thead>
<tr>
<th>A (URI Reference)</th>
<th>⊑ A</th>
</tr>
</thead>
<tbody>
<tr>
<td>owl:Thing</td>
<td>⊑ T</td>
</tr>
<tr>
<td>owl:Nothing</td>
<td>⊑ ⊥</td>
</tr>
<tr>
<td>intersectionOf(C₁ ... Cₙ)</td>
<td>⊑ C₁ ⋱ Cₙ</td>
</tr>
<tr>
<td>unionOf(C₁ ... Cₙ)</td>
<td>⊑ C₁ ⋱ Cₙ</td>
</tr>
<tr>
<td>complementOf(C)</td>
<td>⊑ ~C</td>
</tr>
<tr>
<td>oneOf(o₁ ... oₙ)</td>
<td>⊑ {o₁, ..., oₙ}</td>
</tr>
<tr>
<td>restriction(R someValuesFrom(C))</td>
<td>⊑ R.D</td>
</tr>
<tr>
<td>restriction(R allValuesFrom(C))</td>
<td>⊑ ∨ R.D</td>
</tr>
<tr>
<td>restriction(R value(o))</td>
<td>⊑ R.o</td>
</tr>
<tr>
<td>restriction(R minCardinality(n))</td>
<td>⊑ n.R</td>
</tr>
<tr>
<td>restriction(R maxCardinality(n))</td>
<td>⊑ n.R</td>
</tr>
<tr>
<td>restriction(U someValuesFrom(T))</td>
<td>⊑ U.T</td>
</tr>
<tr>
<td>restriction(U allValuesFrom(T))</td>
<td>⊑ ∨ U.T</td>
</tr>
<tr>
<td>restriction(U value(t))</td>
<td>⊑ U.t</td>
</tr>
<tr>
<td>restriction(U minCardinality(n))</td>
<td>⊑ n.U</td>
</tr>
<tr>
<td>restriction(U maxCardinality(n))</td>
<td>⊑ n.U</td>
</tr>
</tbody>
</table>
A class expression is either:

1. \(\{i_1, \ldots, i_n\}\): an enumeration of instances (using \texttt{owl:oneOf})
   
   Example: \texttt{DNAstrand} \(\equiv\) \{positive-strand, negative-strand \}

2. a property range restriction – these are local restrictions which restrict the range of a certain property only for the described class:
   
   a. \(\exists P.C, \texttt{owl:someValuesFrom}:\) states that some of the values of property \(P\) are instances of class \(C\).
      
      Example: from the Gene Ontology [54] (GO) Molecular Function taxonomy:
      
      \texttt{go:follicle-stimulating-hormone-activity} \(\subseteq\)
      
      \(\exists\texttt{go:partOf.go:autonomic-nervous-system-development}\)
   
   b. \(\forall P.C, \texttt{owl:allValuesFrom}:\) all the values of property \(P\) are instances of class \(C\).
      
      Example: A molecular function can be only a part of another molecular function, and not of any other class, such as \texttt{go:CellularLocation}:
      
      \texttt{go:MolecularFunction} \(\subseteq\) \(\forall\texttt{go:partOf.go:MolecularFunction}\)
   
   c. \(P \ni i, \texttt{owl:hasValue}:\) one of the values of property \(P\) is the individual \(i\).
      
      Example: Any Accession has two properties - \texttt{accessionString} and \texttt{db}, which is used for stating the database.
      
      GenBank sequences can only have GenBank accessions:
      
      GenbankSequence \(\subseteq\) \(\forall\texttt{hasAccession(db \ni \texttt{GenBank})}\)

3. A cardinality restriction – a local restriction which restricts the number of distinct values (cardinality) of property \(P\):
   
   a. \(\geq n P, \texttt{owl:minCardinality}:\) \(P\) has at least \(n\) distinct values
      
      Example: From BioPax [55] – and ontology for representing pathways:
      
      Interaction \(\subseteq\) (participants \(\geq 1\))
   
   b. \(\leq n P, \texttt{owl:maxCardinality}:\) \(P\) has at most \(n\) distinct values
   
   c. \(= n P, \texttt{owl:cardinality}:\) \(P\) has exactly \(n\) distinct values

4. A Boolean class constructor (allowing inductively constructing complex class expressions):
   
   a. \(C_1 \cap \ldots \cap C_n, \texttt{owl:IntersectionOf}.\) The semantics of intersection is similar to that of multiple inheritance (multiple \texttt{subClassOf} statements)
      
      Example: \texttt{ProteinSequenceEntry} \(\supseteq\) \texttt{ProteinSequence} \(\cap\) \texttt{SequenceEntry}
b. $C \sqcup \ldots \sqcup C_n$, owl:UnionOf

Example: The sequence property connects either a Sequence or SequenceFeature to their actual sequence string. Thus, its domain (see below) is defined as:

Sequence $\sqcup$ SequenceFeature.

c. $\neg C$, owl:ComplementOf

The class hierarchy in OWL is defined by the following class axioms:

1. $C \sqsubseteq D$, owl:subClassOf: called class subsumption in Description Logics terms.
2. $C \equiv D$, owl:EquivalentClass: this is equivalent to ($ (C \sqsubseteq D) \text{ and } (C \sqsupseteq D) $).
3. owl:disjointWith: equivalent to ($C \sqcap D \sqsubseteq \perp$ (where $\perp$ is the bottom class)).

Similarly, the property hierarchy is defined by the following property axioms:

1. $\sqsubseteq$, rdfs:subPropertyOf

Example: from BioPax: controlled $\sqsubseteq$ participants

2. $\equiv$, owl:equivalentProperties

In addition, OWL also supports the following property characteristics:

1. owl:TransitiveProperty

Example: The go:partOf property in the Gene Ontology.

2. owl:SymmetricProperty

Example: Isoformic proteins (instances of ProteinSequenceEntry) are related by the symmetric property proteomics:isoform.

3. owl:inverseOf

Example: The proteomics:inFamily property, that relates a Protein to its ProteinFamily, is the inverse of proteomics:familyMember

4. owl:FunctionalProperty – is a property that can have only one distinct value

Example: Each protein domain in the PFAM database is represented by a single Hidden Markov Model (HMM). Thus, the proteomics:hasHMM property, which relates a PFAMDomain to its PFAMHMM, is functional.
5. **owl:InverseFunctionalProperty**

   Example: The `informatics:hasAccession` property, which is used to state the Accession of a DBEntry is inverse-functional. That is – the accession can uniquely identify the database entry.

In Description Logics, the part of the ontology which contains class descriptions is called the *terminology box*, or TBOX. In addition, *assertions* about individuals are stored in the ABOX. The OWL ABox contains the following assertions:

1. \( i \in C, \text{rdf:type:} \) class membership (\( i \) is an instance of class \( C \)).
2. \(<i_1, i_2> \in P, <i, v> \in P\): property value (for object and data properties, respectively)
3. \( i_1 = i_2, \text{owl:sameAs:} \) individual equality.
4. \( i_1 \neq i_2, \text{owl:differentFrom:} \) individual inequality.

OWL makes a distinction between object properties and datatype properties, where an object property may only have a class as its range and a datatype property may only have a datatype as its range. Classes and datatypes are disjoint. This yields a significant lack in expressive power: declaring new datatypes is not possible. A possible solution for this problem is given in Section 3.2.

### 3.1.5 **OWL DL Semantics and Reasoning**

OWL DL has a direct model-theoretic semantics [56] similar to the model-theoretic semantics for Description Logics [50]. It was shown that *entailment* in OWL DL (checking whether one ontology is a logical consequence of another) can be reduced to *satisfiability checking* in Description Logics [52]. Since all interesting reasoning tasks can be reduced to Description Logic (un)satisfiability, it is safe to say that OWL DL is a notational variant of a Description Logic. Furthermore, it was shown that Description Logics are a subset of First-Order Logic [57], thus OWL DL is also a notational variant for a subset of First-Order Logic.

There exist several implementations for reasoning with Description Logics (*e.g.* FaCT++ [58], RACER [59], Pellet [60]) which implement different reasoning tasks in Description Logic languages, using a technique called *tableaux algorithms*. Aided by sophisticated optimization techniques, these reasoners are able to handle real-life
problems, and scale up to thousands of concepts, even though the complexities of reasoning may reach ExpTime for OWL-Lite and even NExpTime for OWL-DL [52].

Two important reasoning tasks in Description Logics are *class membership checking* and *subsumption checking* [50]. The class membership inference is used to check whether an individual is a member of a specific class. Subsumption checking amounts to checking whether one class is a subclass of another, i.e. checking whether one concept is more specific than another concept. Using subsumption checking can assist in achieving several Semantic Web goals. One of these – matchmaking Semantic Web Services and candidate input classes – is one of the important features of the HyperFlow interface, and is further described in Section 5.2.4.

3.2 *The Semantics of using OWL in HyperFlow*

HyperFlow is designed to be a language that allows users to easily define complex queries over *existing* databases and orchestrate the execution of *existing* services. These resources typically do not use OWL as their schemata. Instead, they use other modeling paradigms, such as relational, semi-structured or object oriented databases. It is the intention that the schemata of these resources be mapped to OWL ontologies, just as relational databases are often mapped to object-oriented schemata using object-relational mapping (ORM) technologies. The high expressive power of OWL, as well as its capability for distribution of ontologies, can then be used to enrich the translated schema by adding extra information to the ontology. This brings into the ontology information which originally was not a part of the schema, but was rather included in the resource documentation or in the user interface. Thus, there should be no need to include hard-coded domain-specific knowledge in the implementation of the HyperFlow language.

For example, a link between two entry types (A and B) in disparate databases is usually accomplished by keeping the entry IDs of class/table B in a field of class/table A. The documentation of table A states the meaning of this field. A translation of this case to an enriched OWL ontology would include an object property that connects between the two classes (A and B), even though they are in different ontologies.

A different example involves (local) class restrictions. In regular databases, there is no way to state that a given database contains, for instance, just human sequences. This extra information would be included in the accompanying documentation.
However, by using OWL’s $\forall$ restriction, adding such information to the class is easy. As will be shown in the next sections, ontologies can be enriched even further by using my extensions to OWL, especially datatype objects.

In order to allow OWL ontologies to define database schemata, we must use them with slightly non-standard semantics. OWL ontologies are based on the open-world assumption. Thus, OWL restrictions are mostly used to infer new knowledge from given facts. On the other hand, databases use closed-world models, and assume correctness of the stored data. When coming to describe database schemata using OWL, we naturally need to adopt this closed-world semantics. We can then consider OWL class restrictions as expressive constraints on properties, which can be used to describe the classes as discussed above. This approach is roughly along the lines of OWL-Flight [61], an extension to OWL which is being developed by the Web Service Modeling Ontology (WSMO) working group. In their suggestion, the authors suggest adding constraint constructs to OWL – one constraint for each restriction, to enable supporting both open-world and closed world semantics.

3.3 Custom Datatypes

Although OWL provides considerable expressive power for describing classes, its datatype formalism lacks similar expressivity. This is especially evident when creating semantic descriptions for services, to allow for semantic matchmaking. To approach this problem, I came up with the following desiderata for custom datatypes:

1) Semantically-different datatypes should be distinguishable, even if they are syntactically equivalent. For example, a DNASequenceString should be semantically different from a ProteinSequenceString, even though they are both strings. A service should be able to state in its description that it can take as input only one of them, and thus should not be found when searching for services that can take as input the other.

2) The datatypes should be organized in an extensible hierarchy, so that custom datatypes will be considered sub-types of basic datatypes, or even of other custom datatypes. For example, a new datatype called Length could be declared as derived from float, and other new datatypes, LengthInCms and LengthInInches, could be declared as its sub-types.
3) It should be possible to create new datatypes by applying various *constraints* to existing datatypes (*e.g.* \(x\leq100, \text{regex}(x) = /\text{hello}/\)).

4) It should be possible to create datatypes that combine several previously-defined datatypes using Boolean operators (*e.g.* Is-Integer(\(x\)) \& (\(x < 10 \mid x \geq 100\))).

5) Finally, it could be useful to enable imposing further semantic constraints on datatypes, as can be done for classes. For example, we would like to be able to semantically describe a service which takes *human* chromosomal coordinates and returns the *human* genomic sequence between those coordinates. Given a datatype \(\text{ChromosomalCoordinate}\) which is an *Integer*, this service would declare its inputs to be of the type:

\[
\text{HumanChromosomalCoordinate} \equiv \text{ChromosomalCoordinate} \cap (\forall \text{hasOrganism}.\text{Homo-Sapiens}).
\]

The root of the problem lies in the fact that in OWL, as described above, datatypes are treated differently than classes; they cannot be defined using expressions, cannot have properties and cannot be *owl:subClassOf*, *owl:equivalentClass*, or *owl:disjointClass* to other datatypes.

### 3.3.1 Related work – OWL-E

Recently, Pan and Horrocks [62] proposed an extension to OWL that aims to resolve some of these shortcomings. In their proposal, called OWL-E, the authors introduce a general method to represent custom datatypes. They add to OWL the notion of datatype predicates (such as <, =) and datatype expressions. When combined, those additions allow creating custom datatypes, including complex datatypes that involve Boolean combinations of predicates (*e.g.* \(x < 10 \& x \geq 100\)). This is done while keeping the above-mentioned distinction between datatypes and classes, using *datatype-groups* as the underlying semantics. In order to use custom datatypes in class construction, to construct classes such as \(\text{Adult} \equiv \text{Person} \cap \forall \text{age}(> 18)\) the proposal also introduces 4 new restriction constructs, which mirror the \(\forall, \exists, \leq, \geq\) restrictions. Finally, the authors also showed that OWL-E is still decidable, i.e. reasoners for inferencing over OWL-E could be implemented.

The OWL-E proposal goes a long way towards solving the datatype expressivity problem. However, it is still far from becoming part of the OWL standard, and currently has no support from programming APIs such as Jena and the available
reasoners. Lastly, due to the remaining distinction between datatypes and classes, requirement 5 still cannot be fulfilled in OWL-E.

3.3.2 The Proposed Approach – Datatype Objects

In HyperFlow, I propose and use a different approach to describing datatypes: Datatype Objects. For each primitive XSD datatype [63], I defined a corresponding class object, much the way that modern object-oriented languages (e.g. Java, C#) have classes that encapsulate primitive types. Thus, the resulting Datatypes Ontology contains (under a common ancestor `Datatype`) classes such as `String` and `Integer`. These objects are organized in a hierarchy which mirrors, but expands on, the built-in XSD datatype hierarchy (Figure 2). In addition, I added a few "abstract" classes which allow semantically grouping together datatype classes with related semantic meanings. The datatype object hierarchy is shown in Figure 3.

**Definition:** the abstract data-types are: `Datatype`, `AbstractString`, `Number`, `RealNumber`, `Binary`, `TemporalDatatype`, `ConcreteTime`, and `RepeatingTemporal`.

![Figure 2. The XSD datatype hierarchy](image)
Figure 3. The built-in datatype object hierarchy
My approach provides a number of advantages, which allow me to fulfill the above requirements without waiting for official extensions such as OWL-E. First, in order to create a custom datatype class, one simply has to construct a new class and define it to be a subclass of a Datatype-derived class, either built-in or custom (requirements 1 & 2). Since these are normal classes, they can be constructed using OWL Boolean expressions, unionOf, intersectionOf and complementOf (requirement 4). In addition, the Restriction constructs can be used as in any class (requirement 5).

While this approach does away with the distinction between datatype properties and class properties, it still allows building perfectly legal OWL-DL ontologies, which can be reasoned over by existing reasoners. Using the classes derived from Datatype in class and property descriptions instead of using XSD datatypes and DatatypeProperties allows us to more accurately describe services and perform the matchmaking.

In order to fulfill requirement 3 (using datatype constraints), more work has to be done. Basically, we need a generic way to constraint the range of a datatype, using "atomic" constraints as well as complex constraint expressions. We also need to be able to support a limited kind of reasoning over the constrained datatypes. Namely, we want to be able to check subsumption between Datatypes.

My approach follows the predicate approach suggested in OWL-E, but whereas Pan and Horrocks introduce new language constructs for predicates, I introduce new classes, therefore remaining in OWL-DL. Just as OWL-DL allows construction of classes through restrictions, which are classes themselves, my approach allows datatypes to be constructed using PredicateRestrictions, which are Datatypes themselves. Several such built-in restrictions exist (see Figure 4), some obvious examples of which are LessThanRestriction, GreaterThanRestriction, and EqualsRestriction. Other restrictions, such as LengthRestriction, are the representation of well-known XSD facets [63]. Representing restrictions as classes allows us to combine them arbitrarily using the normal Boolean operators: \( \cap \), \( \cup \) and \( \neg \). In addition, it allows the definition of new custom predicates (e.g. DividesByRestriction) and their easy inclusion in datatype class construction.
To be able to use the predicate we need a way to specify the predicate's right-hand-side parameter. This is achieved by using a property called `predicateParameter`, constrained using the \( \ni \) restriction. The right-hand-side of the predicate expression is an individual. Datatype individuals are by convention named `type_value`.

For example, if we want to specify a datatype that consists of ages greater than 18 and less than 30, we can write:

\[
\text{Age} \ni \\
\text{(GreaterThanRestriction} \ni (\exists \text{predicateParameter.Age}_{18})) \ni \\
\text{(LessThanRestriction} \ni (\exists \text{predicateParameter.Age}_{30}))
\]

### 3.3.3 Semantics of Datatype Objects

#### (i) Value Spaces

Each of the non-abstract built-in datatypes shown in Figure 3 takes its value-space from the definition of its XML Schema Datatypes (XSD) [63] counterpart. Similarly, as in XSD, the semantics of the datatypes subclass relationship is that of a value-space subset. *E.g.:* \( \text{value-space(NonNegativeInteger)} \subseteq \text{value-space(Integer)} \).

For the abstract datatypes, the value-space is defined as follows:

- \( \text{value-space(Number)} \equiv \mathbb{Z} \)
- \( \text{value-space(RealNumber)} \equiv \mathbb{Z} \)
- \( \text{value-space(AbstractString)} \equiv \text{value-space(String)} \)

- Actually, `anyURI`, `NOTATION` and `QNAME` are just strings with a specific restriction and meaning. As such, they should be subclasses of `String`, which will make the `AbstractString` class unnecessary. However, in order to
maintain compatibility with XSD datatypes (and thus with reasoners that support XSD datatypes), this class is added.

- **value-space(Binary) ≡ any binary object**

  The other abstract datatypes do not have well defined value-spaces. Although they are used to group together datatypes, they are not used to allow equality and conversion between the datatypes.

  The value-space of custom datatypes is the subset of the value-space of their superclass, as restricted by their predicate-restrictions. For custom datatypes which are immediate subclasses of an abstract datatype, the value-space needs to be defined by the user.

(ii) **Comparability and Equality**

**Definition:** a simple datatype is one of the following:

- AbstractString
- Number
- Binary
- Boolean
- Date
- Time
- DateTime
- GregorianYear
- GregorianYearMonth
- GregorianMonth
- GregorianDayMonth
- GregorianMonthDay
- Duration

**Definition:** Two datatype classes are comparable if both are derived from some common simple datatype. Formally, A and B are comparable if there exists a simple datatype C such that $A \subseteq C$ and $B \subseteq C$.

Theoretically, since each simple type has a well-defined value space, every pair of comparable classes has a common value-space, and thus can be compared and checked for equality. In practice, comparisons between several comparable datatypes
will require an intermediate conversion step. For example, this occurs when comparing a Double (or any subclass) and an Integer (or any subclass). Since their actual representation is different, a conversion of the integer into a double is needed before comparison can begin. Similarly, a conversion is needed when comparing Double against Float and Float against Integer. Since such representation conversions are typically imprecise (e.g., 1 is sometimes represented in Float as 1.000000001), a certain implementation-defined error threshold is allowed for comparison purposes.

3.3.4 Units of Measurement

Using OWL classes to represent datatypes allows one to add further information about the type and its instances. One such example, which is part of the Datatype Objects ontology, is adding units of measurement to numbers. To do this, we add to the ontology a class called MeasurementUnit, instances of which will represent the different units. Of course, the units can be further organized using a class hierarchy. Any class derived from Number can then be defined to have the appropriate units using the property units. For example, the class LengthInCMs can be defined to have the unit cm, an instance of DistanceUnit, which in turn is a subclass of MeasurementUnit. In Description Logics, this is written:

\[
\text{LengthInCMs} \equiv \text{Length} \cap \text{units} \in \text{cm}
\]

Where:

\[
\text{Length} \sqsubseteq \text{Double} \sqcap \forall \text{units}. \text{DistanceUnit}
\]

\[
\text{typeof(cm) = LengthUnit}
\]

\[
\text{LengthUnit} \sqsubseteq \text{MeasurementUnit}
\]

Note that the direct usage in the domain ontology of datatypes which are associated only with a class of measurement units but not with a specific unit (e.g., usage of Length in the ontology) is prohibited.

Specifying the units of measurement of a datatype adds to its semantics by refining its comparability.

Definition: Two datatypes A and B that have specific measurement units UA and UB are comparable only if UA \(\equiv UB\).
3.3.5 Reasoning with DatatypeObjects

As stated before, since ontologies using DatatypeObjects are perfectly legal OWL-DL ontologies, they can be reasoned over by any DL reasoner. However, in order to support reasoning about complex datatypes that are created with predicates, a new type of reasoning is needed. Generally, when coming to check if $A \sqsubseteq B$, we need to check, in addition to the normal DL subsumption, if the predicates that participate in the description of $A$ are specifying more specific (or at least not less-specific) constraints than those that participate in the description of $B$. Clearly, only predicate restrictions of the same type are comparable – and even among these, checking subsumption is not always easy.

For example, it is very easy to construct a mechanism that checks subsumption between LessThan predicates: if class $A$ has the expression LessThanRestriction $\sqcap (\exists$ predicateParameter.Age_30) in its description and class $B$ has the expression LessThanRestriction $\sqcap (\exists$ predicateParameter.Age_60), we can clearly say that $A \sqsubseteq B$, since every age which is $< 30$ is certainly also $< 60$. On the other hand, providing such an oracle for the StringPatternRestriction, which restricts values using regular-expression syntax, is difficult. For each new predicate, a new subsumption check needs to be implemented. In addition, the various Boolean combinations of the restrictions need to be considered. The implementation of this is reserved for future work.

3.3.6 Using Existing Ontologies

In order to allow the usage in HyperFlow of existing ontologies (which make use of normal OWL datatypes and DatatypeProperties), a way is needed to translate them to the new schema. To this end, I implemented a ModelTranslator program, whose algorithm is given in Appendix 1. This program replaces each datatype property with an object property, and sets its range to be a new a datatype class. Such new classes are generated for each property-range pair. Thus, for example, if a NucleotideSequence object has two datatype properties: sequence and description, whose ranges are xsd: String, the translation will create two disjoint subclasses of StringObject: Sequence-String and Description-String. Furthermore, the
property hierarchy is mirrored in the datatype class hierarchy. In addition, the algorithm translates enumerations, restrictions and instances to the new schema.

Of course, after the translation is done, users can modify the resulting ontology as they please, directly or with any tool such as Protégé’s OWL Plugin [64]. They can then further enhance the expressiveness of the datatypes, using the constructs described above, beyond what is supported in their original OWL documents.

3.4 Collections

Unlike the ODMG object model [65], OWL does not have built-in collections. However, it is rather easy to introduce them as normal OWL classes – an approach first suggested for List in the OWL-S ontology [3]. Furthermore, the $\forall$ local class restriction facilitates defining subclasses for collections of specific classes, e.g. creating a List of Genes.

I defined a collections ontology which contains the base collection classes: Collection, List, Set, Bag, and Map. Using these collection classes we are able to represent in our model any object-oriented database and any collection of items, be it class individuals or datatype object individuals.

3.5 User-Defined Classes

When posing queries in HyperFlow, users may select an arbitrary projection as the result. This projection may include ontology objects, values of data type properties, results of arithmetic functions, results of query operators such as group-by, and results of nested queries. Each such projection defines a new user-defined class, and the query result would be a collection of instances of this class. The user-defined class can then be used in the same ways that ontology classes are used – i.e. in subsequent queries, services and workflows.
Chapter 4.

The HyperFlow Language

This chapter reviews the various constructs of the HyperFlow language [1]. We start by describing the visual representation of the data model. Then, we examine the various dataflow constructs in Section 4.2. In Section 4.3 we explain the novel paradigm introduced in HyperFlow’s query language and review each query construct. Finally, in Section 4.4.5 we conclude with a comparison with previous work.

4.1 Visual Representation of Ontology Entities

Table 1 shows the visual representation of some of the basic ontology entities: classes, instances, and datatypes. Note that we differentiate between ontology classes and view (i.e. arbitrary tuple) classes, and between by-reference and by-value semantics, where applicable. This is useful in queries, where selecting an object by reference (see e.g. Figure 10) will yield – rather than the whole object – only its ID (URI, OID or primary key), sometimes accompanied by some textual description field. This allows us to support a usage strategy common in information retrieval over the World Wide Web, in which the user first searches for IDs and later fetches the complete objects if needed.

<table>
<thead>
<tr>
<th>Table 2. Entity icons in HyperFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>By Reference</strong></td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Instance</td>
</tr>
<tr>
<td>Enumeration of instances</td>
</tr>
<tr>
<td>User-defined class</td>
</tr>
<tr>
<td>Numeric datatype</td>
</tr>
<tr>
<td>String datatype</td>
</tr>
<tr>
<td>Boolean</td>
</tr>
<tr>
<td>Date/Time</td>
</tr>
<tr>
<td>Enumeration of numbers</td>
</tr>
<tr>
<td>Enumeration of strings</td>
</tr>
<tr>
<td>null</td>
</tr>
</tbody>
</table>
Properties of an entity are depicted using property edges, which are labeled (and blue). Constraint nodes, such as in Figure 10, are connected to entities by (black) constraint edges. Finally, dataflow edges are dashed (and red).

Collections are represented, as is common in many visual query languages (VQLs), as rounded rectangles ("blobs") surrounding an entity icon which represents the collection's type (see Figure 5). In case a property of the entity is shown, that property edge will leave the collection (see Figure 23). While this design decision admittedly prevents the user from seeing explicitly the nature of the collection's nesting, it was taken in order to prevent the unmanageable and unintelligible nesting of collection blobs.

![Figure 5. Collections in HyperFlow](image)

**4.2 Data-Flow Constructs**

As in all dataflow languages, HyperFlow analysis workflows consist of a network of modules, connected by dataflow edges (depicted as dashed red lines). HyperFlow further adds the concept of result-set boxes, which represent database views. Result-sets depict the type of information that results from each module and allows language features such as implicit iterations. In this section, we review each of those in turn.

**4.2.1 Modules**

Modules are depicted as boxes that have input ports and output ports. These ports may be named and may have a type, which can be an ontology class, a primitive class, or a (possibly typed) collection class.

In HyperFlow, there are several types of modules: services, nested workflows, and queries. As we discuss in Section 4.3 some query operators, such as group-by, order by, arithmetic functions and aggregate functions, are also depicted as modules inside the query module. We treat them here as services since they can also be used as stand-alone services outside the query.
A service is depicted in the black-box fashion familiar from other visual dataflow languages: a box with the service name and possibly an icon. Workflows and queries can also be shown as black-boxes, which is useful for their reuse. In addition, HyperFlow provides a glass-box depiction of workflows and queries (such as the query in the example, Figure 1), which allows them to be shown and edited in the same diagram as the outer process. The glass-box depiction of a sub-workflow can be used to introduce the notion of scope to the workflow, in particular for iterations, as discussed below.

The input ports of services are predefined and dependent upon a given service profile (such as a WSDL [66] or OWL-S [3] document). In contrast, workflows and queries are user-defined; users add ports to workflows and queries while editing them.

Some services have many parameters, and usually only few of those are relevant to the dataflow. For example, the BLAST [45] service takes in addition to the input sequence (which would usually come from the dataflow) several arguments such as: e-value, word-size, matrix name, and gap cost. Normally users would not want to specify these arguments in the workflow. Rather, they would prefer a properties window, in which they can also accept some suggested defaults. However, there might be cases in which the user wants to give those arguments values from the flow. To support this, HyperFlow enables hiding and exposing input ports. Each input port can be hidden by default and then exposed by the user, and vice versa.

4.2.2 Result Sets

In HyperFlow, each module is immediately followed by a result set box – a yellow box with a dashed outline. This box represents the database view that is (at least conceptually) created to hold the results returned from the module. A result set can store a single element or a collection of elements – which can be ontology objects, datatype objects, tuple objects (i.e. defined by the user in a query), and other collections. A notable exception is the result set of a construct-graph query (see Section 4.3.10) which returns a graph of objects, and requires a database that supports this function (i.e. an RDF database).

The type of objects in the result set is depicted inside the result-set box. In the case of tuple objects, the fields of the tuple may also be shown. In the implementation, we show the result-set type while a query is being built. This lets the user have
immediate visual feedback that can help constructing the query. Another form of feedback is the collection size display, which is added to collections in interactive exploration mode after the results arrive (see Figure 6). Finally, it should be noted that like views in a database, result sets can be named, and this name is shown as a title on top of the result-set box.

Results sets serve as more than visual feedback. First, HyperFlow allows the user to expand (i.e. show) properties of the objects in the box. This provides the user fine control over what is passed to the next module: a user can connect the node representing an object's field, instead of connecting the whole object. Another feature is the ability to conveniently send to the next module either the whole collection (see Figure 1d and e) or the collection's members one by one in an implicit iteration.

4.2.3 Implicit Iteration

When the node that describes the type of members in a collection is connected to the input port of a module, this module is executed once for each collection member. This is called implicit iteration, and is demonstrated in Figure 1c. The result of the execution will be the collection of results from each execution, maintaining the order of the input collection (if it was ordered). In cases where each execution yields a collection, there is no automatic merging – the result will be a collection of collections.

Implicit iterations are more than just a depiction of the simple for each loop. They can also be used to go over multiple collections if the receiving module has more than one input port. In these cases, the result will be an execution of the module for each combination that comprises one item from each collection (see Figure 6a). To further control the iteration, HyperFlow provides several module execution modifiers for controlling the iteration: For instance, when the input collections are ordered lists of the same length, the "i=j" modifier makes it possible to iterate over ordered pairs of list elements (see Figure 6b). Other modifiers (see Figure 7) allow iterating over various combinations of items from a single collection.
HyperFlow also provides input passing modifiers. A module that is marked by such modifier (depicted as a yellow output port) will return a tuple object consisting of the input and the output. If there is more than one input, there are two options: In the first (yellow output port – see Figure 8a) the inputs are shown as different fields of the tuple object. In the second (purple output port– see Figure 8b) the inputs are combined to a tuple of their own, and this tuple, with the output, forms the result object.
4.3 HyperFlow Queries

HyperFlow introduces a combination of declarative graph-based constructs with functional dataflow constructs. The declarative part allows users to specify the ontology subgraph with which the query deals, and add some constraints over the various object and datatype nodes. This is equivalent to the FROM and WHERE parts of an SQL/OQL query, as well as the use of path expressions in OQL. The other parts of a query – SELECT, GROUP-BY, ORDER-BY, functions, etc – are depicted using data-flow constructs, i.e. by using data-flow edges to send the selected schema nodes to the output port, while passing through query operation modules on the way (see e.g. Figure 19). Notice how this distinction nicely mirrors the linguistic distinction between the names of various SQL/OQL constructs; only the constructs which are written in an imperative form (i.e. as commands) are depicted as processing modules.

This section describes HyperFlow's query constructs. For each construct we give an example and describe its syntax and semantics. This description draws heavily upon the description of the closest textual language, OQL (see [65, Section 4.10]).

HyperFlow is an expression language. A query expression is built from typed operands composed recursively by operators. An expression returns a result that can be an object or a datatype literal. In HyperFlow, the type of the results of a query is defined by the dataflow edges that connect entity nodes to the query's output port. This mechanism defines the projection of the query.

HyperFlow is a strongly typed language. Thus, each query expression has a type. This type can be derived from the structure of the query expression, the ontology type declarations, and the type of the objects and literals. For each query expression, we give the rules that allow to (1) check for type correctness, (2) deduce the type of the expression from the types of the sub-expressions, and (3) deduce the result (value) of the expression from the results (values) of the sub-expressions.

4.3.1 Query Constraints

Constraints are shown in HyperFlow as orange hexagons containing the constraint symbol or word. They are connected to the schema node by means of unlabeled black edges. Examples of constraint usage are shown in Figure 9 and Figure 10. HyperFlow supports a natural usage of a wide variety of constraints, including constraints such as OQL’s collection inclusion constraint and SQL.99’s type checking constraint. The
latter, and its sibling \( <\text{ATP citrate lyase}>\), are very useful when hierarchical controlled vocabularies are used in the ontology – such as the organism taxonomy hierarchy in Figure 1 and the Gene Ontology's (GO) Molecular Function hierarchy, in Figure 9.

Every constraint can be seen as a function whose return type is Boolean. If a function with such a signature exists in the databases, HyperFlow supports using it as a custom constraint in queries. One example of this is (j) in Figure 9. Other notable examples are some of the methods common in Geographic Information System (GIS) databases, such as Contains() and Intersects().

\[(a) \quad \text{Gene} \rightarrow \text{name} \rightarrow S \rightarrow = \rightarrow S \rightarrow \text{ATP citrate lyase}
\]

\[(b) \quad \text{Gene} \rightarrow = \rightarrow 1 \rightarrow \text{hgnc:ACLY}
\]

\[(c) \quad \text{Gene} \rightarrow \text{start} \rightarrow \# \rightarrow S \rightarrow 1000
\]

\[(d) \quad \text{Gene} \rightarrow \text{start} \rightarrow \# \rightarrow \text{between} \rightarrow S \rightarrow 1000 \rightarrow S \rightarrow 5000
\]

\[(e) \quad \text{Gene} \rightarrow \text{name} \rightarrow S \rightarrow \text{in} \rightarrow S
\]

\[(f) \quad \text{Gene} \rightarrow \text{aliases} \rightarrow S \rightarrow \text{in} \rightarrow S \rightarrow \text{HsMAD1}
\]

\[(g) \quad \text{Gene} \rightarrow \text{aliases} \rightarrow S \rightarrow \subseteq \rightarrow S
\]

\[(h) \quad \text{Gene} \rightarrow \text{in} \rightarrow 1
\]

\[(i) \quad \text{Gene} \rightarrow \text{known functions} \rightarrow \text{Molecular Function (GO)} \rightarrow \text{of type} \rightarrow S \rightarrow \text{Transport}
\]

\[(j) \quad \text{Gene} \rightarrow \text{is homologous} \rightarrow S \rightarrow \text{Gene}
\]

Figure 9. Usage modes of various constraints

(a) The gene's name is "ATP Citrate lyase".
(b) The gene is identified by the HGNC (human gene nomenclature committee) symbol "ACLY".
(c) The gene starts before or at position 1000 of its chromosome.
(d) The gene starts between position 1000 and position 5000 of its chromosome.
(e) The gene's name is one of the strings in the given list of strings (list items are not shown).
(f) One of the aliases of the gene is the string "HsMAD1".
(g) The aliases of the gene are a subset of the given string collection.
(h) The gene is one of the genes in the given instance collection.
(i) One of the known molecular functions of the gene is Transport or a subclass of it, as described in the Gene Ontology (GO) controlled vocabulary.
(j) The gene is homologous (according to the execution of the "is homologous" database function / method) to another gene, which can be further constrained.
Syntax and Semantics: A query constraint defines a Boolean-typed expression which serves as a selection filter for database objects or literals. Each constraint is connected via an incoming connection to one entity or Boolean operator (lhs) and to at least one entity node (rhs) through an outgoing connection.

- if lhs is an entity node, let LHS = type(lhs)
- if lhs is a Boolean operator node, let LHS = type(lhs's incoming entity)
- let RHS = type(rhs)

In the description that follows, we make use of the following definitions:

1. Typed collection:
   \[\text{Collection}<C> \equiv \text{Collection} \cap \forall \text{members}.C\]

2. Comparable datatypes: see Section 3.3.3

3. Comparable classes:
   A and B are comparable if either:
   - LHS ⊑ RHS  or
   - LHS ⊒ RHS

We are now ready to describe the syntax and semantics of specific constraints:

a. \(=\), \(\neq\) (for datatypes):
   - The expression is legal if LHS, RHS are comparable datatypes
   - Result semantics:
     - For \(=\), true if the value of lhs equals to that of rhs, otherwise false.
     - For \(\neq\), false if the value of lhs equals to that of rhs, otherwise true.

b. \(=\), \(\neq\) (objects):
   - The expression is legal if LHS, RHS are comparable classes
   - Result semantics:
- For \( = \), true if lhs and rhs refer to the same object, otherwise false.
- For \( \neq \), false if lhs and rhs refer to the same object, otherwise true.

c. \( \approx \), \( ! \approx \) (object):
- The expression is legal if LHS, RHS are comparable classes
- Result semantics:
  - For \( = \), true if the value of lhs equals to that of rhs, otherwise false.
  - For \( \neq \), false if the value of lhs equals to that of rhs, otherwise true.

d. \( <, >, \leq, \geq \):
- The expression is legal if LHS, RHS \( \sqsubseteq \) Number and they are comparable
- Result semantics: true if the appropriate numeric relationship between \( lhs \) and \( rhs \) holds.

e. \( \text{in} \):
- The expression is legal if RHS \( \sqsubseteq \) Collection<\( L \)> and L and LHS are comparable.
- Result semantics: true if \( lhs \) is included in the collection \( rhs \).

f. \( \subset, \supset \):
- The expression is legal if:
  - LHS \( \sqsubseteq \) Collection<\( C \)>
  - RHS \( \sqsubseteq \) Collection<\( D \)>
  - C and D are comparable
- Result semantics:
  - For \( \subset \), true if the collection \( lhs \) is a subset of the collection \( rhs \).
  - For \( \supset \), true if the collection \( lhs \) is a superset of the collection \( rhs \).

g. "of-type", "of-exact-type":
- The expression is legal if LHS \( \not\sqsubseteq \) RHS.
- Result semantics:
  - For "of-type": true if the actual run-time type of \( lhs \) \( \not\sqsubseteq \) RHS
  - For "of-exact-type": true if the actual run-time type of \( lhs \) is RHS.

h. \( \text{like} \):
- The expression is legal if LHS, RHS \( \sqsubseteq \) String.
- Result semantics: true if \( lhs \) conforms to the pattern defined by \( rhs \).
i. between:

This constraint is connected to two nodes through outgoing edges: \textit{lower-rhs, higher-rhs}. Let \texttt{LOWER-RHS}=\texttt{type(lower-rhs)}, \texttt{HIGHER-RHS}=\texttt{type(higher-rhs)}.

- The expression is legal if \texttt{LHS}, \texttt{UPPER-RHS}, \texttt{LOWER-RHS} \sqsubseteq \texttt{Number} and are comparable.
- Result semantics: \textit{lower-rhs} $\leq$ \texttt{lhs} $\leq$ \textit{higher-rhs}

j. custom functions - any function that has a Boolean result can be represented as a constraint

- Connected nodes: according to the function's inputs.
- Result semantics: true if the function's result is true.

### 4.3.2 And, Or and Not

HyperFlow allows combining constraints into Boolean expressions. This is done by using the \texttt{AND}, \texttt{OR}, and \texttt{NOT} operators, depicted by (blue) \textit{modifier nodes}, to draw arbitrarily complex expression graphs. The various ways to use the operators are shown in Figure 11. Note that the AND modifier is usually superfluous, as AND is the default operator for combining constraints in a query. Also note the two different usage patterns of \texttt{OR}.

![Figure 11. AND, OR and NOT](image)

(a) Select proteins whose lengths are either larger than 3000 (amino acids) or less than a thousand.

(b) Select proteins whose name is not like 'kinase' and either their function is 'Hormone Receptor' or their function is 'Transport' and their length is larger than 3000.
OQL supports special versions of `and` and `or` – namely `andthen` and `orelse`. These two operators enable conditional evaluation of their second operand, and also dictate that the first operand be evaluated first. In HyperFlow, these are represented by adding execution order numbers near property and constraint edges.

Figure 12. Specifying execution order
Select genes which are starting in coordinate larger than 3000 on some chromosome, and are homologous to the CFTR gene. Since homology checking entails running a service, for performance reasons, the second condition should be run only if the first one is correct.

Syntax and Semantics:

a. OR, AND: These two constructs allow creating compound Boolean expressions from two or more Boolean expressions.

- The expression is legal if it is connected to:
  - Incoming: using a modifier edge to `lhs` – an entity node, `type(lhs)=LHS`
  - Outgoing: through `rhs_1, rhs_2,...,rhs_n` – each can be either:
    - A constraint node that applies to `lhs` (using a modifier edge)
    - A Boolean node
    - An entity node which represents a property of `lhs`.
      In this case, `rhs_i` is connected to the operator by a property edge.

- Result semantics:
  - OR: \( \bigvee_{i=1,n} result(rhs_i) \)
  - AND: \( \bigwedge_{i=1,n} result(rhs_i) \)

b. NOT: Given a Boolean query expression, this operator generates an expression with the opposite semantics.

- The expression is legal if it is connected to:
  - Incoming: `lhs`, an entity node type `(lhs) = LHS`
- Outgoing: $rhs$ – which is either:
  - A constraint node
  - A Boolean node
  - An entity node which is the $lhs$ of a Boolean expression. The entity is connected to the operator by a property edge.
- Result semantics: $\neg$Result($rhs$)

### 4.3.3 Joins

A join is created in HyperFlow by connecting two schema nodes (that do not represent fields of the same object) using a constraint (Figure 13a). Equi-joins can be created without using a constraint, as is demonstrated in Figure 13b. In addition, a Cartesian product is created whenever the nodes are not connected, as in Figure 13c.

In order to allow naming specific fields of the resulting output class, HyperFlow provides a specialized module, the AS module that corresponds to the SQL aliasing construct AS, which can be placed on the outgoing dataflow edge (see Figure 13a).

Figure 13. Join examples
(a) Select pairs of genes where one completely overlaps the other.
(b) Select pairs of molecules that have the same molecular weight. (equijoin)
(c) Select all pairs of genes. (cross product)
Syntax and Semantics: The syntax is as explained in the previous page. The result of a join is a bag of elements of type \( (X_1, X_2, \ldots, X_n) \), where each \( X_i \) is the type of the corresponding participant ontology node \( x_i \). This bag is populated with a Cartesian product, filtered by the associated constraints, if there are any.

4.3.4 Modifiers for All, Some, Transitivity, and Optionally

*All* and *some* are the equivalents to OQL’s 'ALL' and 'EXISTS', respectively. These modifiers can be placed on the property edge headed *towards* the collection or on the constraint edge directed *away* from it. The first case (see Figure 14a) is analogous to OQL’s "for all x in" and "exists x in" constructs, and to SQL and OQL’s left-hand-side ALL and EXISTS. The second case (see Figure 14b) is analogous to SQL and OQL’s right-hand-side ALL and EXISTS.

*Transitively* is used to transitively follow properties.

![Figure 14. All, transitively](image)

(a) All of the known functions of the gene are of type Transport.
(b) The molecule’s molecular weight is higher than all the numbers in the given list.
(c) The protein is located in the 'cytoplasmic vesicle' – either directly or in any other cellular component that is located inside it.

The *optionally* modifier is used to define outer joins:

![Figure 15. Outer joins using optionally](image)

(a) Get all pairs consisting of a protein, and, when available, a fatty acid that has the same molecular weight (left outer join).
(b) Get all pairs consisting of either a protein, a fatty-acid, or when both have the same molecular weight – both (full outer join).
Syntax and Semantics:

1. **ALL**: When placed legally in a query expression, ALL is always connected by an outgoing edge to a collection. It then defines a Boolean expression whose result is true if the query expression holds for all \( x \) in the collection. Formally:
   a. Left-hand-side form:
      - The expression is legal if it is connected to:
        - Incoming: \( lhs \) – an entity node
        - Outgoing: \( rhs \) – a collection, whose element node is the \( lhs \) of the expression \( expr \).
      - Result semantics: true if for each \( x \in rhs, expr(x)=true \)
   b. Right-hand-side form:
      - The expression is legal if it is connected to:
        - Incoming: \( lhs \) – a query constraint node which is a part of the expression \( expr \).
        - Outgoing: \( rhs \) – a collection.
      - Result semantics: true if for each \( x \in rhs, expr(x)=true \)

2. **SOME**: When placed legally in a query expression, SOME is always connected by an outgoing edge to a collection. It then defines a Boolean expression whose result is true if the query expression holds for some \( x \) in the collection. Formally:
   a. Left-hand-side form:
      - The expression is legal if it is connected to:
        - Incoming: \( lhs \) – an entity node
        - Outgoing: \( rhs \) – a collection, whose element node of is the \( lhs \) of the expression \( expr \).
      - Result semantics: true if there exists \( x \in rhs \) such that \( expr(x)=true \)
   b. Right-hand-side form:
      - Connected nodes:
        - Incoming: \( lhs \) – a query constraint node which is part of the expression \( expr \)
        - Outgoing: \( rhs \) – a collection
      - Result semantics: there exists \( x \in rhs \) such that \( expr(x)=true \)
3. **Transitively:** When placed legally in a query expression, *transitively* has an outgoing property edge which represents a transitive property. It then causes the expression to evaluate on the transitive closure of that property, and return true if the expression holds for one of the elements of this closure.

- The expression is legal if it is connected to:
  - Incoming: \( lhs \) – an entity node
  - Outgoing: \( rhs \) – an entity node, connected by a property edge which represents a property \( p \) of \( lhs \) which is transitive.
- Result semantics: true if there exists a value \( x \) for \( rhs \) such that \( x \) is in the transitive closure of the property \( p \) for \( lhs \) and \( \text{expr}(x) = \text{true} \).

4. **Optionally:** When placed legally in a query join expression, *optionally* causes the result to include also those \( rhs \) objects which do not conform to the join condition.

- The expression is legal if it is connected to:
  - Incoming: \( lhs \) – an entity node, which serves as the \( lhs \) of the join's condition expression.
  - Outgoing: \( rhs \) – a constraint node, which serves as the constraint in the join's condition expression.
- Result semantics: causes the join to include objects that contain \( rhs \) values which do not pass the condition. In these cases, the value of rest of the result fields will be *null*.

### 4.3.5 Down Casting

To go down the class hierarchy, a user may explicitly *convert* the class *down to* the needed subclass:

![Diagram showing the 'Nucleotide Sequence' class hierarchy and a down-cast operation](image)

Figure 16. Down casting

From the input sequences, get only those whose molecule type is "trna". Since the "molecule type" property is available only to nucleotide sequences, a down-cast is needed before applying the constraint.
Syntax and Semantics:

• Input: any element x.
• Argument: any ontology class C, such that C ⊑ declared-type(x).
• Output: the element x as an instance of type C (or an exception if this x's dynamic type is not C and not a subclass of C).

4.3.6 Selecting Groups

HyperFlow has some built-in support for some of the queries which are natural to ask, but usually difficult to phrase in an OQL query. For example, a user may ask to get all the genes that fit some criteria, fetching along with each gene ID also its collection of transcript RNA sequences. The way to do so in HyperFlow is to connect the required collection directly to the output port (see Figure 17) – this saves the need to write a nested query, as one would need to do in OQL.

Another construct that saves the need to write a nested query when grouping is the collect operator. This operator is used to collect, from a collection of instances of some class, the values of some property of that class. For example, in Figure 18, the names of the RNA sequences in the "mRNA transcripts" collection are collected into a collection which is then chosen in the output projection. Again, in OQL and SQL this is done using a nested query, which we make superfluous here.
Figure 18. Collect
Get each gene and the names of its transcript RNA sequences

Syntax and Semantics:
- Input: any element x, type(x)=X, which is connected via an incoming property edge to a collection-type node y, type(y)=Y, and Y has a property P with range X.
- Output: Let C be the collection of which y is the type node. C = \{y_1, y_2, ... y_n\}. Then the output of this query operation is \{ x_i | x_i = p(y_i) \}.

4.3.7 Grouping

The HyperFlow analog of OQL’s GROUP BY statement is the Group A by B module. As its name suggests, it aggregates the results from the previous steps (that enter through port A) into groups, based on the value of the object or field that is connected to port B. The results are tuples, each consisting of the values of the aggregation criteria and the collection of objects that fit these values (partition, in OQL terms). The aggregated objects themselves can be of any type, including arbitrary projections, thus possibly creating a tree of objects.

Figure 19. Group by
Get, for each molecular function instance, all the genes that are known to perform it.
Syntax and Semantics:

- **Inputs:**
  - A - one or more entity nodes.
  - B, C, etc. – each gets one entity node which serves as a *partition attribute*.
- **Output:** The Cartesian product visited by the query is split into partitions. For each element of the Cartesian product, the partition attributes are evaluated. All elements that match the same values according to the given partition attributes belong to the same partition. Thus, the partitioned set, after the grouping operators, is a set of instances of the following user-defined class: Each structure has the valued properties for this partition (the partition attributes), completed by a property that is conventionally called *partition*, and that is the bag of all elements of the Cartesian product matching this particular valued partition.

HyperFlow makes it easy to filter out some of the aggregated groups based on an aggregate function (*i.e.* to use the HAVING construct). In order to do this, the user instructs the system to show a result set *inside the query*, after the group-by module. The results displayed can then be connected via dataflow to any aggregate function module, whose result can again be selected, constrained, or made to flow to other modules.

Get, for each molecular function instance that is performed by more than 10 genes, all the genes that are known to perform it.

---

**Figure 20. Group by and having**

Get, for each molecular function instance that is performed by more than 10 genes, all the genes that are known to perform it.
**4.3.8 Ordering**

ORDER BY is depicted in HyperFlow by an "Order By" module. This module has two input ports – one for the input objects and one for the sort criterion. In order to sort by more than one criterion, extra criteria input ports can be added (see Figure 21). In addition, for each port the sort order (increasing or decreasing) can be ascribed.

![Figure 21. Order by](image)

Order the proteins by length and then by decreasing order of their isoelectric point (pI).

**Syntax and Semantics:**

- **Input:**
  - A - one or more entity nodes.
  - B, C, etc. – each gets one entity node which is the result of an order expression, and has a specified order modifier, which is either ↑ - ascending (default), or ↓ - descending.

- **Output:** After building the Cartesian product defined in the query, filtering the results according to the constraint expressions, and grouping if needed, ordering is performed. This query function returns the selected elements in a list, sorted in a lexicographic order according to the order expressions.
4.3.9 Removing Duplications

Another SQL construct which is depicted as a module is DISTINCT. We call the module "remove duplicates" to better describe it as a service (see Figure 22).

![Diagram of Protein: length and remove duplicates](image)

Figure 22. Removing duplicates from the result

**Syntax and Semantics:**
- Input: one or more element nodes, which represent the query.
- Output: This query operation causes the elimination of duplicates from the query output, resulting in a set without duplicates.

4.3.10 Construct and Construct Graph

HyperFlow provides two modules that help users organize their output: The construct module is an implementation of the "struct" (see Figure 23) and "new" (see Figure 24) constructors in OQL. It combines all its inputs as fields in a single output class. The resultant object can be named, just as in the AS module.
Figure 23. Creating new result objects using construct

Syntax and Semantics:
- Input: any two or more elements \((x_1, x_2, ..., x_n)\).
- Argument: a name of the user-defined class (optionally).
- Output: an instance or the user-defined class whose properties are \((x_1, x_2, ..., x_n)\).

Figure 24. Creating new ontology objects using construct
Syntax and Semantics:
- Input: each port $X_i$ takes one element $x_i$ according to the properties of the argument class.
- Argument: a name of an existing ontology class.
- Output: an instance or the ontology class whose property values are $(x_1, x_2, ..., x_n)$.

The second module is the construct graph module (see Figure 25). This module is used as an editor in which a new graph that is made of the various inputs can be created. In the new graph, the objects can be connected using any connections, including ones that are newly-defined. This allows to use the newly proposed construct queries in the forthcoming RDF query language SPARQL [67].

Figure 25. Constructing a new graph

Syntax and Semantics:
- Input: any two or more element nodes $(x_1, x_2, ..., x_n)$.
- Argument: a description of a graph that contains the input nodes, connected using either ontology properties or through new properties.
- Output: This query operation causes the creation of a new view that contains, for each combination $(x_1, x_2, ..., x_n)$ in the input, the sub-graph defined in the argument.
4.3.11 Functions and Methods.

Using the dataflow metaphor gives HyperFlow a natural method of depicting functions as modules. This applies to built-in arithmetic and string functions, as well as to aggregate functions and conversions (listtoset, flatten, element), set operators, and custom database functions. In order to clarify set operator semantics, the modules that depict them show icons with the respective Venn diagrams. Finally, we allow all query constraints, as well as Boolean operators, to be used as functions with a Boolean return value (see Figure 27).

Figure 26. Collection and set operators.

Figure 27. Boolean constraints as functions
Find for each gene in the input if it is located on the positive strand.
When coming to support object-oriented methods, we make a distinction between two types of methods, which we support in two different techniques. The first type contains methods with no argument. Since those methods depend solely on the object, we treat them as if they were ordinary properties, as is common in some object-oriented languages (e.g. properties in C#). We treat all other methods as what methods really are – functions that get as input, in addition to their stated arguments, also the object on which they operate.

**Syntax and Semantics:**

1. **Set operations:**
   a. intersect, union:
      - **Input:** $A_1, A_2, \ldots, A_n$, where each $A_i$ is a collection
      - **Output:** $\cap A_i, \cup A_i$, respectively
   b. difference ($A/B$)
      - **Input:** $A, B$ – collections
      - **Output:** $A/B$

2. **Collection operations:**
   a. first, last:
      - **Input:** $\text{List}<C>
      - **Output:** returns the first/last element of the input list.
   b. $[i]$: getting the $i$-th element of a list:
      - **Input:** $\text{List}<C>$
      - **Argument:** a non-negative integer $i$, $i<\text{length of the input list}$.
      - **Output:** returns the element from the input list whose position is $i$.

       Note that the first element is in position 0.
   c. $[i:j]$: extract a sub-list from a list:
      - **Input:** $\text{List}<C>$
      - **Arguments:** two positive integers: $i, j$
      - **Output:** returns the $\text{List}<C>$ that starts with position $i$ and ends at position $j$.
   d. element: retrieving the element of a singleton:
      - **Input:** $\text{List}<C>$ which contains a single element.
      - **Output:** returns the single element.
e. +: concatenating two lists:
   - Input: A- List<C₁>, and B- List<C₂>
   - Output: returns the concatenated list. This will be a List<mss(C₁,C₂)>,
     where mss(X,Y) is the most specific superclass of classes X and Y.

f. flatten:
   - Input: Collection<Collection<C>>
   - Output: returns a Collection<C> populated with the elements of the
     collections in the original input.

g. to-set:
   - Input: List<C>
   - Output: returns a Set<C> populated with the elements of the input list.
     Loses the order and eliminates duplicates.

   a. Input:
      - A = $a$, type($a$) ⊑ Number
      - B = $b$, type($b$) ⊑ Number
      - Output: $a+b$, $a-b$, $a*b$, $a/b$, respectively

4. Boolean constraint operations: see semantics of constraints (Section 4.3.1).

5. Custom functions: according to their definitions.

4.3.12 Undirected Edges

Another feature that increases the expressive power of HyperFlow queries is the
use of undirected property edges – a feature which is borrowed from the QGraph [68]
visual query language. These edges are useful when both nodes can have the property
and the user does not care which of them will actually have it. It is applicable first and
foremost for relationships where both sides are of the same class, but not solely to
these cases, since in OWL several classes can have the same property. This feature is
easily translated to a normal query that uses disjunction or even union of two
subqueries.

Syntax and Semantics: Given two entity nodes cls and rhs, where type(cls)=C and
type(rhs)=R. A unidirectional edge of property $P$ is legal if both $C$ and $R$ has the
property $P$. 
Semantics: the following constraint is added to the query: $rhs=P(lhs) \lor lhs=P(rhs)$.

4.3.13 Subqueries

A nested subquery is depicted in HyperFlow as a query module nested inside the main query module (Figure 28). The results from the inner query are shown in the outer query, and can be related to specific nodes using constraints and modifiers. To create a correlated subquery, an input port is added to the subquery (Figure 29). It accepts input from a node in the outer query, thus making explicit the fact that the inner query is run once for each outside node instance.

Figure 28. Uncorrelated nested query

Return just those proteins whose length is greater than all the lengths of the proteins whose names end with "kinase".

Figure 29. Correlated nested query

Get the genes whose names end with "kinase" and where 550 is one of the lengths of their RNA transcripts.
4.3.14 Query Naming and Recursion

HyperFlow allows queries and subqueries to be named. Named queries can be stored and reused at a later occasion. Moreover, a named query can be used as a subquery nested in any query, including itself in a recursive fashion. An example of a recursive subquery is given in Figure 30. This type of queries can be translated to SQL's WITH RECURSIVE statement.

Figure 30. A recursive subquery

Recursively find all regulating genes (genes that encode proteins called transcription factors, which regulate the gene by binding a site on the DNA nearby the gene).
4.4 Language Analysis

In this section, we shall analyze HyperFlow and see to what extent it fulfills several criteria that are usually used to examine query languages (taken from [69]):

Expressive Power – This property indicates how powerful queries can be. In Appendix 2, I prove that HyperFlow’s expressive power is at least as strong as the expressive power of OQL. This is done by showing a reduction from OQL’s Extended Backus Naur Form (EBNF) to HyperFlow constructs, thus proving that each legal OQL query can be depicted in HyperFlow. In addition, HyperFlow supports several features that OQL lacks, such as outer joins, recursion, transitive property traversal and graph selection. For a complete list of features, see Table 3 in the next section.

Closure – The closure property requires that the results of an operation are again elements of the data model. In HyperFlow, this criterion is fulfilled: the results of queries and operations are objects, collections of objects, or graphs of objects.

Adequacy – A query language is called adequate if it supports all the characteristics of the underlying data model. HyperFlow is adequate, as it supports all the features of the extended OWL ontology language described in Chapter 3: classes, individuals, datatypes, properties and collections are HyperFlow’s primitives. The class and datatype hierarchy is supported (e.g., by the "of type" predicate). The notion of an object’s ID (URI) is supported by having a "reference" and "value" representations of the primitives (see Section 4.1), as well as two types of equality (see Section 4.3.1). Unlike several RDF query languages (see [69]), the value-spaces of datatypes are supported instead of the lexical spaces – i.e. the values can be compared and used in calculations. Collections are also supported, including set-theory operations and predicates, as well as grouping, ordering, and collection functions.

Orthogonality – The orthogonality of a query language requires that all operations may be used independent of the usage context. HyperFlow is highly orthogonal, as this applies to all operations but collect, which can get as input only a node which is pointed to by a property edge emerging from a node that is in a collection.
4.5 Related Work and Comparison

4.5.1 Visual Query Languages

Existing visual query languages (VQLs) use a variety of paradigms, which will be reviewed in this section (for a full review see [70]). VQLs also differ in the way and the extent to which they support features such as Boolean predicate expressions, ordering results, grouping, applying aggregate functions, and using subqueries. Table 3 shows a feature-comparison between several prominent VQLs and HyperFlow. As can be seen, HyperFlow has more expressive power than any of the other languages.

Perhaps the most prominent VQL paradigm is that of graph-based queries, which use a graph representation of the relevant sub-schema, just as in the declarative part of HyperFlow. Few of the languages which follow this paradigm are Gql [71], QGraph [68], MDDQL [72], VOQL [73], and HVQS [74]. The figures below show example queries taken from the articles describing these languages. Note the different approaches taken within this paradigm: use of variables in some languages, embedding of predicates in the graph, representation of multiply-valued relationships (blobs vs. double arrow edges), etc.

Figure 31. Gql queries
(a) Find the names of suppliers whose status > 10 or are located in Athens.
(b) Find the name of suppliers whose status > 4 and are located in Paris. For each supplier also find, for each part which he ordered in a quantity of more than 900, the part names and order quantity.
Figure 32. A MDDQL query
Find the gender, date of birth, transfer information, clinical diagnosis, destination, date, and regular medication for discharged male patients born after 1.1.1950 and weighing less than 100 kg, who have reported any kind of medication as medical history, have received immediate therapy, and admitted to the hospital in condition with any kinds of symptoms and any kinds of ECG on admission.

Figure 33. A QGraph query
Find movies nominated for Best Picture in 1997 that did not win.
Another common VQL paradigm is to represent tables and classes in a tabular form. This approach was introduced by the first visual query language, Query by Example (QBE) [75]. In QBE, users select the tables to query and write the constraints in the columns of these tables. In order to join multiple tables, or even to use multiple constraints on the same table, variables must be declared (in Figure 36, _S, _Id, _A). Constructs such as projections, grouping and ordering are written (e.g. P., G., and O. respectively) in the columns. An additional "Conditions" column is used to define some constraints, such as those involving multiple columns.

(a) | Sailors | sid | sname | rating | age | Reserves | sid | bid | day |
---|-------|-----|-------|-------|-----|---------|-----|-----|-----|
    | _Id   | _P. | _S    | > 25   |     | _Id     |     |     | ‘8/24/96’ |

(b) | Sailors | sid | sname | rating | age | Conditions |
---|-------|-----|-------|-------|-----|------------|
    |       |     |       | G.P.  | _A  | AVG. _A > 30 |

Figure 36. QBE queries
(a) Names of sailors who’ve reserved a boat for 8/24/96 and are older than 25 (each table’s name is written in the first column of the table).
(b) Group sailors by their rating, and print the rating of groups where the average age is larger than 30.
QBE has influenced the graphical query facilities offered in many products, including CASE tools such as Microsoft’s Access, as well as many visual query languages. In many such languages, (e.g. VOODOO [76] and VQE [77]) the tabular form had been replaced by frames. A frame is a rectangle which contains the class' various (datatype) attributes – much like classes in a UML class diagram. Unlike in QBE, relationships between classes/tables are depicted as edges between the frames.

Figure 37. A VOODOO query
Find the name of the department whose head is Smith.

Two other VQL paradigms are highly influenced by dataflow diagrams. One is filter-flow, which is implemented in the Kaleidoquery language [78], in which objects are filtered as they go down the flow (see Figure 38); this allows for an effective and natural construction of Boolean expressions, but other constructs, such as result structuring, joins, and grouping are less intuitive.

Figure 38. A Kaleidoquery query
Find the names of companies which have employees which are older than 60 or get a salary of more than 25,000.
The other paradigm is to construct a dataflow of database operators, such as filter, project, join, order, and so on; this is done in DFQL [79] (see Figure 39), among others. This approach has the drawback that it makes no visual use of the database schema, and as such does not actually provide a way to see the query that is being built.

HyperFlow’s queries are depicted using a mixed approach that combines the graph-based and the dataflow paradigms. A somewhat similar approach was taken in QUIVER ([80],[81], see Figure 40); our proposal, however, is more expressive and – we believe – more intuitive.

There are other less-used VQL paradigms. For example, see the iconic paradigm introduced in Iconic SQL [82] (see Figure 41).
Besides its advantages in expressivity (evident from Table 3 in the next page) and intuition, HyperFlow has one more advantage over other visual query languages. The inclusion of a VQL inside a dataflow language, and the visual representation of result-sets, allow users, for the first time, to visually and easily use the mechanism of database views. Until now, views had to be defined and used explicitly, e.g. with a "save as view" command in the user-interface. In addition there was no immediately-evident flow of information between the views. With HyperFlow, users implicitly describe views, and use them in the flow, which is an easy and natural manner to use views.

Figure 41. An Iconic SQL query.
<table>
<thead>
<tr>
<th>Feature</th>
<th>HyperFlow</th>
<th>QUIVER</th>
<th>Kaleidoquery</th>
<th>Gql</th>
<th>OQgraph</th>
<th>MDDQL</th>
<th>HVQS</th>
<th>VOQL</th>
<th>VOOOO</th>
<th>VQE</th>
<th>QBE</th>
<th>DFQL</th>
<th>Iconic SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data model</td>
<td>OO (OWL-based)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>Relational</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>Relational</td>
<td>Relational</td>
</tr>
<tr>
<td>Projection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Binary Constraints</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IN (constants)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Disjunction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Negation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Relationships</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Arbitrary joins</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Outer Joins</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Existential Quantifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Universal Quantification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transitive properties</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group By</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Having</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Aggregate Functions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Binary Set Operators</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Collection Operators*</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Custom Functions / Methods</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Order By</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Subqueries</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distinct</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OF TYPE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Construct</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Construct Graph</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Output Field Aliases</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Recursion</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Closure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Notes: NV – supported in a non visual manner. P – Partially supported.

* - Collection operators – listtoset, flatten, element, etc.
4.5.2 Dataflow Languages

HyperFlow is unique in that it is not just a VQL but also a language for describing information-handling processes, such as scientific workflows. Several related visual environments have been created recently to help scientists compose complex workflows. Notable research projects are Kepler [37], Taverna [36] (part of the myGrid project), and Triana [83], and pertinent commercial products are Pipeline-Pilot [31], InforSense KDE [40] and the phased-out übertool [39]. The dataflow paradigm is also utilized for other uses, such as visualization (as in OpenDX [84]), simulations (as in Simulink [85]), and software development. The figures below show example workflows in the aforementioned languages and systems.

As noted in e.g. [37], the characteristics of scientific workflows are different from those of general purpose workflows that can be described with emerging standards such as BPEL4WS [29]. Primarily, scientific workflows are dataflow oriented, whereas business workflows (which drive the standards) are mostly control-flow oriented. Indeed, all the aforementioned scientific workflow tools use the dataflow paradigm, and HyperFlow naturally takes the same approach. HyperFlow shares many of the basic constructs with other dataflow languages, but some of its important features (such as implicit loops) are supported by only a few of them and in a manner which in our opinion is less convenient. Other features, such as native queries, are completely novel in HyperFlow.
Figure 42. A Kepler workflow (including nested workflows)

Figure 43. A Taverna workflow
Figure 44. A Triana workflow

Figure 45. A PipelinePilot workflow

Figure 46. An übertool workflow
Figure 47. A KDE (Knowledge Discovery Environment) workflow

Figure 48. An OpenDX (a.k.a. IBM Visualization Data Explorer) workflow
Chapter 5.

Implementation: The HyperFlow Environment

In order to make HyperFlow a valuable tool in the hands of end-users, an easy-to-use interface must be implemented. For this goal, I have developed the HyperFlow Environment, the main component of which is an ontology-aware graphical editor which supports the user in creating correct HyperFlow queries and workflows in an easy and intuitive way. The editor is implemented as a plug-in for the Eclipse platform [86], which was chosen for its comprehensive extensibility mechanisms. These capabilities are important for our long-term goal of creating the Bioinformatics Assay Environment (see Chapter 6), of which HyperFlow is a central component.

Other technologies which underlie the implementation is Eclipse's Graphical Editing Framework (GEF) [87], Eclipse Modeling Framework (EMF) [88], and the Jena [89] ontology-representation API. Services are represented using the OWL-S [3] ontology. This chapter describes the environment's features, reviews the technologies that underlie its implementation, and describes ontological and graphical issues that were faced in the implementation.

5.1 Features

5.1.1 Windows

The HyperFlow Environment is shown in Figure 49. Its main windows are:

- **Navigator** (not shown): An Eclipse component which is used for managing projects and workflow files.
- **Ontology Browser** (A): A window displaying the various classes in the available ontologies. In each ontology, classes are shown in an inheritance tree.
- **Class Properties** (B): A window that displays the properties of the class selected in the Ontology Browser.
- **Services Window** (not shown): Shows a list of available services.
- **HyperFlow Editor** (C): The main window that is used to edit HyperFlow files.
5.1.2 Interaction Mechanisms

Manipulating a diagram in the HyperFlow editor is done through standard graphical editor gestures such as moving, aligning, and resizing (where applicable). It supports features such as unlimited undo/redo stack and tooltips (showing class comments from the ontologies, for example). However, this editor is not a general purpose editor; underlying mechanisms exist for guiding the user in creating a HyperFlow diagram and ensuring its correctness. These interaction mechanisms are described in this section.

A familiar GUI feature which is used in the environment is drag-and-drop between windows. For instance, queries often start by dragging a class icon from the Ontology Browser to a query box. If the same class is dragged to the diagram or to a sub-workflow box, the user will be asked if he wants to insert a single instance or a collection of instances. The appropriate entity will then be shown inside a new result-set. In a similar way, services and existing workflows can also be dragged from the Service Browser and the Navigator, respectively, and dropped in the editor canvas.

Another predominant interaction mechanism is the use of context-sensitive pop-up menus. Most of the interaction in HyperFlow actually starts with clicking on an item
or the background of a module or diagram and choosing from the appropriate list of actions available. Among such actions are "insert a new query" and "add an input port" to a query or a workflow.

Many of the actions on the context-sensitive menu are of a type called connection expansion. For example, "Follow Property" is an action available in the context menu of a class node. Clicking on it causes the editor to reveal a submenu which lists all the properties of the class (Figure 50). Each of these, in turn, opens another submenu with the possible ranges of the property (in OWL, a property can have more than one range). The user chooses the range that he or she wants to include in the query and then indicates where to position the appropriate class or datatype node (Figure 51). The node is then inserted to the query, with the selected property edge leading to it from the original class node.

![Figure 50. Follow Property popup menu item](image)

![Figure 51. Expanding the property](image)

The same interaction type is used for adding constraints: the "Add Constraint" item in the context menu of a class or datatype icon will show a submenu containing constraints appropriate to it, which the user chooses and places (Figure 52). Note that the editor indicates that the resultant query is illegal because the new constraint is only connected to its left-hand-side operand. The editor paints in red the constraint's background and the entire query border, as is shown in Figure 53.

To rectify this situation, the user can click on the added constraint node and then select "Add a Right-Hand Value" (Figure 53). This option allows inserting the needed value, again by connection expansion. This option is only available when the constraint is not already connected to a right-hand-side value. Before inserting the
appropriate icon, the user is asked to provide a value. If the right-hand-side is a datatype, the value entered is checked against the type definition, to see if a valid value was entered. This validation process can check any DatatypeObject, even those created using complex Boolean expressions of PredicateRestrictions.

![Figure 52. Add Query Constraint](image1)

Figure 52. Add Query Constraint

![Figure 53. Create New Right-Hand Value](image2)

Figure 53. Create New Right-Hand Value

Instead of adding a new value, the user can choose to connect the constraint node to an existing value node. This is done by selecting the “Connect to value” menu option and then clicking on the value node. The system prevents the user from creating illegal connections that would lead to type errors (see Figure 54). The same legality checks are performed when a user wants to change an existing connection (property, constraint, or dataflow) between nodes in a query. The legality of the new query is assessed and only semantically correct connections are allowed.

![Figure 54. Rejecting illegal connections in Connect to Value](image3)

Figure 54. Rejecting illegal connections in Connect to Value
The HyperFlow Editor is designed to make it easy not only to compose queries, but also to use services. To this end, the same interaction mechanism is used on result-sets in the workflow diagram: In the popup-menu, the system suggests the appropriate services that can be executed on the selected result-set. It does so using semantic matchmaking between the result-set type to service profiles, written in the semantic web services description language OWL-S [3], described below. Note that it is possible to specify services that accept a certain (possibly typed) collection. These services will be suggested only when the user pops-up the context menu for a collection, not for the icon which represents the collection's type.

Another important feature that is implemented in HyperFlow is the ability to reuse a section of a workflow. This feature is called refactoring. The user selects the part of the workflow to be reused, and selects the "Refactor into New Workflow" action from the context menu. A new diagram is then opened, containing the selected sub-workflow, with the appropriate input and output ports. This diagram can then be saved and reused in subsequent sessions.

5.2 The Semantic Framework

5.2.1 Technologies Used

(i) Jena – A Semantic Web Framework for Java

Jena [89] is a Java framework enabling programmatic access for [90], RDFS [49] and OWL [2] documents. It is an open-source library that grew out of work with the "HP Labs Semantic Web Programme" [91] and has become the de-facto standard API for semantic-web document manipulation.

Jena provides two levels of APIs – an RDF-level API and an Ontology API. The latter contains classes that represent the different elements that comprise an ontology, such as OntModel, OntClass, OntProperty, Individual, Restriction (and its subclasses AllValuesFromRestriction, SomeValuesFromRestriction, MinCardinalityRestriction, etc), IntersectionClass, and UnionClass. The Ontology API also supports several inference-related capabilities: A generic interface to reasoners, simple built-in rule-based reasoners, and concrete interfaces to Description Logic reasoners through the DIG protocol [92].
(ii) OWL-S – An Ontology for Describing Services

Current ("non-semantic") web-services are usually described using the Web Service Description Language (WSDL) [66]. A WSDL document contains technical information about the service, such as its network address, parameters types (using XML Schema), output type, authentication mechanisms, and so on. However, when coming to semantically annotate web services, WSDL documents lack semantics and are too technical. For that reason, the DARPA Agent Markup Language (DAML) project had created a standard services ontology called OWL-S [3].

![Figure 55. Top level of an OWL-S service description](image)

A Service in OWL-S is described by the objects shown in Figure 55. A ServiceProfile instance explains "what the service does", in a way that is suitable for semantic matchmaking. Among the properties of this object are the different parameter types (these are usually URIs of domain ontology classes), the output type, the service type (using some type hierarchy), and so on.

The ServiceModel tells a client how to use the service by detailing the semantic content of requests in greater detail than in the profile, for example including arguments which are not important to matchmaking. It also allows defining a compound service – a series of invocations which can be conceptually described as a single service. For example, searching a nucleotide database using BLAST is conceptually a single operation. However, when using BLAST through Soaplab [11], a set of compound bioinformatics services (see Section 2.1), the client must perform a series of steps – each of them being a web-service execution. The client must first open a session, then upload the search sequence, next activate the actual BLAST search, then retrieve the results and finally close the session. This sequence of operations is modeled in the ServiceModel, and the Soaplab BLAST as a whole can be modeled in a ServiceProfile, and thus can be found during matchmaking.
The third object in the upper-level service description is the ServiceGrounding, which gives the actual details of how to interact with the service. Currently, the only fully-developed grounding is for WSDL-described web services. However, OWL-S can be easily extended to describe also other kinds of services, such as local program executions.

5.2.2 Ontology Management

Upon initialization, the HyperFlow Environment reads all domain ontologies to memory. Each ontology is represented in-memory by a separate Jena OntModel, as well as merged into a central OntModel — the global ontology. The per-ontology view of the domain is then used by the Ontologies Browser to show the class hierarchy of each ontology. The global view is used by the other semantic features, such as following properties, modifying connections, and matchmaking.

After reading the domain ontologies, the system initializes the list of service descriptions. These descriptions are given in ontologies that contain instances of OWL-S classes. For purposes of matchmaking, we are only interested in the service profiles. Thus, after loading the service ontologies, the system finds the instances of the OWL-S ServiceProfile class, and these are later used for matchmaking.

5.2.3 The Semantic Utilities

The Jena library provides an excellent in-memory representation of OWL ontologies. However, its strict correspondence to the serialized format, while important to certain kinds of applications such as ontology editors, is problematic for applications like HyperFlow. This is due to the fact that OWL semantics are not taken into account in the representation. Thus, tasks which may seem trivial work incorrectly in Jena. For example, when asking for all the instances (individuals, in OWL terms) of a class using the OntClass.listInstances() method, Jena returns only those individuals which are explicitly declared to be instances of the class, but not individuals which are declared to be instances of a subclass.

One possible solution is to use a description-logic reasoner such as RACER, FACT or Pellet (see Section 3.1.5) to answer most of the interesting questions. Reasoners are invaluable during ontology creation. I anticipate they will be found very instrumental also during the service description phase, where complex
descriptions of the service parameters may be mapped accurately to ontology classes. However, despite numerous optimization techniques, reasoners are still limited by their high time complexity (worst-case NExpTime). This creates a performance problem which makes them practically unusable for programs which constantly inspect the ontology, such as the HyperFlow Environment.

Instead of using a reasoner, I implemented a set of heuristics that emulate to a large extent the OWL semantics. These heuristics are called the semantic utilities. These can be accessed using the static methods of the OWLSemanticUtils class, which implements the Façade design pattern [93].

The semantic utilities use the inductive nature of class descriptions to take into account the superclass hierarchy, equivalent classes, intersection classes and union classes when answering the following questions:

Class and property hierarchy:
- Is class C1 a superclass of class C2?
- Is property P1 a super-property of property P2?
- Is class C an enumeration class?
- Get all values defined by HasValue (∃) restrictions on the given property at the given class or somewhere along its hierarchy.

Instances:
- Get all instances of a class C.
- Is individual I an instance of class C?
- Get all values of a given property for individual I (both stated and those defined by HasValue restrictions on the individual's types).

Properties and ranges of a given class
- Does class C1 represent a legal range of property P at class C2? (Taking into account global property ranges and local AllValuesFrom restrictions).
- What are the properties applicable to class C? What are their legal ranges and cardinalities?

The last question is very important for the HyperFlow Editor, as it is used in the "Follow Property" action, which is an essential gesture used in query building. Despite its seeming triviality, this question is not answered even by DL reasoners. The heuristics used to answer this complex question is described in Appendix 3.
These heuristics do not aim to provide the full range of DL inference provided by reasoners. However, they are sufficient for ontologies which already include the conclusions of the more complex inferences. Using a reasoner during ontology development is thus required to get accurate results. However, this has actually become a common step in ontology development, and is facilitated by the Protégé [64] ontology editor, which allows ontology designers to easily insert into the ontology statements inferred by a reasoner.

5.2.4 Service Matchmaking

An important feature of the HyperFlow Editor is service matchmaking, in which services which are applicable to the selected class are suggested to the user.

Definition: Given an ontology class, an applicable service is a service that takes as input either (see Figure 56):
1. a superclass of the given class;
2. a superclass of one of the given class’s datatype object attributes.

Note that for this definition, a class is considered a superclass of itself.

Example: A service that takes a NucleotideSequenceString is applicable to both NucleotideSequenceString and NucleotideSequenceEntry class.

Figure 56. Semantic matchmaking
A service is considered to be applicable to a candidate classes if one of the superclass relationships marked with "?" is true.

Semantic matchmaking is performed by checking the applicability of each service. Subclass checking is done by traversing the class hierarchy. Thus, the time complexity of matchmaking is $O(nm)$ where $n$ is the number of services and $m$ is the depth of the ontology's class hierarchy.
5.3 The Graphical Framework

5.3.1 Technologies Used

(i) Eclipse

Eclipse is an open source platform-independent software framework for delivering rich-client applications (as opposed to "thin clients" which are typically web clients). It is often described as an open platform "for anything and nothing in particular". So far this framework has typically been used to develop IDEs (Integrated Development Environments), such as the highly-regarded Eclipse Java IDE, which is actually the distribution of the Eclipse Platform bundled with a set of plug-ins called the Java Development Toolkit (JDT). However, Eclipse is used for other types of client application as well – from school management (the schoolclipse project [94]) through CRM (customer relations management) systems (Inventage's Capri product) to a planning and control system for NASA's next Mars exploration missions [95, 96]. Eclipse was originally developed by IBM, but is now developed by the Eclipse Foundation, an independent not-for-profit consortium of software industry vendors, among which are BEA, Borland, CA, HP, IBM, Intel, and SAP.

Central to the Eclipse platform is its sophisticated extensibility mechanism, which allows components to be bundled as plug-ins that are plugged into extension points (see Figure 57). Each plug-in contributes extensions to one or more extension points and can also define extension points to allow further extensibility. Extension points may declare an accompanying API, which often includes a Java interface that developers of extensions have to implement. For example, to add a new editor (such as the HyperFlow Editor) to the system, one creates a plug-in that provides an extension to the extension point named org.eclipse.ui.editors. The developer specifies in the plug-in manifest file the name of the extending class, which in this case implements the interface org.eclipse.ui.IEditorPart. He also specifies other information such as what are the filename extensions of files that should be opened with the editor. The platform is responsible for the plug-in lifecycle – e.g. instantiating the plug-ins when needed and providing all the extensions of a given extension point, so that the extended component could use them. In this way the platform supports interoperability between loosely-coupled independently-developed components.
While Eclipse is based on Java, it doesn’t use Java’s UI components. Instead, Eclipse uses SWT – IBM’s third generation widget toolkit for Java. This is a successor to Sun’s first and second generation toolkits - AWT and Swing, respectively. AWT provides only low-level widgets (lists, text fields, buttons, etc.) which are implemented directly with native widgets on all underlying window systems. Swing provides higher-level widgets, such as trees, tables, and rich text – all emulated using AWT. However, these emulated widgets invariably lag behind the look and feel of the equivalent native widgets, and the user interaction with them is usually different enough to be noticeable.

SWT addresses this issue by taking a different approach: While SWT provides a common, platform independent API, its implementation uses native widgets (even high-level ones) wherever possible; where no native widget is available, the SWT implementation provides a suitable emulation. This strategy allows SWT to maintain a consistent programming model in all environments, while allowing the underlying native window system’s look and feel to shine through to the greatest extent possible. This also allows tighter integration with unique capabilities of each platform – for example, under Windows it is possible to use ActiveX controls inside Eclipse.

Eclipse’s user interface also depends on an intermediate GUI layer called JFace which simplifies the construction of applications based on SWT. JFace includes components for image and font registries, common dialogs, preference pages, wizards, progress reports (for long running operations), actions and viewers.

Actions are an implementation of the Command design pattern [93] – i.e. an object which represents a particular command which can be triggered by the user via a
button, menu item, or item in a tool bar, depending on where the programmer installed it.

*Viewers* implement the Model-View-Controller (MVC) design pattern ([97], and see a description in the next section) for certain SWT widgets; standard viewers for lists, trees, and tables support populating the viewer with elements from the client's domain and keeping the widgets synchronized with changes to the domain instances. Viewers also handle common behavior and provide higher-level semantics than available from the SWT widgets. For example, when the user expand a tree node, drawing the children is taken care of by the JFace widget, provided only that it was supplied with two adaptors: a content provider that knows how to bring child model elements, and a label provider that knows how to produce the specific string label and icon needed to display a given domain element.

(ii) **GEF – The Graphical Editing Framework**

GEF is an API for creating SWT-based graphical editors. It is suitable for building any graphical editor, and has been used in a variety of projects from UML editors to visual user interface design. Like JFace viewers, it provides an implementation of the Model-View-Controller (MVC) design pattern. However, graphical editors are much more complex, feature rich and diverse than the fixed widgets that JFace provides. Thus, this library is much more complex, has a steep learning curve, and its use requires a large development effort.

In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of objects, each specialized for its task. The view manages the graphical and/or textual output to the display. The controller interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate. Finally, the model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). We will now see how MVC is implemented in GEF.

The *view* part of GEF is the library Draw2D, a SWT-based library providing graphical features such as drawing figures on a canvas, layer management, event processing for input devices, etc. When developing with GEF, one has to provide the
classes which represent the different figures. These are implementations of the IFigure interface and usually sub-classes of the Figure abstract class. Other objects in this layer are responsible for creating figures such as connections (links between figures), borders, and layers. Other objects implement view-related algorithms such as finding minimally-interfering paths for connections.

The other side of the MVC pattern is the model side. GEF is application-domain neutral and does not formally assume anything about the model. However, a model object has to provide some expected behavior, either by itself or via some adapter. For example, for most editors some way to store visual information such as location and size is needed. In addition, some sort of notification capabilities must be provided, in order to inform the controllers (the middle parts of the MVC pattern) of changes in the domain objects. The controller is then responsible to update the view according to the notification.

In GEF, there is usually a 1:1:1 correspondence between the domain model object, the controller (called EditPart in GEF) and the Figure (see Figure 58). Usually, a different EditPart subclass is created for each different model class. Instances of these classes are created using an EditPartFactory that knows how to translate from model objects to EditPart objects. The EditParts are responsible for creating the correct Figure object and updating it. To allow creating complex figures (e.g. icon + label) Figures can be nested using a built-in implementation of the Composite design pattern [93].

Figure 58. The MVC architecture of a typical GEF editor

EditParts have another responsibility: enabling model editing. In GEF, editing is done using the currently selected tool (select, pan, zoom, etc.) or through the normal JFace Action mechanism. The tool or action creates a Request object that represents
the appropriate user action (see Figure 59). This object is then sent to the appropriate EditPart — i.e. the EditPart who is under the cursor or which is currently selected. For example, the Select tool would create a move request as a response to a drag start event and send it to the EditPart which corresponds to the top-level figure under the cursor.

![Diagram of user interactions in GEF](image)

**Figure 59. User interactions in GEF**

Next, the EditPart creates an appropriate Command object, which is the one that will actually implement the requested model modification. For that purpose, the Command is initialized with references to the domain objects necessary for the change. Different EditParts can create different Commands for the same kind of Request. In addition, if the EditPart doesn't support this kind of request it will return null, and the tool will show an appropriate feedback (e.g. a no-entry sign). This promotes flexibility in treating requests – different EditParts can behave differently.

After obtaining a Command from an EditPart, the active tool can execute it through the CommandStack — an implementation of a redo/undo stack. To support redo and undo, Commands have execute, undo and redo methods. The CommandStack executes the Command and the model is modified, and this triggers an update of the view, as explained above.

As can be seen from the part of GEF described here, GEF's design is extremely object-oriented and pattern-oriented. This enables much flexibility and reuse. For example, the code that actually instantiates Command objects is not hard-coded in the
EditParts themselves but in plug-in components called EditPolicy objects, which implement the Strategy design pattern [93]. This enables different types of EditParts to use the same policy (reuse) and also allows dynamically changing them at run-time (flexibility).

GEF provides the programmer with many more features, among which are: action feedbacks, standard tools, tool palettes, connections with bendpoints and with decorations (arrows, etc.), connection routers (Manhattan, etc.), direct-editing facilities, and support for Eclipse’ mechanisms for drag and drop. All are given as fine-grained components, which often have to be subclassed to fit the model and the required behavior, and which have to be integrated manually to create a complete editor.

(iii) EMF – The Eclipse Modeling Framework

In recent years, the notion of Model-Driven Architecture (MDA) has gathered much attention. Many UML editors today provide code generation facilities that create classes according to UML diagrams. The Eclipse Modeling Framework (EMF) [88] takes this approach further. Given an EMF model the framework generates code that has, besides the correct static features such as class hierarchy, fields and methods, also the following characteristics:

- Correct functionality with respect to relationships: inverse relationships, multi-cardinality relationships (1:1, 1:n, n:n), and containment vs. reference.
- Model change notification via the Observer pattern.
- Persistence through XML-based serialization.
- Multiple inheritance using an interface hierarchy and corresponding implementation classes.
- Factory classes.
- Abstract classes which implement the Switch design pattern [93].
- A very efficient reflective API for manipulating EMF meta-objects generically.

Thus, programmers using EMF are relieved from implementing many mundane features, and can concentrate on adding the business logics of their programs. One usage for which EMF is ideally suited is creating models for GEF editors. This is done in the HyperFlow Editor.
5.3.2 GefFace – A Framework for Building Graphical Editors with GEF

As described before, GEF is designed to be completely model-independent. As a consequence, many components (EditParts, Commands, etc.) need to be subclassed to fit the model elements. Even though many graphical editors share the same set of basic visual behaviors – moving items, resizing items, nesting items in containers, etc. – these behaviors have to be re-implemented for every new model.

During the development of HyperFlow, I have created a library called GefFace, which aims to solve this problem. Among the components included in GefFace are:

- An EMF-based model which contains base classes for graphical elements such as Node, Connection, Container, Diagram, and StickyNote (see Figure 60).
- Corresponding EditParts.
- Commands for common visual behaviors.
- EditPolicy classes for installing the appropriate Commands in the EditParts.
- Figures (SelectableLabel, NamedConnectionFigure, ContainerFigure, StickyNoteFigure, etc.)
- The ConnectionStrategy framework – facilitating creation of rules for controlling the enablement of connection-related Commands (Strategy design pattern). Strategies can be combined to complex AND/OR rules by using the Composite design provided in ANDConnectionStrategy and ORConnectionStrategy.
- Additional Tools not supplied with GEF (e.g. ConnectionExpansionTool).
- A graphical-editor-specific popup-menu and sub-menu framework.
- Other components (e.g. ConnectionAnchors, ConnectionLocators).
In addition, GefFace aims to reduce the long set-up time required to start developing with GEF: GEF lacks a ready-to-use framework for creating Eclipse editors and views. (A view is an Eclipse window that does not show a document - e.g. the File Navigator, the Tasks View). Instead, every project that uses GEF needs to create its own editor using the supplied fine-grained components. While this provides the opportunity to fine-tune the installed capabilities, it also causes the initial set-up time to be very high.

GefFace provides a ready-to-use abstract editor and view classes – GraphicalEditor and GraphicalView, respectively. Among the features they provide are:

- Editor/view life-cycle management (allocate/dispose of resources, etc).
- Diagram document life-cycle management. (new, save, load, resource change tracking).
- Action registry initialization for built-in actions (including registration of actions in menus and action bars, enabling key bindings, etc).
- Initialization of GEF components such as the GraphicalViewer and the Palette.
- Listening to drag and drop events.
- Context-menu creation.
- Outline page creation.

A concrete editor that uses GefFace (such as the HyperFlow editor) implements certain (template) methods, which create the required model-specific components, for example:

- createEditPartFactory()
- createOutlinePage()
- createDiagramStorer()
- createDropTargetListener()
- createDragTargetListener()

Thus, in just a few lines of code, the developer can have a fully-functional graphical editor. Obviously, the developer can also override and extend the base class methods in order to, for instance, add editor-specific actions to the popup-menu.
5.4 Tying it All Together – the Implementation of the HyperFlow Editor

5.4.1 Model

The model hierarchy of the HyperFlow Editor is shown in the next two figures. Figure 62 (on the next page) shows how the various elements are derived from the GefFace model. Figure 61 shows the hierarchy of "typed elements" – elements which have attached some ontology resource (an instance of Jena's `OntResource` class), which can be a class, an individual, or a property. The appearance of an ontology class or individual in a diagram is called a Graphcial Instance (GI). There are GIs for classes, datatypes, individuals, datatype individuals (`DatatypeObject`'s equivalent of datatype value literals), and collections (whose types are subclasses of the appropriate collection from the Collections ontology – see Section 3.3.6). `PropertyConnection`s and `Port`s are also associated with a resource. Finally, constraints and services are also associated with a resource, which is the appropriate OWL-S service profile (constraints are represented as services to allow reusing the matchmaking process, as discussed below).

The `OntResource` in a GI will often be an *anonymous* expression created in runtime (for example, to keep as much information as possible from the definition of a query in the result type), and not a normal *named* ontology resource. Thus, when saving the diagram, it is not always sufficient to store the URI, since anonymous expressions do not have one. To support this, I implemented a mechanism for serializing OWL expressions from Jena classes to a textual description and vice versa.
Figure 62. HyperFlow Editor's Model – GefFace inheritance.
(The classes above the dashed lines are GefFace classes)
(a) containers (modules, results, collection)
(b) nodes (class, datatype, individual, constraint…) and ports
(c) connections (property, modifier, dataflow)
5.4.2 View

The HyperFlow Editor makes use of icons to represent the different types of graphical instances, modifiers, and constraints (for which the icon is specified in the profile and is thus not hard-coded). Graphical Instances are represented by a special kind of figure – an AutoArrangingLabeledFigure – which supports changing the internal arrangement of the figure (whether the icon will be to the left/right/below/above the label text). This is used in order to minimize the amount of overlap between lines and labels.

5.4.3 Controller

The HyperFlow Editor uses an EditPart hierarchy which corresponds to the model described above. Some of the interesting characteristics and typical behaviors of these objects are:

- **TypedNodeInstance** – can re-arrange its figure as described above. Provides a tooltip which contains the resource’s name and its rdfs:Label, or (for anonymous class expressions) a user-friendly detailed description (the text is created by a utility class OWLDisplayUtils).

- **CollectionInstance** – calculates its size according to the size of the child TypedNodeInstance that represents its type.

- **Port** – can be positioned only on the sides of the module (service/query/workflow). They change the orientation of the figure according to the side on which they are positioned and the type of port – input or output.

- **Query, Workflow** – allow direct-edit of their name. Automatically generate a ResultContainer with the appropriate type.

- **ResultContainer** – allows direct-edit of its name.

These are made possible using classes specific to the HyperFlow Editor, such as:

- **Specific EditPolicies** regarding moving and re-parenting objects.

- **Specific ConnectionStrategies** which control creating connections as well as reconnecting sources and targets (these are used also in some of the actions below).
5.4.4 Actions

The following actions are available in the context-sensitive menu in addition to the common actions from GEF and GefFace:

- **AddInputPortAction** – applicable to **Queries and Workflows**.
- **ConnectToOutputPortAction** – applicable to **GIs in a Query or Workflow**.
- **CreateQueryAction** – applicable to **Queries** (to create subqueries) and **Workflows**.
- **CreateWorkflowAction** – applicable to **Workflows** (to create subworkflows).
- **DataFlowConnectionCreationAction** – applicable to **GIs**. Allows connecting dataflow connections only to compatible **InputPorts** (according to matchmaking).
- **ExpandQueryConstraintMenuCreatorAction** – applicable to **TypedNodeInstances**. Creates a submenu which suggests appropriate constraints, according to matchmaking of the GI's ontological type against the constraints' OWL-S Profiles.
- **ExpandQueryConstraintAction** – shows as a menu item in the above submenu. When selected, launches the process of actually expanding (adding to the diagram) the selected query constraint.
- **ExpandPropertyMenuCreatorAction** – applicable to **ClassInstances**. Creates a submenu which suggests properties which are applicable to the class (according to the algorithm described in Section Appendix 3 and used through OWLSemanticUtils).
- **ExpandPropertyRangesMenuCreatorAction** – shows as a menu item in the above submenu. When selected, creates another submenu which suggests classes and instances which are valid ranges or values to the selected property at the selected class.
- **ExpandPropertyAction** – shows as a menu item in the above submenu. When selected, launches the process of actually expanding the selected property and range.
- **ConstraintToGICreationAction** – applicable to **QueryConstraints** (which are not connected to enough outgoing connections according to their profile). When selected, allows connecting the origin node to
GIs representing the right-hand-side of a constraint or modifier. The GI should have a correct type (according to the left-hand-side element and the type of constraint) to be connected.

- **ExpandConstraintValueAction** – applicable to **QueryConstraints** (again, which are not connected to enough outgoing connections according to their profile). Allows adding a new datatype-value as the right-hand-side argument of a constraint. When selected, brings up a dialog box in which the user can enter the value. Before adding the value to the diagram, it is checked for validity against the definition of the **DatatypeObject** (including complex descriptions that use **PredicateRestrictions**).

- and more.

These specific actions, behaviors, and strategies tie together the semantic and graphical features and provide a user-friendly interface that enables easily creating correct queries and workflow.
Chapter 6.

Summary and Further Work

6.1 Summary

In this work I have presented HyperFlow, a novel visual language in which – for the first time – queries, services and workflows are seamlessly integrated. In addition, HyperFlow’s ontology model – which is based on OWL – provides expressive domain modeling; again, this was not done before (as far as I know) for a full-fledged VQL. HyperFlow’s approach provides a way to specify queries and information-handling workflows of various complexities in a natural way. This is further facilitated by the HyperFlow Environment which allows creating HyperFlow diagrams in an easy, point-and-click manner.

During the implementation of HyperFlow, several general-purpose by-products were created: DatatypeObjects – a framework for the definition of custom, complex, semantically-typed datatypes; OWLSemanticUtils – heuristics that apply OWL semantics to Jena Ontology API objects; and GefFace – a framework for building Eclipse-based graphical editors and views with GEF and EMF. These developments should be useful to others researchers and developers who are building Semantic Web applications and graphic editors.

To fulfill our goal of allowing end-users to conduct complex analyses, it is my intention to further develop the HyperFlow language and implementation. Special attention will be devoted towards the creation of the Bioinformatics Assay Environment.

6.2 Further Work

The following sections will list steps for further work on the language and its implementation.
6.2.1 Language

The HyperFlow language, as described here, contains mostly dataflow constructs. One future direction is to further increase the expressive power of the language by adding several control-flow constructs, which may include conditional execution, explicit loops, support for SQL's CASE statement, service orchestration, and error-handling.

Preliminary discussions with prospective users about the language's intuition and effectiveness in helping to construct information handling processes have been reassuring. Still, there is bound to be room for improvements in the language and interface based on users' feedback. These could be discovered by collaborating with potential users and conducting a usability study.

6.2.2 Implementation

The HyperFlow Environment will not be complete without the ability to execute the formulated queries and workflows, and to visualize the results. This will require substantial development, some of which is discussed in this section.

HyperFlow presents an object-oriented view of the information. An object-oriented model is more natural to most users than the relational model, especially in domains with complex entries and many relationships. However, most database engines are relational. Using these databases will require a technology known as object-relational mapping (ORM). Several implementations of ORM frameworks are available, such as Hibernate and Apache OJB. However, they are not sufficient, as they lack many features required by HyperFlow, and will thus need to be extended.

One feature which is crucially needed is support for views. Currently, ORM solutions allow retrieving result-sets, but not their re-use in subsequent queries. This needs to be added. In addition, support for server-side views must be added as well, to prevent unneeded large transfers and allow usage of non-materialized views. One challenge will be to support views comprising complex structs, which will need to be represented by multiple relational views. Another point that needs to be addressed is the mapping of ontology individuals (that participate in enriched class descriptions, see Section 3.2) to database entries.

After mapping the data to the enriched object-oriented model, queries will need to be automatically generated from their HyperFlow representation. This may require
building a textual OQL query, although other possibilities exist, such as using the *query by criteria* API available with some the ORMs. Mapping the graph part of HyperFlow should be pretty straightforward: create a variable for each class in the `FROM` clause, apply the constraints between the variables in the `WHERE` clause, and construct the needed projection in the `SELECT` clause. However, the algorithm will get more complex when dealing with other constructs such as group-by and subqueries. In addition, pre-submission query optimizations may be desirable.

Besides query generation and execution, workflow execution will also need to be implemented. This will require developing features such as support for iterations, service monitoring and control, and workflow nesting. Optimizations could be done to make use of advanced service capabilities, such as *stateful* services that prevent – through the use of *sessions* – the need to transfer information back and forth. In addition, *wrappers* for common types of services (other than web-services) such as local command-line programs and grid services, will need to be developed. This may require defining new OWL-S grounding classes (see 5.2.1(ii)).

When the execution framework will be ready, the interface will need to support two modes of operation: *interactive (ad-hoc) exploration* and *design-and-execute* (see Section 1.2). In addition, *result-set management* capabilities, for dealing with views (storing them for further analysis, deleting unneeded intermediate results, etc.) will need to be added to the interface. To support browsing complex ontologies, a *graphical ontology browser*, which will enable viewing the ontology as a graph, will be needed. Lastly, dynamic screens for dealing with service arguments (as well as hiding/revealing them in the diagram) will need to be developed.

Another important feature is to provide *visualizations* of the retrieved information. For this purpose, *viewers* could be plugged into the environment. Each viewer will register the classes it can show, and matchmaking will be done to allow users to choose among appropriate viewers. Generic viewers, such as tables and charts should be developed. Eventually, the system will provide mechanisms for developing viewers that support analysis by *direct manipulation*. Operations performed in this way will still be added to the HyperFlow diagram which represents the analysis session, so that HyperFlow’s features of traceability and reproducibility will be kept.

Viewers and services that support statistical analyses should also be bundled-in, perhaps by integration with the statistical language R [46]. In addition, domain-specific viewers could be created, to provide specialized visualizations in whatever
domains the HyperFlow Environment is used. In particular, bioinformatics domain-specific wrappers and viewers will be developed, to create the Bioinformatics Assay Environment (see Section 2.4).
Appendix 1.

ModelTranslator

The following algorithm translates a normal OWL-DL ontology to one that uses DatatypeObject.

ModelTranslator(ontology O)
1. add the Datatypes ontology into O's imports
2. For each $\forall$, $\exists$ restriction $R$ in $O$:
   $rr = R$'s restriction range
   $p = R$'s restricted property
   Set $rr = \text{translatePropertyRangeToClass}(p, rr)$
3. For each $\exists$ restriction $R$ in $O$:
   $rr = R$'s restriction range
   $p = R$'s restricted property
   Set $rc = \text{translatePropertyRangeToClass}(p, rr)$
   Set $rr = \text{translateDataToIndividual}(P, rc)$
4. For each individual $i$ in $O$:
   For each datatype property $p$ and its value $v$:
     $r = p$'s (global) range
     Set $c = \text{translatePropertyRangeToClass}(p, r)$
     Set $v = \text{translateDataToIndividual}(v, c)$
5. For each datatype property $p$ in $O$:
   $r = p$'s (global) range
   Set $r = \text{translatePropertyRangeToClass}(p, r)$
   Set $p$ to be an ObjectProperty

translatePropertyRangeToClass(property $p$, datatype range)
1. className = $p$'s local name + ' - ' + typeOf(range)
2. if (translationMap contains className)
   return translationMap.get(className)
3. Set cls = a new class called className
4. for each super property $supP$ of $p$:
   1. $supC = \text{translatePropertyRangeToClass}(supP, supP$'s global range)
   2. make $supC$ a super-class of cls
5. if no superclass was assigned yet:
   1. $supC = \text{translateDatatypeToClass}(range)$
   2. make $supC$ a superclass of cls
6. if range is an enumeration:
   1. enum = new enumeration class
   2. for each data value $v$ in the enumeration:
      add translateDataToIndividual($v, cls$) to enum's individuals
   3. set cls to be equivalent to enum
7. if range is not the global range of $p$ (it came from a restriction)
   1. $r = \text{translatePropertyRangeToClass}(p, p$'s global range)
   2. make $r$ a superclass of cls
7. add (className => cls) to translationMap
8. return cls

translateDataToIndividual (data-value v, class cls)
1. name = cls's URI + v's lexical value
2. if (valuesMap contains name)
   return valuesMap.get(name)
3. Set i = new individual of type cls
4. add property-triple (i, datatypes:value, v)
5. add (name => i) to valuesMap
return i
Appendix 2.

HyperFlow's Expressive Power

In this appendix I will show that HyperFlow's expressive power is at least as strong as that of OQL. This is done by showing a reduction from OQL's grammar, represented in Extended Backus Naur Form (EBNF) to HyperFlow's constructs, thus proving that each legal OQL query can be depicted in HyperFlow.

In the following pages, we will follow the OQL EBNF [65] and describe for each group of production steps the equivalents in HyperFlow. In many cases, the reader is referred to the appropriate subsections in Chapter 4.

OQL EBNF Syntax Conventions

An EBNF rule has the form

\[ \text{symbol} ::= \text{expression} \]

where the syntax expression \textit{expression} describes a set of phrases named by the non-terminal symbol \textit{symbol}. The following notations are used for the syntax expressions:

- \( n \) is a nonterminal symbol that has to appear at some place within the grammar on the left side of a rule, all nonterminal symbols have to be derived to terminal symbols.
- \( t \) represents the terminal symbol \( t \)
- \( x y \) represents \( x \) followed by \( y \)
- \( x | y \) or \( ( x | y ) \) represents \( x \) or \( y \)
- \( \{ x \} \) represents \( x \) or empty
- \( \{ x \} \) represents a possibly empty sequence of \( x \)

OQL Grammar and its HyperFlow equivalents

\[
\begin{align*}
\text{queryProgram} & ::= \text{declaration} \{ ; \text{declaration} \} \{ ; \text{query} \} \mid \text{query} \\
\text{declaration} & ::= \text{import} \mid \text{defineQuery} \mid \text{undefineQuery}
\end{align*}
\]
import ::= import qualifiedName [as Identifier]

qualifiedName ::= identifier { .identifier }

The import declaration is used to uniquely define (using a fully-qualified name) the classes used in the program / query. In HyperFlow, the classes are meant to be selected from some ontology browser, so this is usually unnecessary. However, in order to deal with name clashes of similarly-named classes from different ontologies, and to promote reuse, the import information is kept and can be accessed in accompanying query settings and diagram settings dialogs.

defineQuery ::= define [query] identifier [([parameterList])]

parameterList ::= type identifier (, type identifier )

undefineQuery ::= undefined [query] identifier

The defineQuery declaration is OQL's mechanism of defining named views and parameterized queries. In HyperFlow, this is done by adding a name to the corresponding view (result box) or query module (see Sections 4.2.1 and 4.3.14). Queries in HyperFlow can be parameterized by adding input ports. Views and parameterized queries can be removed from the system by undefineQuery declarations in OQL, and by user-interface capabilities in HyperFlow.

query ::= selectExp | expr

selectExpr ::= select [distinct] projectionAttributes
fromClause
[ whereClause ]
[ groupClause ]
[ orderClause ]

projectionAttributes ::= projectionList | *

projectionList ::= projection { , projection }

projection ::= field \ expr [ as identifier ]
As explained in Section 4.3, the projection is specified in HyperFlow by connecting the fields or expression results to the output port. Selecting all attributes of an object is done by using the 'by value' class icons (Section 4.1).

Aliasing (the as keyword) is done using the AS module (Section 4.3.3). Removing duplicates (distinct) is done by using the "Remove Duplicates" module (Section 805502985).

\[
\text{fromClause} ::= \text{from iteratorDef} \{, \text{iteratorDef} \}
\]

\[
\text{iteratorDef} ::= \text{expr} \[\text{as} \ \text{identifier}] \mid \text{identifier in expr}
\]

In HyperFlow, iterators are represented by class icons in the query module. Naming the class implicitly identifies the extent, so there is no need to explicitly name the extent. To define an iterator over a complex expression, rather than an extent, its icon is connected using an 'in' constraint to the expression's result-set.

\[
\text{whereClause} ::= \text{where expr}
\]

Query constraints and joins, which are represented as expressions in the where clause, are represented in HyperFlow using constraint icons (Sections 4.3.1 and 4.3.3).

\[
\text{groupClause} ::= \text{group by fieldList} \{ \text{havingClause} \}
\]

\[
\text{havingClause} ::= \text{having expr}
\]

See Section 4.3.7 for a description of the group-by module and the depiction of the 'having' construct.

\[
\text{orderClause} ::= \text{order by sortCriteria}
\]

\[
\text{sortCriteria} ::= \text{sortCriterion} \{, \text{sortCriterion} \}
\]

\[
\text{sortCriterion} ::= \text{expr} \[\text{asc} \mid \text{desc}] \}
\]

See Section 4.3.8 for a description of the order-by module.

\[
\text{castExpr} ::= (\ \text{type} \ ) \text{expr}
\]
See Section 4.3.5 for a description of the casting module.

```
orExpr ::= orelseExpr { or orElseExpr }
orelseExp ::= andExp { orelse andExp }
andExpr ::= quantifierExpr { and quantifierExpr }
quantiﬁerExpr ::= andthenExpr
   | for all inClause : andthenExpr
   | exists inClause : andthenExpr
inclause ::= identifier in expr
andThenExpr ::= equalityExpr { andthen equalityExpr }
```

See Section 4.3.2 for a depiction of the Boolean operators "or", "and", "orelse", and "andthen" in HyperFlow, and see Section 4.3.4 for a depiction of the "for all" and "exists" constructs.

```
equalityExpr ::= relationalExpr { ( = | != ) [ compositePredicate ] relationalExpr }
   | relationalExpr { like relationalExpr }
relationalExpr ::= additiveExpr { ( < | <= | > | >= ) [ compositePredicate ] additiveExpr
compositePredicate ::= some | all | any
additiveExpr ::= multiplicativeExpr { + multiplicativeExpr }
   | multiplicativeExpr { - multiplicativeExpr }
   | multiplicativeExpr { union multiplicativeExpr }
   | multiplicativeExpr { except multiplicativeExpr }
   | multiplicativeExpr { \ multiplicativeExpr }
multiplicativeExpr ::= inExpr { * inExpr }
   | inExpr { \ inExpr }
   | inExpr { mod inExpr }
   | inExpr { intersect inExpr }
inExpr ::= unaryExpr { in unaryExpr }
```

The relational, equality, and "in" predicates are depicted in HyperFlow by their corresponding constraint icons, as described in Section 4.3.1. They can be modified by "some" and "all" modifiers, as shown is Section 4.3.4.
The various arithmetic and set-theoretic operators are depicted as in-query modules, as described in Section 4.3.11.

\[
\text{unaryExpr} \ ::= + \text{unaryExpr} \\
\quad | - \text{unaryExpr} \\
\quad | \text{abs} \text{unaryExpr} \\
\quad | \text{not} \text{unaryExpr} \\
\quad | \text{postfixExpr}
\]

Unary operators are depicted in HyperFlow as modules with one input port (except the following).

\[
\text{postfixExpr} \ ::= \text{primaryExpr} \{ [ \text{index} ] \} \\
\quad | \text{primaryExpr} \{ . | -> \} \text{identifier} \{ \text{argList} \}
\]

\[
\text{index} \ ::= \text{expr} \{ , \text{expr} \} \\
\quad | \text{expr} : \text{expr}
\]

\[
\text{argList} \ ::= ( \{ \text{valueList} \} )
\]

List selection (using [ ] ) is performed in HyperFlow using several in-query modules, that cater for the different selection options. A single element can be chosen by using a single index, which can be either connected via an input port to the module or directly written within it. Multiple elements can be selected using a collection of indexes or by specifying indexes for selection start and selection end.

Method invocation is also depicted in HyperFlow as a module, as described in Section 4.3.11.

\[
\text{primaryExpr} \ ::= \text{conversionExpr} \\
\quad | \text{collectionExpr} \\
\quad | \text{aggregateExpr} \\
\quad | \text{undefinedExpr} \\
\quad | \text{objectConstruction} \\
\quad | \text{structConstruction} \\
\quad | \text{collectionConstruction} \\
\quad | \text{identifier} \{ \text{argList} \}
\]
Each of these various operations has a corresponding HyperFlow in-query module.

Constructing new structs and objects is done in HyperFlow using the "new object" and "construct" modules (Section 4.3.10), respectively.
Implicit collection construction does not need a special construct in HyperFlow. Instead, the resultant collection is shown (the user inserts it into the query) and the elements are inserted non-visually (i.e. in a separate dialog box).

Query parameters are depicted in HyperFlow as making the previous views flow into the query via input ports.

The rest of the OQL EBNF productions are basic rules that list the available types, define the permitted identifiers, the literals, and so on. These apply also to HyperFlow.
Appendix 3.

Finding The Properties of a Class and their Ranges

The most complex of the semantic utility heuristics is the one which finds the applicable properties of a class and their ranges (a question which is not answered even by DL reasoners). This section gives a sketch of the algorithm.

The properties of a given class are:
1. properties that have this class in their domain (called *declared* or *direct* properties)
2. properties of superclasses
3. properties of equivalent classes
4. for an intersection class: union of properties of all the operand class
5. for a union class: intersection of the properties from each operand class
6. for restriction classes - the restricted property

In addition, the heuristic finds for each property:

- its most specific legal range (according to the global range, the property hierarchy and the local *AllValuesFrom* restrictions)
- its minimum and maximum cardinality (according to the global *FunctionalProperty* annotation and the local *Cardinality* restrictions and *SomeValuesFrom* restrictions)
- its known potential values (according to the *HasValue* restrictions and to values of enumerations in the range)
The algorithm:

```java
getPropertyInfosForClass(OntClass cls) {
    propertyInfos <- new PropertySet
    openedClasses <- new Set
    return recursiveGetPropertyInfosForClass(cls, openedClasses)
}

recursiveGetPropertyInfosForClass(OntClass cls, Set openedClasses) {
    // use the openedClasses set to prevent infinite cycles when
    // classes use cyclic definitions (such as union classes)
    if (openedClasses.contains(cls))
        return null
    openedClasses.add(cls)
    propertyInfos = new PropertySet();

    // first, add infos for cls's declared properties
    for each property p in cls.declaredProperties {
        info <- new PropertyInfo
        info.property <- p
        info.range <- p.range
        if p.isFunctionalProperty()
            info.maxCardinality <- 1
        propertyInfos.add(info)
    }

    // stop condition
    if (cls == owl:Thing)
        return propertyInfos

    // now, recursively add infos from the (direct!) superclasses
    for each class sup in cls.listSuperClasses
        propertyInfos.addAll(recursiveGetPropertyInfosForClass(sup, openedClasses))

    // do the same for equivalent classes
    for each class eq in cls.listEquivalentClasses
        propertyInfos.addAll(recursiveGetPropertyInfosForClass(eq, openedClasses))

    // for intersection classes, add the infos from all the operands
    if (cls.isIntersectionClass())
        for each operand class o in cls.listOperands()
            propertyInfos.addAll(recursiveGetPropertyInfosForClass(o, openedClasses))

    // for union classes, add the intersection of the infos of each
    // operand
    else if (cls.isUnionClass())
        for each operand class o in cls.listOperands()
            propertyInfos.intersect(recursiveGetPropertyInfosForClass(o, openedClasses))

    // add restriction information
    if (cls.isRestriction)
    {
        info <- new PropertyInfo
        info.property <- cls.getRestrictedProperty
```
if (cls.isMaxCardinalityRestriction())
    info.maxCardinality <- cls.getMaxCardinality()
else if (cls.isMinCardinalityRestriction())
    info.minCardinality <- cls.getMinCardinality()
else if (cls.isCardinalityRestriction())
    info.minCardinality <- cls.getCardinality()
    info.maxCardinality <- cls.getCardinality()
else if (cls.isSomeValuesFromRestriction())
    info.minCardinality <- 1
else if (cls.isAllValuesFromRestriction())
    info.range <- cls.getAllValuesFrom()
else if (cls.isHasValueRestriction())
    info.allowedValues.add(cls.getHasValue())
propertyInfos.add(info)
infos.addAllowedValuesFromEnumeratedRange();
openedClasses.remove(cls);
return propertyInfos;
}

PropertyInfo is a class whose attributes are:
- Property: OntProperty
- minCardinality: int
- maxCardinality: int
- allowedValues: Set

PropertySet is a special Set which keeps PropertyInfo objects. The set semantics is kept with respect to the property attribute of the PropertyInfo. Whenever the algorithm tries to add a PropertyInfo which has information about a property for which another PropertyInfo is already stored – the stored information is updated using PropertyInfo.setInfo():

112
PropertyInfo.setInfo(PropertyInfo newInfo) {
    // the range is either the new range (if it is more specific)
    // or the intersection class between old and new ranges
    if (newInfo.range ⊑ this.range)
        this.range <- newInfo.range
    else if not(this.range ⊑ newInfo.range)
        this.range <- newInfo.range ∩ this.range

    // the minimum cardinality is only updated UPWARD
    if (this.minCardinality < newInfo.minCardinality)
        this.minCardinality = newInfo.minCardinality

    // the maximum cardinality is only updated DOWNWARD
    if (this.maxCardinality > newInfo.maxCardinality)
        this.minCardinality = newInfo.minCardinality

    // the old and new allowed value set are intersected (set
    // intersection, not class intersection!)
    this.allowedValues <- this.allowedValues ∩ newInfo.allowedValues
}

In addition, after each addition of a new PropertyInfo to the PropertySet, the latter
automatically updates (using the method from above) all stored PropertyInfos that
have information about sub-properties:

PropertySet.changeSubPropertiesInfos(PropertyInfo newInfo) {
    for each subProp in newInfo.property.listSubProperties() {
        subPropInfo <- this.getInfoFor(subprop)
        if (subPropInfo != null)
            subPropInfo.setInfo(newInfo)
    }
}
References


[93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.
[94] "The schoolclipse project", https://schoolclipse.dev.java.net/.

    http://weblogs.java.net/blog/scottschram/archive/2005/03/nasa_explores_e.html.

    2005 Conference, 2005,

HyperFlow: שפה ויזואלית מבוססת אונטולוגיות
dהמconciliation של שאילתות ורימיות-חותמות
לצרד טренд מודע של מתחמות קצב

דולב דותן
HyperFlow: שפה ויזואלית מבוססת אונטולוגיות
המשלבת שאילתות וזרימת-נתונים
לאורך עיבוד מיריע "ז" משנתוש קזר

הобор על מחקר

לשם מילוי חלקי של הפרישות לקבלת התואר
מגייסר למידעים במדעי המחשב

דולב דותן

הוגש לסמינר לסטודנטים - מוכן טכניlogan לישראלי
ארד תשס"ו - דצמבר 2006
המחק汚עשת בשכיחיות פרוף"ה רן פינטר בפקולטה להנדעי המחשב.

ברצוני להודות ל הפרוף"ה רן פינטר על היעוץ וההדרכה, ובמיוחד על תמיכתו בכל המזונות שהşıים בשכיחיות בדוקה המחבר. בלעדיו אחורייה זאת של וחופש האקדמי נרחב, ועדבר ו-outline את ת الخامע והעונה.

ברצוני להודות ל הסבר על התייש ובויוס זות על העיניים וה黩נכה.

筆ודה ממייתלת לארשיב ריטל – על אהבתו וידדה כוכב עם על עוזרת ביערה לפי.

אני מודה להקיני וולבריה על התחפוכת הספיית הגניבה בקהילות המבונד:

IBM
# Table of Contents

1. **Motivation**
   - 1.1 Introduction
   - 1.2 Motivation
   - 1.3 Background

2. **Research: Study in-silico and Research Environment in Bioinformatics**
   - 2.1 Introduction
   - 2.2 Existing technologies to support Study in-silico
     - 2.2.1 Centralized approach
     - 2.2.2 Decentralized approach
   - 2.3 Use of HyperFlow to define Study in-silico
   - 2.4 Introduction to the research environment in bioinformatics

3. **Data Mediation**
   - 3.1: Background
     - 3.1.1 Introduction
     - 3.1.2 and its implications
     - 3.1.3 and its implications
     - 3.1.4 of OWL-DL
     - 3.2 The Semantics of Using OWL in HyperFlow
     - 3.3 Synchronized and customized data types
       - 3.3.1 relevant work
       - 3.3.2 Self-type
       - 3.3.3 Semantics of self-type
       - 3.3.4 and units of measurement
       - 3.3.5 and reasoning
       - 3.3.6 of custom data types
     - 3.4 Collections
     - 3.5 Data models

4.1. איזוף ויזואלי של ישויות אונטולוגיות
4.2. רכיבים.
4.2.1. תואצות
4.2.2. הפרשות ישויות
4.2.3. נתונים
4.2.4. איזוף-HTTPRequest
4.3. שאילתות ב-HyperFlow
4.3.1. שאילתות
4.3.2. מילות חיבור בוליאניות
4.3.3. מילות חיבור בוליאניות
4.3.4. צירופי
4.3.5. המציינים
4.3.6. בחירת קבוצות
4.3.7. הקבוצה
4.3.8. סדרים
4.3.9. הסדרת קבוצות
4.3.10. רו (concurrent):
4.3.11. מיקוד מחרוזות
4.3.12. הצים הד-כוויתים
4.3.13. שאילות מקומיות
4.3.14. ניגון שמות שאילות ורכורטאות
4.4. אליוט השפה
4.5. סקר ספרות והשוואה
4.5.1. שפות שאילות ויזואליות
4.5.2. שפות רימיט-נתונים
5.1. התכונות
5.1.1. התכונות
5.1.2. אופני אינטראקציה
5.2. מנגננים
5.2.1. מנגננים בשימוש
5.2.2. ספרות
5.2.3. Jena
5.2.4. OWL-S

HyperFlow של שפה: סכמת העבורה של HyperFlow

HyperFlow של שפה: סכמת העבורה של HyperFlow

ב- C++ - מימוש
5.1. תכונות
5.1.1. התכונות
5.1.2. אופני אינטראקציה
5.2. מנגננים
5.2.1. מנגננים בשימוש
5.2.2. ספרות
5.2.3. Jena
5.2.4. OWL-S
5.2.2
ניהול האונטולוגיות
5.2.3
כלל העור הסמינרי
5.2.4
שידור סטטי
5.3
מנגנוני הגרפיקה
5.3.1
טכנולוגיות בשימוש
Eclipse (i)
5.3.2
המשגרת לעריכה גרפית
GEF (ii)
5.3.3
משגרת המשיכת
EMF (iii)
5.4
הגרפיקה של kaffinal
5.4.1
הモデル
5.4.2
הצגנה
5.4.3
בקרה
5.4.4
פעולות
6
סקירת עבודה נוספת
6.1
סקירה
6.2
עיבוד תוספת
6.2.1
עיבוד לע Expr
6.2.2
עיבוד לע תופעה
6.3
נספח 1
נספח 2
נספח 3
ביאליוגרפי
רשימת אורים

<table>
<thead>
<tr>
<th>מספר</th>
<th>אורות</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>הולך פשוט על גלויות בביואינפורמטיקה</td>
</tr>
<tr>
<td>27</td>
<td>היררכית טיפוסי התוכן על XSD</td>
</tr>
<tr>
<td>28</td>
<td>היררכית עקף טיפוסי התוכן הבסיסי</td>
</tr>
<tr>
<td>30</td>
<td>הצבת-האליניאות הבסיסיות</td>
</tr>
<tr>
<td>36</td>
<td>HyperFlow-ב</td>
</tr>
<tr>
<td>39</td>
<td>מעבר על שייתוך כל עלים</td>
</tr>
<tr>
<td>39</td>
<td>מעבר על אוסף כלcheon</td>
</tr>
<tr>
<td>39</td>
<td>עזרה לקולט דרכך</td>
</tr>
<tr>
<td>41</td>
<td>אופנים שמעון באליניאות השוים</td>
</tr>
<tr>
<td>42</td>
<td>שمواد בא ltdין בשי צומת שכמה</td>
</tr>
<tr>
<td>44</td>
<td>נב, א, ל-א</td>
</tr>
<tr>
<td>45</td>
<td>צוות בודיק</td>
</tr>
<tr>
<td>46</td>
<td>(joints) גיוס עקרונות</td>
</tr>
<tr>
<td>47</td>
<td>&quot;באומן הרצוג&quot;</td>
</tr>
<tr>
<td>47</td>
<td>(outer join) &quot;ײַאפַּײַפורטראָיט&quot; כ&quot;ל Zendir זֶנְדִיר-חֶף</td>
</tr>
<tr>
<td>49</td>
<td>המטרה טיפוסי כלפי Mosul (down casting)</td>
</tr>
<tr>
<td>50</td>
<td>בורח הברכון</td>
</tr>
<tr>
<td>51</td>
<td>(collect) &quot;אסטון&quot;</td>
</tr>
<tr>
<td>51</td>
<td>קבץ לפל</td>
</tr>
<tr>
<td>52</td>
<td>having-ל</td>
</tr>
<tr>
<td>53</td>
<td>קבץ לפל</td>
</tr>
<tr>
<td>54</td>
<td>סדר לפל</td>
</tr>
<tr>
<td>54</td>
<td>הורדתكلفויות המתחづאה</td>
</tr>
<tr>
<td>55</td>
<td>...construct-ב</td>
</tr>
<tr>
<td>55</td>
<td>...construct object-ב</td>
</tr>
<tr>
<td>56</td>
<td>עירוב גוף דרש</td>
</tr>
<tr>
<td>57</td>
<td>פṚhoot על שייתוך מקבץ</td>
</tr>
<tr>
<td>57</td>
<td>אוליגוס ביים-ליגיים זרコー</td>
</tr>
<tr>
<td>60</td>
<td>(uncorrelated) שאילתת מقانون-לא-ッシיה</td>
</tr>
<tr>
<td>60</td>
<td>שאילתת מقانون מתחזאת</td>
</tr>
<tr>
<td>61</td>
<td>שאילתת-קורסיבית</td>
</tr>
<tr>
<td>63</td>
<td>Gql</td>
</tr>
<tr>
<td>64</td>
<td>MDDQL</td>
</tr>
</tbody>
</table>
רשימת טבלאות

20 .......................................................................................................................... OWL-DL

35 .......................................................................................................................... HyperFlow

69 ..........................................................................................................................HyperFlow

טבלא 1. מרכעי
טבלא 2. עלמות של ישויות ב-HyperFlow
טבלא 3. השוואת תכונות בו מופר שפת ישות של HyperFlow וויזואלית ובי

תקציר מורחב

概述

近年来，随着数据量的激增，处理数据量和获取数据的能力已成为迫切的需求。在过去，数据处理的手段主要是数据清洗和数据分析。随着技术的发展，特别是大数据和人工智能技术的发展，数据处理的工具和方法也发生了巨大的变化。

在数据处理中，数据的清洗和分析是两个重要的环节。数据清洗主要包括数据的校验和替换，而数据分析则是通过统计和分析手段，挖掘出数据中的有用信息。随着大数据和人工智能技术的发展，数据处理的工具和方法也发生了巨大的变化。在数据处理中，数据的清洗和分析是两个重要的环节。数据清洗主要包括数据的校验和替换，而数据分析则是通过统计和分析手段，挖掘出数据中的有用信息。随着大数据和人工智能技术的发展，数据处理的工具和方法也发生了巨大的变化。在数据处理中，数据的清洗和分析是两个重要的环节。数据清洗主要包括数据的校验和替换，而数据分析则是通过统计和分析手段，挖掘出数据中的有用信息。随着大数据和人工智能技术的发展，数据处理的工具和方法也发生了巨大的变化。
HyperFlow. שפת 2

על מנט תלך על הגדות של התכלית ערב מידי מואדחים כפ SH-patroעל, אתי ומיעוט
לכלת את חמ עמין ששת שפת ויו-. של כ כמי מום את
ויואלת השתי המדגמים בק ס Orioles מים של שפת ויואלת: שפת שיאולאת ויוואלת
שפחת ירموت-תנוגי ויואלת. במקורה הזועב זית מוסק חלקי, מינו שלחר ובבל ש-
שפחת מאמNotNilת שלבל פסות של שיאולאת לתוך תכלית ורו, אלא יג ממספת HyperFlow
יכלו בימי מונבר, ג(instruktיבית, פשוטות, ג בקנאות שיאולאת וגו בחרת התכלית
ורומן.

HyperFlow מתחсот למחו כמעכל פEgypt שפת השיאולאת מוספת לשערת שפת
העלא יכ כמלפ קוממט שפת אוחדה לשער לוב מוסק הווהטיות מים. שיאולאת
뷰צנות באופי אינטראקטיבי, זב שפות ביצה רק מרכז מיהר יותר הדורמה ה.(declarative)
– צורק שמתיאופ לאמול שבי שיוויים ממתושש קבוט לחושב על שיאולאת מרכבודה.
החלק הציטונית מפגש המשמעות ליעי את הת-גרה של סכנת מסד הווהטיות שבר ושסكت
שיאולאת, גלקסף איילינג (מעון =, ) של העפות השווייה. חול הז שקול חלב ש-
– SQL/OLC, מורה. החלקים והפסים של השיאולאת – בק מורה. SQL/OLC WHERE הת-
tenant ובו – מתייצב על נטני-ין \( \text{SELECT, GROUP-BY, ORDER-BY} \)
tenant: חפי גורמי תונים (ממורכז ממתושש מידי-גרמות הורמה) מאמפיתר "לזרום" את
האל}><\( \text{GROUP-BY, scripts גלריים של פריטי חזרה, } \text{SELECT, GROUP-BY, ORDER-BY} \)
Presence, סידור זכי \( \text{לזרום סידורי "ערביים" דרכו החזרת בכניסה המתחמותי.} \)
בנוסח הראשית שיאולאת בקנות, מסתע פ시험 שערת הבהונות לקבולה
יכלו ל黑龙江省 שיווהים מיבורז נקז שורתי רשת, שיווהים פקומיסים
ל(\( \text{web services, GROUP-BY, scripts גלריים של פריטי חזרה, } \text{SELECT, GROUP-BY, ORDER-BY} \)
(\( \text{GROUP-BY, scripts גלריים של פריטי חזרה, } \text{SELECT, GROUP-BY, ORDER-BY} \)
בנוסף, התוכנית מחזה להליס מadows מחזות זיוהים אינטראקטיבי.
תhões דagnostics ו- \( \text{scopes גלריים של פריטי חזרה, } \text{SELECT, GROUP-BY, ORDER-BY} \)
לוב, \( \text{לוסקים מחזה להליס מadows מחזות זיוהים אינטראקטיבי, } \text{SELECT, GROUP-BY, ORDER-BY} \)
(\( \text{iteration גלריים של פריטי חזרה, } \text{SELECT, GROUP-BY, ORDER-BY} \)

II
HyperFlow Editor

The working environment I developed is based on the Eclipse platform. It is a framework for developing applications using the HyperFlow language, which is a graphical, ontology-based language for specifying workflows. The HyperFlow language is based on the OWL-S standard, which allows for the description of Web services.

The system is designed to be user-friendly, with a point-and-click interface. Users can easily define processes by dragging and dropping elements onto the canvas. The system also provides semantic matchmaking, which helps users find relevant services.

The system is based on the Semantic Web, using OWL (Web Ontology Language) for the description of concepts. The system supports user-defined data types and collections, allowing for the definition of complex data structures.

The working environment I developed is a tool for specifying workflows, and it is designed to be user-friendly, with a point-and-click interface. Users can easily define processes by dragging and dropping elements onto the canvas. The system also provides semantic matchmaking, which helps users find relevant services.

The system is based on the Semantic Web, using OWL (Web Ontology Language) for the description of concepts. The system supports user-defined data types and collections, allowing for the definition of complex data structures.

The working environment I developed is a tool for specifying workflows, and it is designed to be user-friendly, with a point-and-click interface. Users can easily define processes by dragging and dropping elements onto the canvas. The system also provides semantic matchmaking, which helps users find relevant services.

The system is based on the Semantic Web, using OWL (Web Ontology Language) for the description of concepts. The system supports user-defined data types and collections, allowing for the definition of complex data structures.

The working environment I developed is a tool for specifying workflows, and it is designed to be user-friendly, with a point-and-click interface. Users can easily define processes by dragging and dropping elements onto the canvas. The system also provides semantic matchmaking, which helps users find relevant services.

The system is based on the Semantic Web, using OWL (Web Ontology Language) for the description of concepts. The system supports user-defined data types and collections, allowing for the definition of complex data structures.
4. 마크하 머커: ניסיון in-silico וסיבוב המマーker בביואינפורמטיקה

There are many hyper-flows in the research field of bioinformatics. The main motivation behind the development of HyperFlow is that it is a flexible and efficient tool for researchers, especially in the field of bioinformatics. It can be used for various analysis, including signal processing, image analysis, and data visualization. HyperFlow is a tool that allows researchers to perform high-quality analysis, including statistical tables and graphs. It is also useful for researchers who want to use this tool for their own analysis. In summary, HyperFlow is a powerful tool for researchers in the field of bioinformatics. It is a tool that can be used for various tasks, including data analysis, image analysis, and signal processing. It is also useful for researchers who want to use this tool for their own analysis.