VIQUEN: A VISUAL QUERY ENGINE FOR RDF

NICOLA DELL

Abstract. The biological and biomedical communities have been developing highly structured, rich data sets for representing, analyzing and integrating complex biomedical knowledge on the semantic web. However, the complexity of the query languages that have been developed to access these resources makes it difficult for non-technical users to explore the data sets that have been developed, limiting the utility and widespread adoption of both the data sets and the query languages. We present VIQUEN, a graphical query engine designed to allow users to express sophisticated queries, which are then compiled into a more complex underlying query language. After the query has been executed, users may explore the resulting graph using VIQUEN's graph visualization component. Preliminary evaluation suggests that VIQUEN is capable of expressing a wide variety of real use case biomedical queries.

1. Introduction

Ontologies, thesauri, and data sets available on the semantic web are actively utilized by the biological and biomedical communities for the representation, integration and sharing of knowledge. For example, ontologies have been used to automatically grade brain tumors [17], annotate medical images [15] and integrate data from heterogenous databases [33]. In order to utilize the knowledge stored in these data sets, users frequently wish to extract a subset of the data, either to learn more about the concepts that are stored in the data set or for use in another application. Sophisticated semantic web query languages, such as SparQL [13], vSparQL [29] and IML [28] have been developed to facilitate this information retrieval. However, the users interested in extracting information from these data sets are frequently not computer scientists or technical experts. They tend to find the syntax and grammar of the semantic web query languages prohibitive.

The goal of our work has been to explore methods which make it easier for users to formulate queries and view definitions which utilize the rich RDF data sets available on the semantic web [23]. Research in the database community [4] indicates that graphical or visual query systems tend to be superior to equivalent text editor systems in a number of ways. Firstly, graphical systems may be designed to guide the user's actions so as to minimize the risk of lexical or syntactic errors. In addition to this, some of the complexity of the query syntax may be hidden, allowing users to compose queries more efficiently by focusing solely on the query content. Furthermore, visual query systems may be used as educative tools

which foster a greater understanding of both the data sets in question and the query language.

A number of visual query tools, discussed in detail in section 3 of this paper, exist for querying the RDF data sets available on the semantic web. Many of these tools either focus on generating queries using the declarative semantic web query language, SPARQL [13], or are custom built for a particular data set. However, higher level transformation query languages, such as IML [28], have recently been developed to enable users to construct semantic web queries more intuitively. Furthermore, most of the visual query tools available do not provide a graphical visualization of the query results. Additionally, while several tools exist for visualizing RDF data, the majority of these tools do not provide functionality for generating queries in addition to the visualization. There is thus a need for graphical tools which support both the formulation and execution of queries over a variety of RDF data sets, as well as providing a visualization of the query results for further exploration and analysis.

In this paper we introduce VIQUEN, a graphical tool for semantic query construction, execution and visualization that is based on the IML data flow graph transformation language for manipulating RDF data. VIQUEN consists of three main parts, each of which will be described in greater detail later on in this paper. The first part is a query-building environment in which high-level query operations are expressed by individual graphical components that are then chained together to represent the entire query. The second part is the execution environment, which presents the IML query that has been compiled from the graphical query components. In this environment, the user may examine and execute the compiled query, and obtain the resulting RDF data. The third part of VIQUEN consists of a visualization environment which depicts the query results as a graph consisting of nodes and edges. This visualization facilitates further exploration of the query results by providing a number of options for browsing and manipulating the data.

Our objectives in this paper are to describe VIQUEN, illustrate the kinds of queries that may be formulated using the system, and present a preliminary evaluation of the expressivity of the system. The rest of this paper is organized as follows. In section 2, we give some background on the underlying data models and query languages that support the system. We then discuss other projects that relate to our work on VIQUEN in section 3, before providing a detailed presentation of the VIQUEN system, including a discussion of each of the three main system components as well as implementation details in section 4. In section 5, we present a number of sample queries that have been successfully generated using VIQUEN, including several real view definition use-cases. In section 6 we evaluate VIQUEN on the expressivity of the queries that can be composed. We finish with a discussion of both the strengths and limitations of the system, the larger questions that arose during this project, as well as an outline of our suggestions for future work.

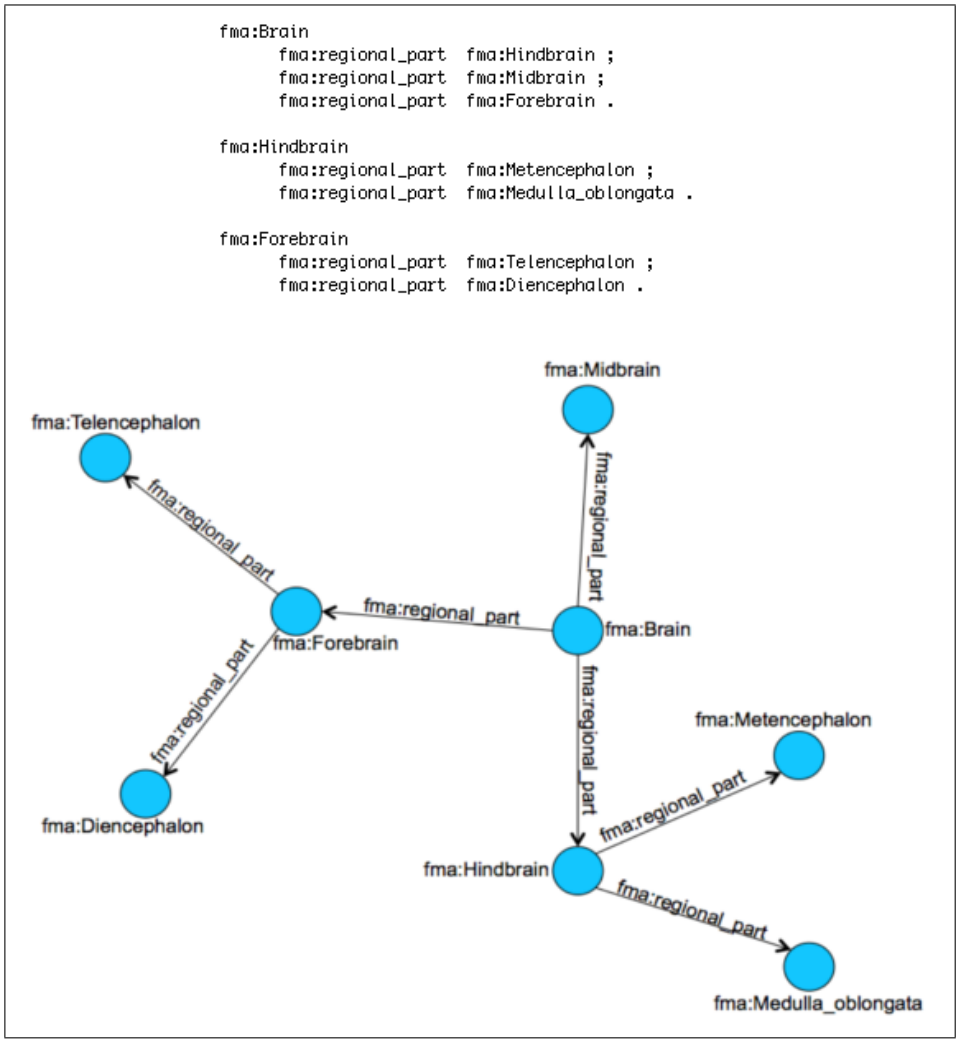


FIGURE 1. A textual and visual depiction of an RDF snippet

2. BACKGROUND

2.1. **RDF, SPARQL and vSPARQL.**

RDF (Resource Description Framework) [23] is the model developed by the W3C for describing data on the semantic web. RDF defines statements about related resources using a directed, labeled graph of triples (subject, predicate, object). Resources may be either URIs or literals, and predicates indicate relationships between resources. Figure 1 shows both a textual and a visual depiction of a snippet of RDF taken from the Foundational

Model of Anatomy (FMA) [25], consisting of a node representing the brain and part of the tree of regional part properties rooted at the brain node.

SPARQL [13] is the query language developed by the W3C for querying RDF data. A query indicates specific triple patterns to be found in an RDF graph using a combination of ground facts and variables. If the triple pattern is found in the RDF graph, the query is successful and a set of variable bindings corresponding to those instances can be returned, either as a result set or as an RDF graph.

```
SELECT ?x ?y ?z
FROM NAMED <brain_rparts> [
     # BASE
     CONSTRUCT { fma:Brain fma:regional_part ?c }
     FROM <http://sig.biostr.washington.edu/fma3.0>
     WHERE {
           fma:Brain fma:regional_part ?c  .
     }

     UNION

     # RECURSIVE
     CONSTRUCT { ?c ?b ?e }
     FROM <http://sig.biostr.washington.edu/fma3.0>
     FROM NAMED <brain_rparts>
     WHERE { GRAPH <brain_rparts> { ?a ?b ?c } .
             ?c ?b ?e .
     }
]
WHERE { GRAPH <brain_rparts> { ?x ?y ?z } }
```

(a) vSPARQL Query

```
INPUT
<http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_TREE { fma:Brain [ forward (fma:regional_part)  ]  }
GRAPH <http://sig.biostr.washington.edu/fma3.0>
}
OUTPUT <http://localhost/set1>
```

(b) IML query

FIGURE 2. A simple query in (a) vSPARQL and (b) IML

vSPARQL [29] is a set of extensions to the SPARQL query language designed to allow the creation of view definitions over RDF data. The language allows extraction, modification, and augmentation of RDF data. Figure 2(a) shows the vSPARQL query that would extract the snippet of RDF depicted in Figure 1.

2.2. **Intermediate Language (IML).**

IML [28] is a data flow graph transformation language for manipulating RDF data. It has been designed to remove some of the technical burden of creating queries and view definitions over RDF data. The syntax of both SPARQL and vSPARQL is similar to SQL, and while this may be advantageous for users with SQL experience, it can be prohibitive for non-technical users. Additionally, there is a mismatch between the high-level operations users wish to use to transform an ontology, for example to find the part hierarchy of the liver, to the declarative syntax of SPARQL. To eliminate some of this mismatch, IML provides a set of high-level graph operations that can be combined in a data flow style to represent queries. A view definition contains a set of subquery blocks that each have a specified set of input graphs and a named output graph. The results of these subquery blocks can then be directed to other subquery blocks as input graphs. Once they have been fully specified, IML queries are compiled into the more declarative SPARQL and vSPARQL query languages before being executed over the relevant RDF data sets. Figure 2(b) shows the IML query corresponding to the vSPARQL query shown in Figure 2(a). A more detailed discussion of the high-level graph operations and functionality provided by IML may be found at [28].

3. Related Work

There are a number of projects that relate to our work on VIQUEN. These fall naturally into two categories, those that focus on visualization of RDF and those that focus on query formulation. To the best of our knowledge, there is no good system that effectively combines both graphical support for query writing and visualization of the resulting graph.

3.1. **Visualization of RDF.**

There are numerous tools that have been developed to visualize RDF. IsaViz [16], is a visual environment that allows users to author as well as browse RDF models represented as graphs. Welkin [30] is a tool for visualizing RDF data that does not focus on discovering or extracting specific subsets from the data set, but is rather designed to allow the user to understand the global shape and cluster characteristics of the entire data set. Paged Graph Visualization (PGV) [7] is another tool for RDF data visualization and provides an environment where users can select a small set of objects to examine rather than presenting the entire RDF data set. Jambalaya [32] is a tool for visualizing schemas and instances of ontologies. It allows users to browse an ontology at several levels of detail by zooming in and out. FlexViz [11], which has been incorporated into the National Center for Biomedical Ontologies (NCBO) Bioportal [19], is a visualization tool for browsing a single data set where the concepts are represented by nodes and the relationships between concepts are represented by arcs. While some of the visualization tools mentioned above provide similar functionality to VIQUEN's visualization component, none of these systems provide support for querying over the RDF data sets using semantic web query languages.

RDF Gravity [14] is another tool for visualizing RDF. It predominantly focuses on graph visualization, although the system does support text based search for concepts in the RDF graph. Additionally, a user can enter a query using the semantic web query language RDQL [24] and the resulting triples are displayed in the visualization. However, the query is solely text-based and is typed in by hand. There is no visual support to assist with query formulation. RDFscape [31] is a tool that has been developed as a plug-in for the Cytoscape visualization platform [26]. RDFScape supports querying both through low-level text based queries, and using RDQL, and some preliminary work on visual queries which compile to RDQL queries is mentioned briefly. The query results may then be visualized in a graph in Cytoscape.

3.2. **Visual formulation of semantic web queries.**

There have been a number of attempts to develop systems that assist users with formulation of Semantic web queries. NITELIGHT is a web-based graphical tool for constructing SPARQL queries. The tool is primarily intended for use by those with previous experience of SPARQL and includes a columnar ontology browser, an interactive graphical design surface, a SPARQL syntax viewer and an integrated semantic query results browser. However, the functionality of the results browser is limited to displaying the raw output of the query processor in XML format. Additionally, only simple queries have been presented in the paper, and it appears that a complex query would quickly clutter the workspace and become unmanageable for the user. The iSPARQL Visual Query Builder [22] is another tool which focuses on formulation of SPARQL queries and provides similar functionality to NITELIGHT. SPARQLViz [3] also aims to support users with SPARQL query construction. SPARQLViz presents the user with a sequence of forms using a wizard-like interface. The information entered on the forms is then used to construct the SPARQL query. SPARQLViz has been developed as a plugin for IsaViz [16], and hence the query results may be visualized using the IsaViz system.

The SEWASIE query tool [5] is meant to support a user in formulating a precise query even if the user is unaware of the vocabulary of the information system holding the data. The interface is driven by an underlying ontology describing the domain of the data in the information system. The output of the system is a generated conjunctive query ready to be executed by some query evaluation engine. One of the drawbacks of this tool is that the user interface looks complex, and it may be difficult for novice users to extract information out of the ontology. Furthermore, the system does not provide any means for executing the generated conjunctive query or visualizing the query results.

OntoVQL [10] is a visual query language that offers basic functionalities for querying OWL-DL ontologies. It has been designed independently of any semantic web query language and hence may be mapped to a number of different OWL query languages. However, the visual query language does not match the full expressive power of OWL query languages. For example, datatypes and negation are features that are not available in OntoVQL but

are provided by the textual OWL query languages. This makes the OntoVQL system only suitable for simple queries mostly asked by naive users.

Emily [27] is a system that uses a more text-based approach to query formulation. Users enter a triple pattern that may include an unknown variable, and the system will return the corresponding set of triples from the Foundational Model of Anatomy (FMA) [25]. Emily also provides the user with a simple tree structured visualization of the query results. However, Emily is hard-coded to be used only with the FMA, and is unable to express sophisticated queries such as those presented in Section 5 of this paper.

4. VIQUEN

VIQUEN enables the formulation of semantic web queries using a set of graphical query components and GUI-based editing actions. The visual queries are automatically compiled into the IML query language, before being executed over the specified RDF data sets. The query results may then be visualized as a graph. We first provide an overview of the technologies utilized in VIQUEN's implementation before describing in detail each of the three main system components: the query-building environment, the execution environment and the visualization environment.

4.1. **Implementation.**

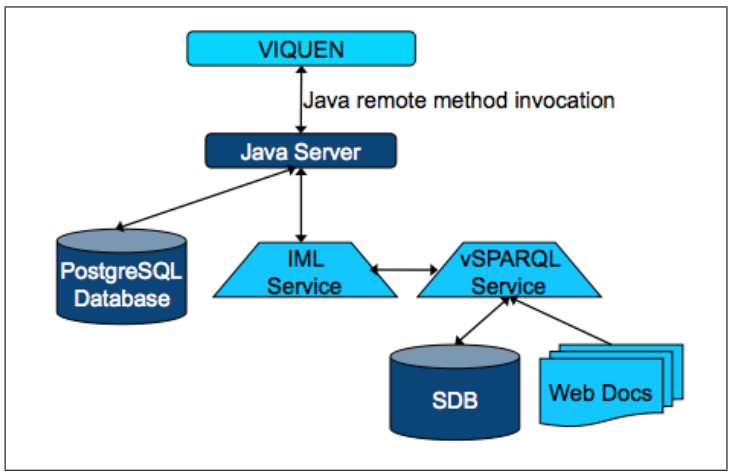


FIGURE 3. VIQUEN system architecture

VIQUEN has been implemented as a platform independent Java application. The GUI components of the system were built using the Java Swing toolkit, and both the query-building environment and the visualization environment utilize the JGraph visualization library [2]. After the queries have been compiled into IML, they are executed using the Java AMF (Action Message Format) connection protocol which connects to the Query

Manager server [8]. Figure 3 shows the architecture diagram for query execution. The resulting RDF data sets are parsed prior to visualization using the Jena Framework for building semantic web applications [12].

4.2. **Query-building environment.**

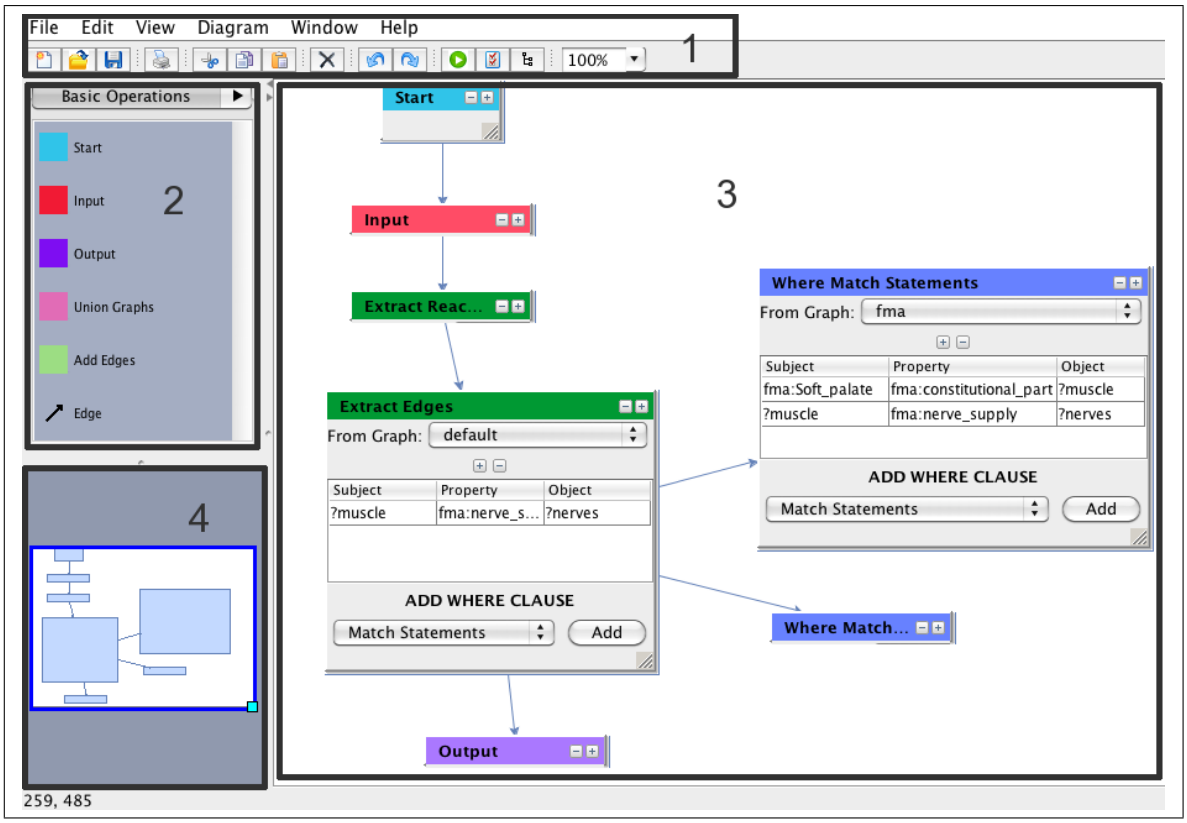


FIGURE 4. VIQUEN's query builder environment

VIQUEN's query-building environment, shown in Figure 4, may be divided into four main parts: the toolbar and system menus (Figure 4-1), the operation library palettes (Figure 4-2), the main query-building workspace (Figure 4-3) and the query-building workspace outline (Figure 4-4). The toolbar and system menus provide easy, single-click options for managing the workspace, including saving and loading queries, copying, pasting or deleting query operations, compiling queries into IML or changing the look-and-feel of the application. The toolbar also contains a shortcut *Data Sources* button which allows users to add, remove or edit the data sources and namespaces specified in the query.

The operation library palettes (Figure 4-2) contain icons which represent query operations that may be added to the workspace. The query operations have been divided into five

different palettes, with similar operations being grouped together. The *Extract* palette contains shortcuts for the five *Extract* query operations, the *Delete* palette for the four *Delete* operations, the *Replace* palette for the seven *Replace* operations, the *Where* palette for the four *Where* operations, and the basic palette, depicted in Figure 4-2, for the remainder of the query operations: *Start*, *Input*, *Output*, *Add Edges* and *Union Graphs*. Operations are added to the query-building workspace by dragging and dropping the appropriate icon from a palette. Each palette additionally contains an *Edge* icon for adding edges to direct the flow of the query.

The main query-building workspace (Figure 4-3) has been designed to take advantage of the data flow graph transformation style of IML. Each high-level query operation is represented in its own visual component. Components of the same type are the same color for easy identification. For example, all of the *Extract* operations are green, while all of the *Where* operations are blue. The visual components are then chained together, using directed edges, to compose the entire query. Thus, the system allows a long and complex query to be broken down into a number of smaller, more manageable steps which may be worked on individually. A query must begin with a *Start* operation, which indicates the point from which the system will start to compile the query. By positioning the *Start* operation appropriately, users may execute different chunks of the query individually and examine the results before combining them into a larger query. In this way, our approach allows users to formulate queries incrementally.

After the *Start* operation, the query is defined by adding one or more subquery blocks to the workspace. Each subquery block begins with an *Input* node, which defines the data sources to be used as input to the query, and ends with an *Output* node, which specifies the output graph for the block. This output graph may be added to the list of available input data sources by clicking on the *Add to data sources* button and specifying a name for the graph. Between the *Input* node and the *Output* node, the user may add a number of different query operations, including *Add Edges*, *Union Graphs*, *Extract* operations, *Delete* operations and *Replace* operations. A detailed description of each query operation is presented in Appendix A of this paper.

VIQUEN attempts to make query formulation easier by recognizing that different query operations often have common attributes. For example, *Extract Tree* and *Delete Tree* require identical parameters, while *Add Edges*, *Extract Edges* and *Delete Edges* each require a triple pattern to be specified. In order to ensure the consistency of query operations, VIQUEN breaks down each visual query operation into a number of smaller pieces, each of which are used in a variety of different operations. This consistency across operations will make it easier for users to learn how to use the system, since several operations may have identical or very similar usage. An example of one such component that is utilized in a number of different query operations is the option to add a *Where* clause. It consists of an *Add where clause* label followed by a drop-down menu indicating the four different types of where clauses available: match RDF statements, union RDF statements, filter constraints or optional constraints. *Where* clauses are important query constructs because they allow

unknown variables to be specified in the query. These variables are then bound to sets of RDF triples using the constraints and conditions specified in the *Where* clause.

Another important way in which VIQUEN enables easier query formulation is by making the user aware of which parameters are required for a particular operation. For example, from the *Extract Tree* operation, we can immediately see that we need to specify a graph on which to perform the operation, a root node from which to start the extraction, a list of properties and, for each property, an edge direction to follow. Furthermore, we can see that we have the option of adding a *Where* clause to this operation. Thus, the user does not need prior knowledge of the parameters required for each query operation, making formulating queries quicker and easier.

A complex query may have a large number of operations that need to be defined, and showing the full details of each operation simultaneously would be overwhelming and would clutter the screen. VIQUEN provides a number of features to avoid this problem. Firstly, the query-building environment allows users to collapse query operations that are not currently being edited. This helps to reduce screen clutter as well as allowing the user to focus solely on the query operation currently being edited. Additionally, keeping completed query components collapsed will prevent accidental modification of the query parameters. Secondly, as the number of query operations grows, the size of the workspace automatically expands and does not shrink operations in order to fit them all on the screen. Instead, only a portion of the workspace is displayed, with the rest of the workspace accessible by scrolling horizontally and vertically. Additionally, an outline of the entire workspace is provided at the bottom left-hand side of the screen (Figure 4-4) with a dark blue rectangle indicating the fraction of the workspace currently being viewed. Clicking and dragging on this rectangle provides another means for navigating the query workspace. Lastly, it may be time consuming and tedious for the user to constantly lay out the position of the query operations manually. VIQUEN thus provides a number of automatic lay out options which structure the flow of the query operations in a space efficient manner.

After completing the query formulation, the user will be ready to compile and execute the query. Clicking on the *compile query* button in the application toolbar will automatically compile the query into IML and open the query execution environment.

4.3. **Execution environment.**

The query execution environment, shown in Figure 5, consists of three main components: the query component (Figure 5-1), the results component (Figure 5-2) and a simple menu bar (Figure 5-3). The query component displays the generated query in IML. This provides the user with an opportunity to examine the IML, check the query for possible errors, and learn more about the query language. In generating the query, the system partitions the graphical query-building workspace into layers of subquery blocks according to the distance of each block from the *Start* operation. For example, all subquery blocks connected to the *Start* operation are in layer one, while all of the subquery blocks connected to layer one blocks are in layer two. The query is then generated by first compiling all of the layer one

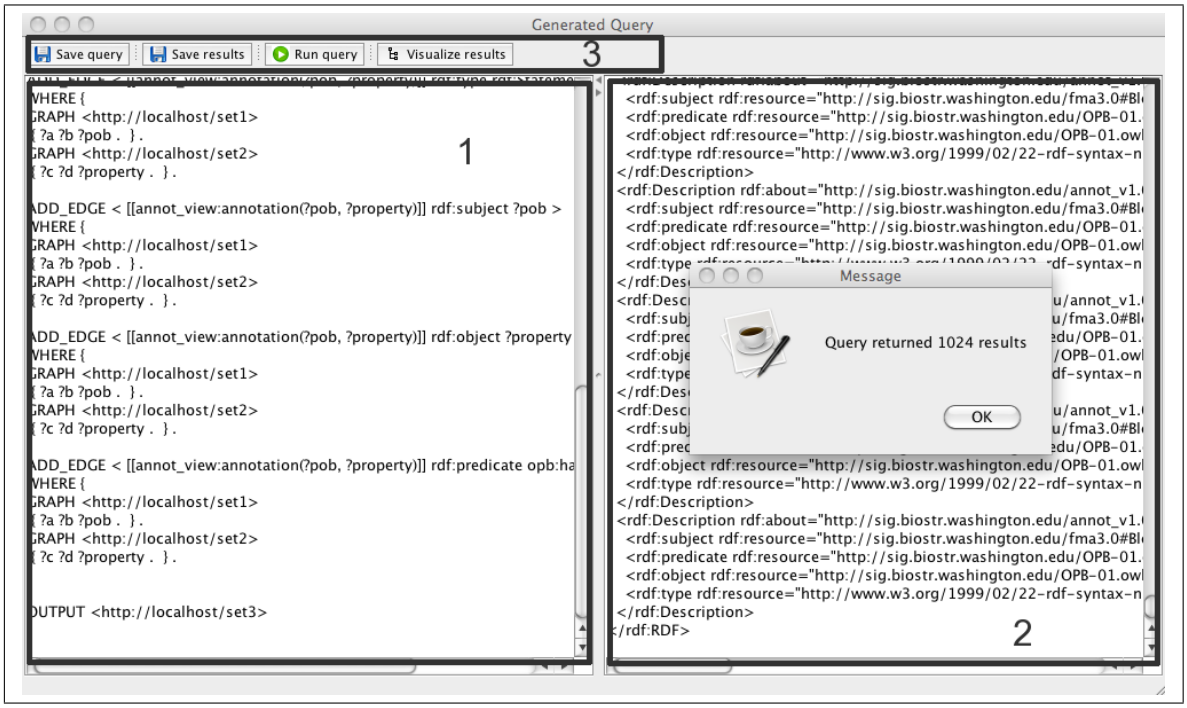


FIGURE 5. VIQUEN's execution environment

blocks, followed by all of the layer two blocks, and so on until all of the blocks that are reachable from the *Start* node have been compiled. Clicking on the *execute query* button in the execution environment toolbar will cause the query to be executed by submitting the IML query to the Query Manager server [8] using the Java AMF connection protocol.

After execution, the results of the query are returned in raw RDF/XML format, and are displayed in the result component (Figure 5-2). In addition to this, the system will alert the user to the number of RDF triples that have been returned by the query. This is an important piece of information, and may impact how the user chooses to visualize the results. For example, if a query returns a very large number of RDF triples, the user may not want to attempt to view the entire results graph on the screen at one time. At any stage, the generated query or the resulting RDF may be saved to a local file for later use, either in VIQUEN's visualization environment or in other applications such as the Query Manager [8]. After the query has been executed, the user will click on the *visualize results* button in the toolbar to open VIQUEN's visualization environment.

4.4. **Visualization environment.**

The visualization environment, shown in Figure 6, facilitates exploration and manipulation of an RDF graph. In order to minimize the amount of effort required for a user to become

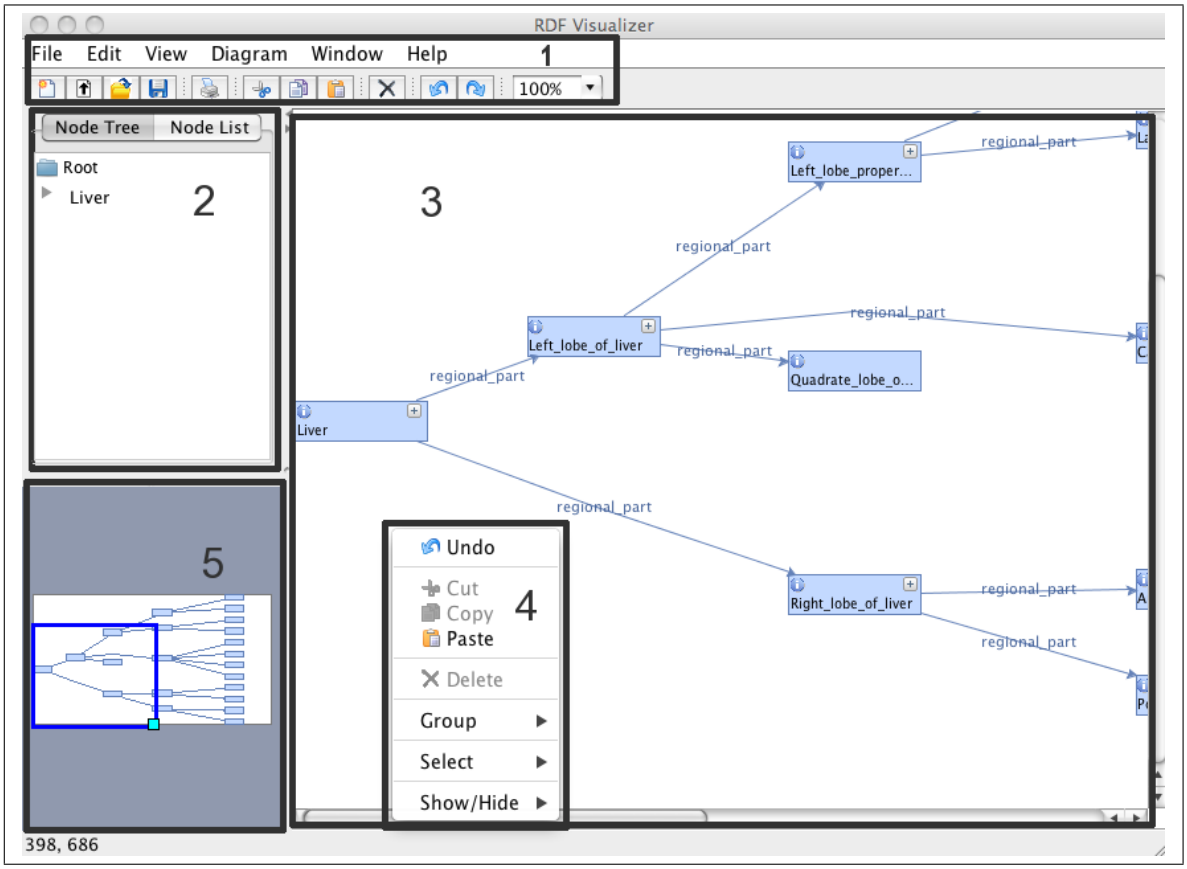


FIGURE 6. VIQUEN's visualization environment

familiar with the system, the visualization environment has been designed in a fashion consistent with the query-building environment, and utilizes similar layouts, menus and toolbars (Figure 6-1). As in the query-building environment, several automatic graph lay out options are available which give the user a variety of options for deciding the best layout for a particular graph. Visualizations can be loaded from and saved to disk using the same file format as that for saving visual queries. Furthermore, in addition to visualizing query results, the visualization environment has been designed to operate as a standalone component, and can load locally saved RDF files. This allows output from other query systems such as the Query Manager [8] to be loaded and visualized in VIQUEN.

A number of features have been provided to enable easy manipulation and navigation of an RDF graph. The upper left-hand side of the workspace (Figure 6-2) contains a node tree showing the graph of nodes, possibly containing multiple roots, which the user may browse to gain a better sense of the structure of the graph. In addition to the node tree, an alphabetized list of all of the nodes is provided which enables users to quickly and easily

search for a particular node of interest. Clicking on a node in either the node tree or the node list will make the node available for viewing and manipulation in the following way: if the node is currently visible in the workspace, the system will select it and scroll to it. Alternatively, if the node is not currently visible, the system will make the node visible, along with its children and parent nodes.

The main part of the visualization workspace (Figure 6-3) depicts the RDF visually as a graph consisting of nodes connected by edges. The nodes represent the subject and object of the RDF triple, while the edges represent the properties. Since queries may potentially return a large number of RDF triples, VIQUEN does not attempt to display the entire results graph on the screen at one time. This would be overwhelming and make the resulting graph difficult to understand and navigate. Instead, we limit the number of nodes initially displayed, and so need to find a suitable root node from which to start the visualization. To identify a root node, we first look for any nodes that have outgoing edges and no incoming edges. If there are many such nodes the system chooses any one to start from, but makes the others available as roots in the node tree (Figure 6-2). Since RDF graphs may be cyclic, it may be the case that there are no nodes with outgoing edges and no incoming edges. In this case, the system looks for those nodes that have the biggest difference between the number of incoming edges and the number of outgoing edges, and uses those as roots. Further research into techniques for how best to select a suitable node from which to start browsing the RDF graph would be an interesting area for future work.

Properties in the visualization are displayed as directed edges, starting at the subject of the RDF triple and going to the object, with the edge label consisting of the property name. Visible nodes are displayed in blue colored rectangles labeled with the name of the node. A node may be selected and moved by clicking and dragging it in the workspace, which enables manual manipulation of the graph layout. Positioning the mouse pointer over a node's *information* icon will display several additional pieces of information relating to the node, including the total number of incoming and outgoing edges for the node, since they may not all be visible, as well as the full name of the node, since this is frequently too long to be fully displayed inside the node's geometry. We anticipate that the most frequent action users will wish to perform on a node is to make its child nodes visible, and as a result we have provided a shortcut *show children* button inside the node's geometry which, when clicked on, will make all of the child nodes visible. This button is only displayed in nodes that have children, and thus also quickly indicates which nodes are leaf nodes and cannot be expanded further.

Additional functionality for further visualization and exploration of the RDF is made available to the user in a pop-up menu (Figure 6-4) which may be accessed by right clicking in the main visualization workspace. As well as the basic *cut*, *copy*, *paste*, *delete* and *undo* actions, three submenus have been implemented which group actions into *select* actions, *group* actions and *show/hide* actions. The *select* submenu has options for manipulating which portion of the graph is currently selected. Users may choose to select all of the

nodes, none of the nodes, the children of a particular node or the entire subtree rooted at a particular node. The *group* submenu provides options which allow for a number of nodes to be grouped together and then collapsed into a single representative group node. The group may then be expanded and collapsed as a single unit, or opened in a separate visualization workspace for more detailed manipulation. Using this functionality, users can create and drill down through multiple layers of the graph. The *show/hide* submenu provides a variety of choices for manipulating currently visible nodes. After selecting a node, the user can choose to show or hide the child nodes, parent nodes or the subtree of the graph rooted at that node. Additionally, there are options to show or hide the entire graph, or only the selected portion of the graph.

As the user manipulates the RDF graph, it is likely that there will be more nodes to display than would comfortably fit on the screen. When this happens, VIQUEN does not attempt to shrink the visualization in order to accommodate all the nodes. This would clutter the workspace and make the node information difficult to read. Instead, the workspace automatically expands to incorporate all of the visible nodes, and only a fraction of the currently visible workspace is displayed. A smaller outline of the entire visible graph is provided on the bottom left-hand side of the workspace (Figure 6-5), with a dark blue rectangle indicating the portion of the workspace currently being viewed. Clicking and dragging on this blue rectangle will update the displayed portion of the workspace accordingly.

5. Sample Queries

We will now illustrate VIQUEN's expressivity by presenting a variety of queries that have been generated using the system. For each query, we first describe the query before presenting the strategy employed to build it. We then show the graphical VIQUEN query along with a visualization of the resulting RDF graph returned by executing the query over the appropriate RDF data sets. The corresponding automatically generated IML queries may be found in Appendix B of this paper.

All of the sample queries are motivated by actual requests from biomedical researchers. They query over four different information sets: NCI Thesaurus [20], Reactome [1], Ontology of Physics for Biology [6], and the Foundational Model of Anatomy (FMA) [25]. The NCI Thesaurus (NCIt) is an open-source vocabulary containing information about cancer. It contains over 34,000 concepts and is available in OWL format. Reactome is an open-source database of biological pathways that contains information on humans and 22 other non-human species. Reactome consists of an OWL schema and associated data and contains more than 3.6 million RDF triples. The Ontology of Physics for Biology (OPB) is a ontology containing concepts from classical physics necessary for representing, annotating, and encoding quantitative models of biological processes. It is developed in OWL and contains approximately 2,000 RDF triples. The Foundational Model of Anatomy (FMA) is a reference ontology representing the structure of the human body. We use the

OWL version of the FMA, which contains 1.7 million RDF triples [21]. Queries 5.1 to 5.5 inclusive are basic queries that might be written by users wanting to extract information from the data sets, while queries 5.6 to 5.12 inclusive are view definition queries that have been taken from [29].

5.1. **Abdominal cavity query.**

5.1.1. *Description.* Use the FMA to find all of the organs that are contained in the abdominal cavity.

5.1.2. *Strategy.* We start by creating an unknown variable *?organ* to represent the organs in the *Abdominal cavity*. We then use a where clause to bind this variable and extract the edges from the FMA which match the pattern *?organ: contained in: Abdominal cavity*.

Figure 7 shows the query-building workspace for the abdominal cavity query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 25.

5.2. **Left frontal lobe query.**

5.2.1. *Description.* Use the FMA to find all of the regional parts of the left frontal lobe of the brain.

5.2.2. *Strategy.* In this query, we want to make sure that we discover all of the direct *regional parts* of the *Left frontal lobe* as well as any of their regional parts. We thus extract a tree of nodes in which the root is specified to be the *Left frontal lobe* and the property that we follow is *regional part*.

Figure 8 shows the query-building workspace for the left frontal lobe query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 26.

5.3. **Lung parts query.**

5.3.1. *Description.* From the FMA, extract the tree of nodes starting from the lung and following all of the different part properties. Rename these different part properties to the more simple *part* property.

5.3.2. *Strategy.* We begin this query by first extracting a tree of nodes from the FMA which starts at the *Lung* node and follows the *regional part*, *constitutional part* and *systemic part* properties. We then replace each of these properties with *part*.

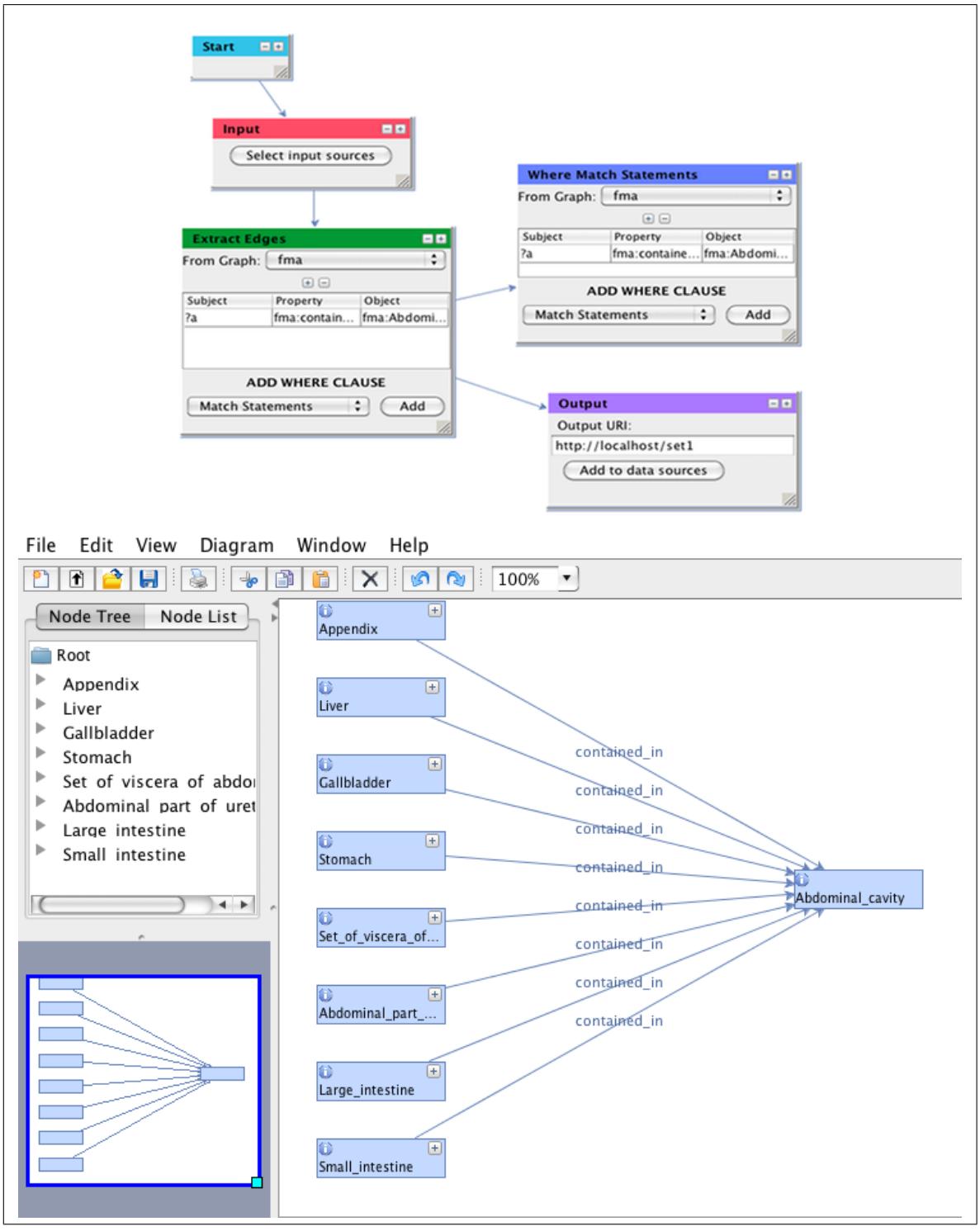


FIGURE 7. Graphical query workspace and results visualization for the abdominal cavity query

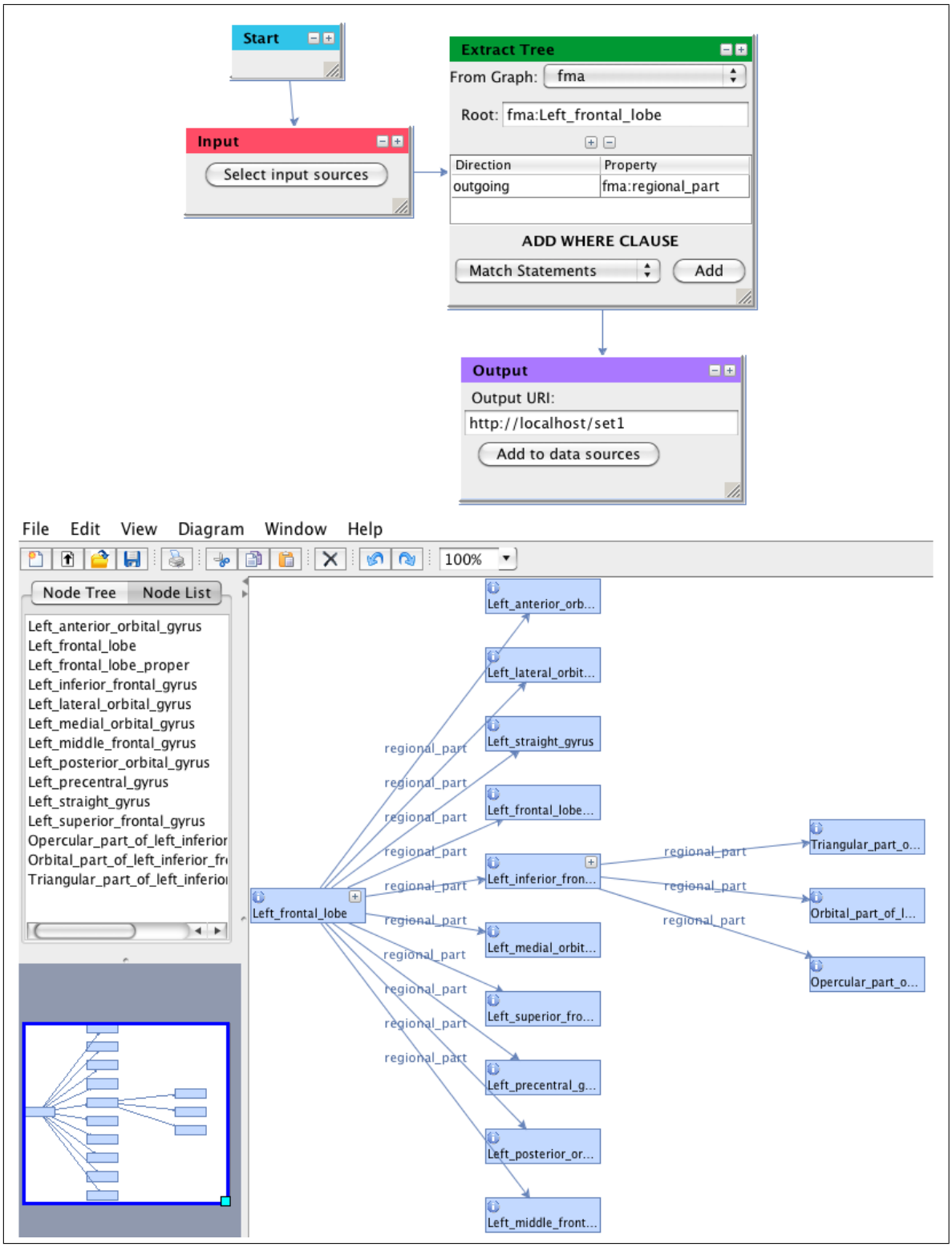


FIGURE 8. Graphical query workspace and results visualization for the left frontal lobe query

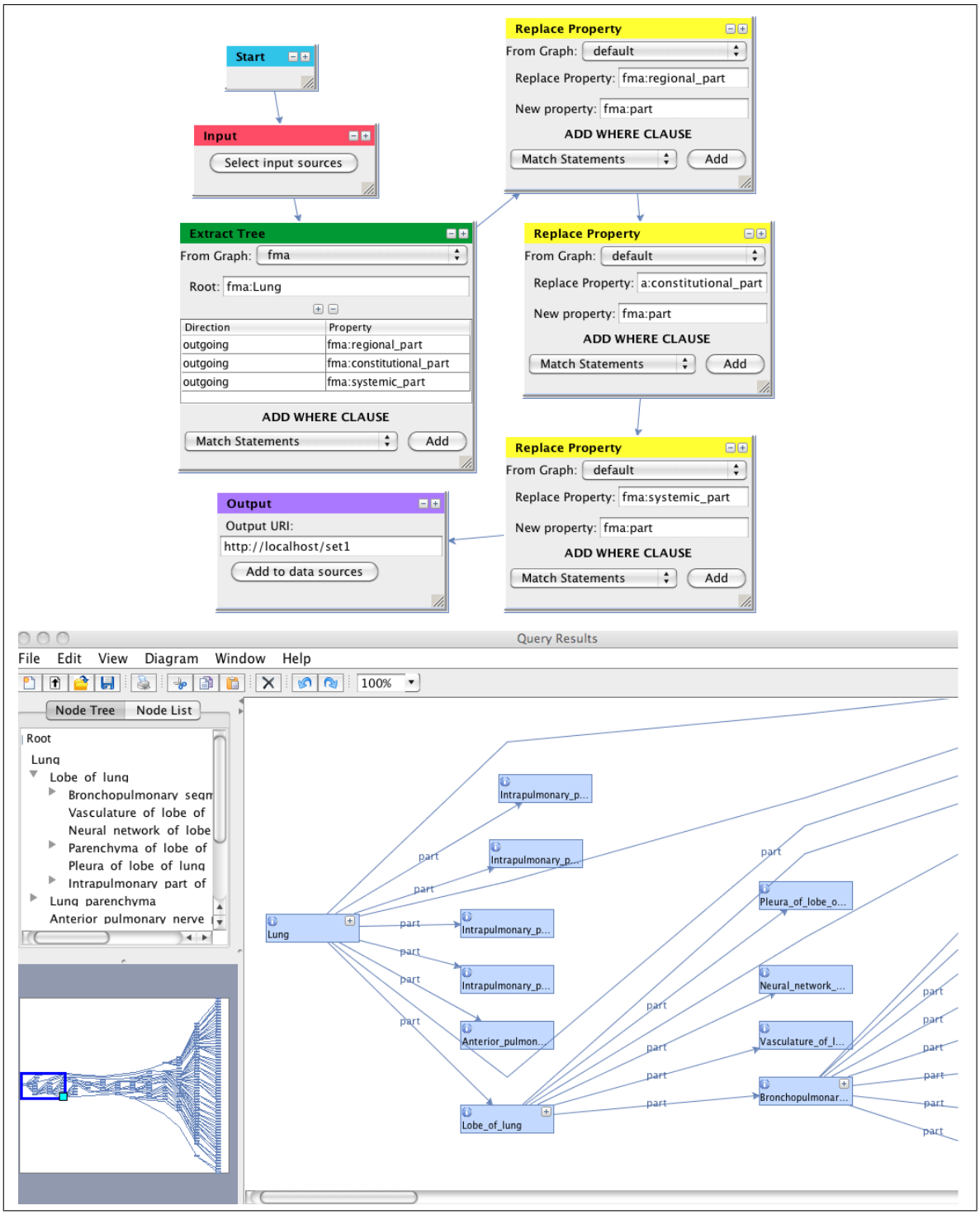


FIGURE 9. Graphical query workspace and results visualization for the lung parts query

Figure 9 shows the query-building workspace for the lung parts query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 27.

5.4. **Craniofacial query.**

5.4.1. *Description.* From the FMA, extract the name, label and ID number of all of the parts of the skull and the face.

5.4.2. *Strategy.* This query may be broken up into a number of smaller subqueries. First, we extract the tree of nodes starting at the *skull* node and recursively following all the outgoing edges with either the *regional part* or *constitutional part* property. We then extract a similar tree starting from the *face* node. The next step in the query is to compute the union of these two extracted trees, before returning to the FMA and, for each of the concepts in the union, extracting the edges which represent the *name*, *label* and *FMA ID* number.

Figure 10 shows the query-building workspace for the craniofacial query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 28.

5.5. **Soft palate query.**

5.5.1. *Description.* From the FMA, extract the nerve supplies for all of the muscles of the soft palate.

5.5.2. *Strategy.* We may identify the muscles of the soft palate using the subclass hierarchy in the FMA. All of the muscles of the soft palate will be derived from the *muscle organ* concept. Thus, the first query operation will be to extract all the concepts that are reachable from the *muscle organ* node. From this set of reachable nodes, we find which nodes are also *constitutional parts* of the *soft palate* and, for each of these, we then find the *nerve supply*.

Figure 11 shows the query-building workspace for the soft palate query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 29.

5.6. **Mitotic cell cycle.**

5.6.1. *Description.* From Reactome, extract the mitotic cell cycle, all of its component processes, and their accompanying labels. This extraction will be used by an application for exploring processes and their subprocesses. Although the data leverages an OWL specification, only process data should be included in the view.

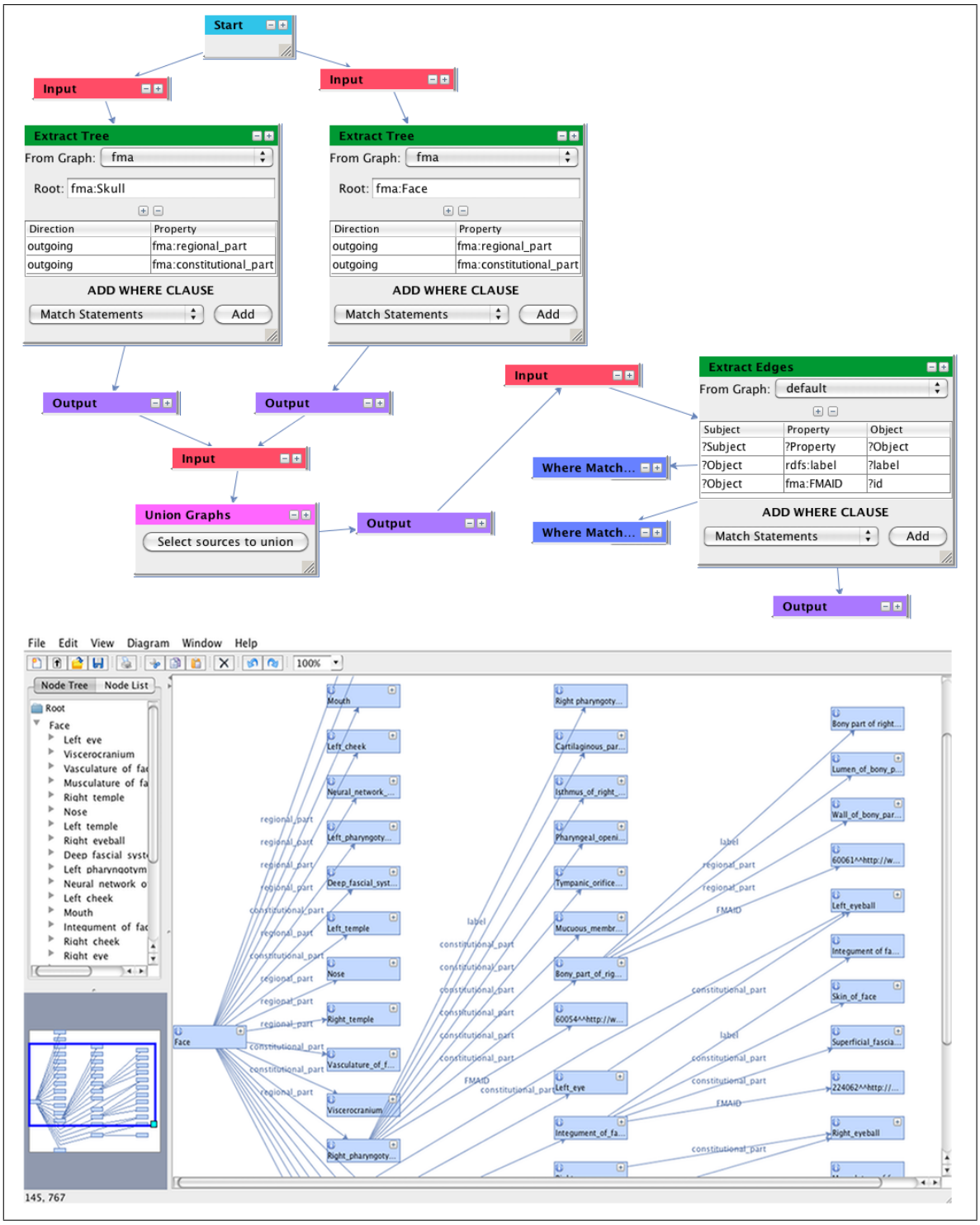


FIGURE 10. Graphical query workspace and results visualization for the craniofacial query

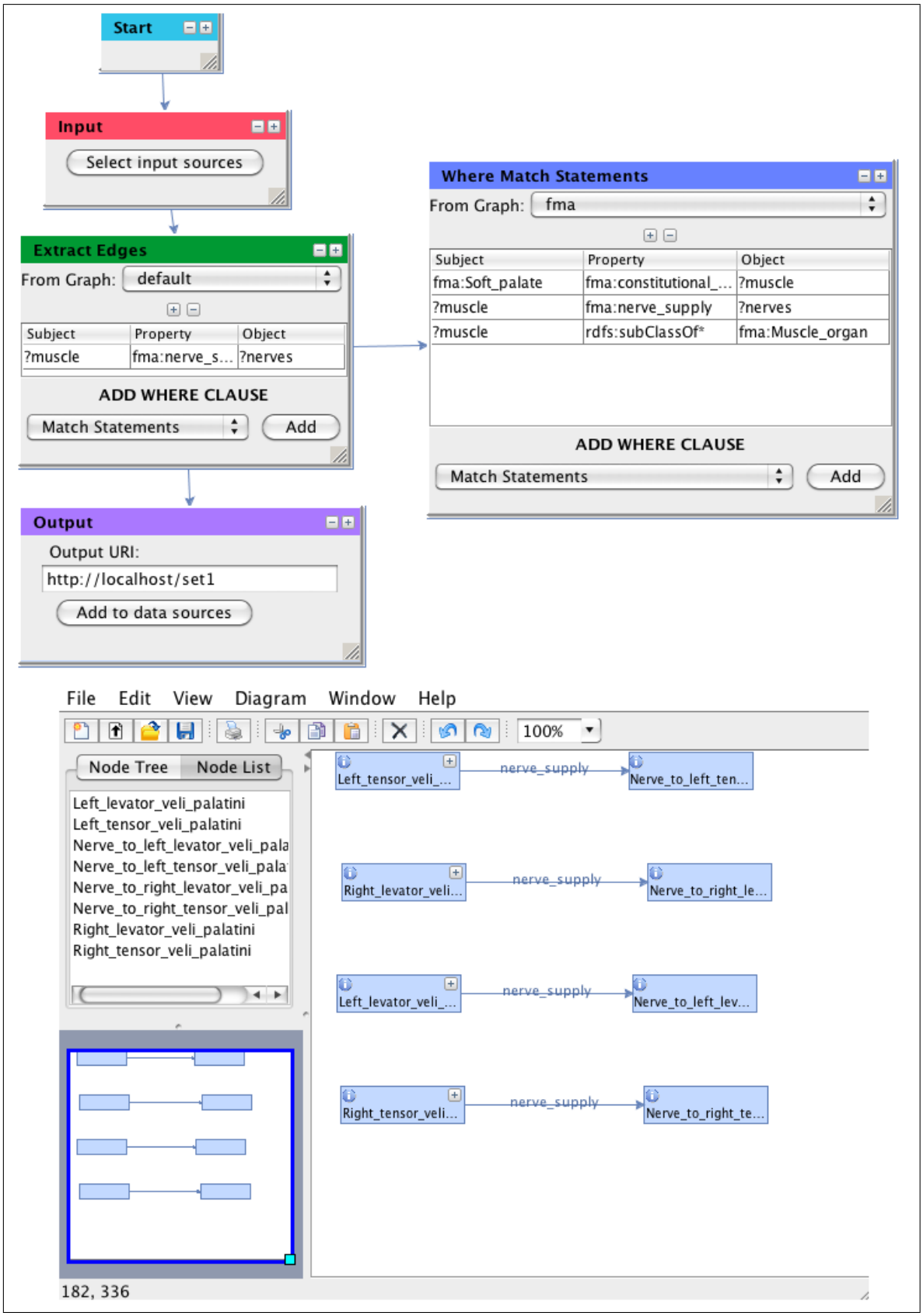


FIGURE 11. Graphical query workspace and results visualization for the soft palate query

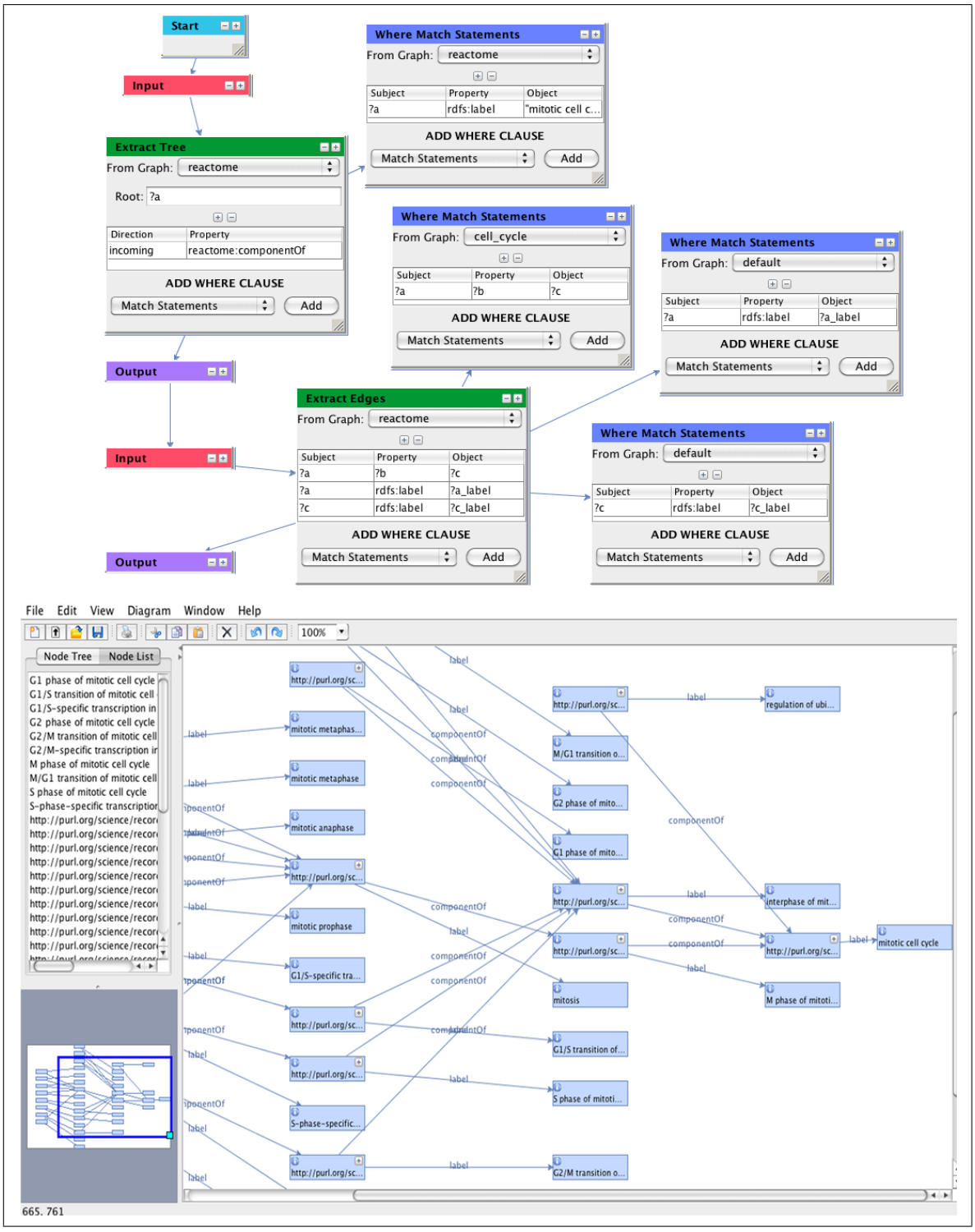


FIGURE 12. Graphical query workspace and results visualization for the mitotic cell cycle query

5.6.2. *Strategy.* In Reactome, nodes are not named with the English words corresponding to the structures that they represent but are instead named using numeric values. We thus begin the query by first finding the node which represents the *Mitotic Cell Cycle* by looking for the node with the *label* property *Mitotic Cell Cycle*. We then extract the tree of nodes that are components of this *Mitotic Cell Cycle* node and, for each of the nodes in this tree, we also extract the *label*.

Figure 12 shows the query-building workspace for the mitotic cell cycle query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 30.

5.7. **Organ spatial location.**

5.7.1. *Description.* From the FMA, extract the spatial information that can be used by image recognition software to automatically identify objects in medical images of the gastrointestinal tract. The result should only include the organs found in the gastrointestinal tract and their associated orientation and location properties.

5.7.2. *Strategy.* We start this query by extracting the set of nodes that are reachable from the *Gastrointestinal tract* following all of the FMA part relationships: *regional part*, *constitutional part* and *systemic part*. We also find the set of nodes that are derived from the *Organ* node. Following this, we compute the join between these two sets of nodes, to find those parts of the *Gastrointestinal tract* that are also derived from the *Organ* node. For this set of nodes we find the spatial properties, *orientation*, *continuous with*, *continuous with distally*, *continuous with proximally*, *attributed continuous with* and *contained in*. In addition to this, if it is the case that the node has the *orientation* property or the *attributed continuous with* property, we also want to include the edges that are derived from these properties.

Figure 13 shows the query-building workspace for the organ spatial location query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 31.

5.8. **NCI Thesaurus (NCIt) simplification.**

5.8.1. *Description.* This use case comes from [9]. The goal was to turn the complex chains of triples connecting classes, representing property restrictions, into direct class connections for browsing. The target was to produce a view which contains the same associations seen in the NCIt browser.

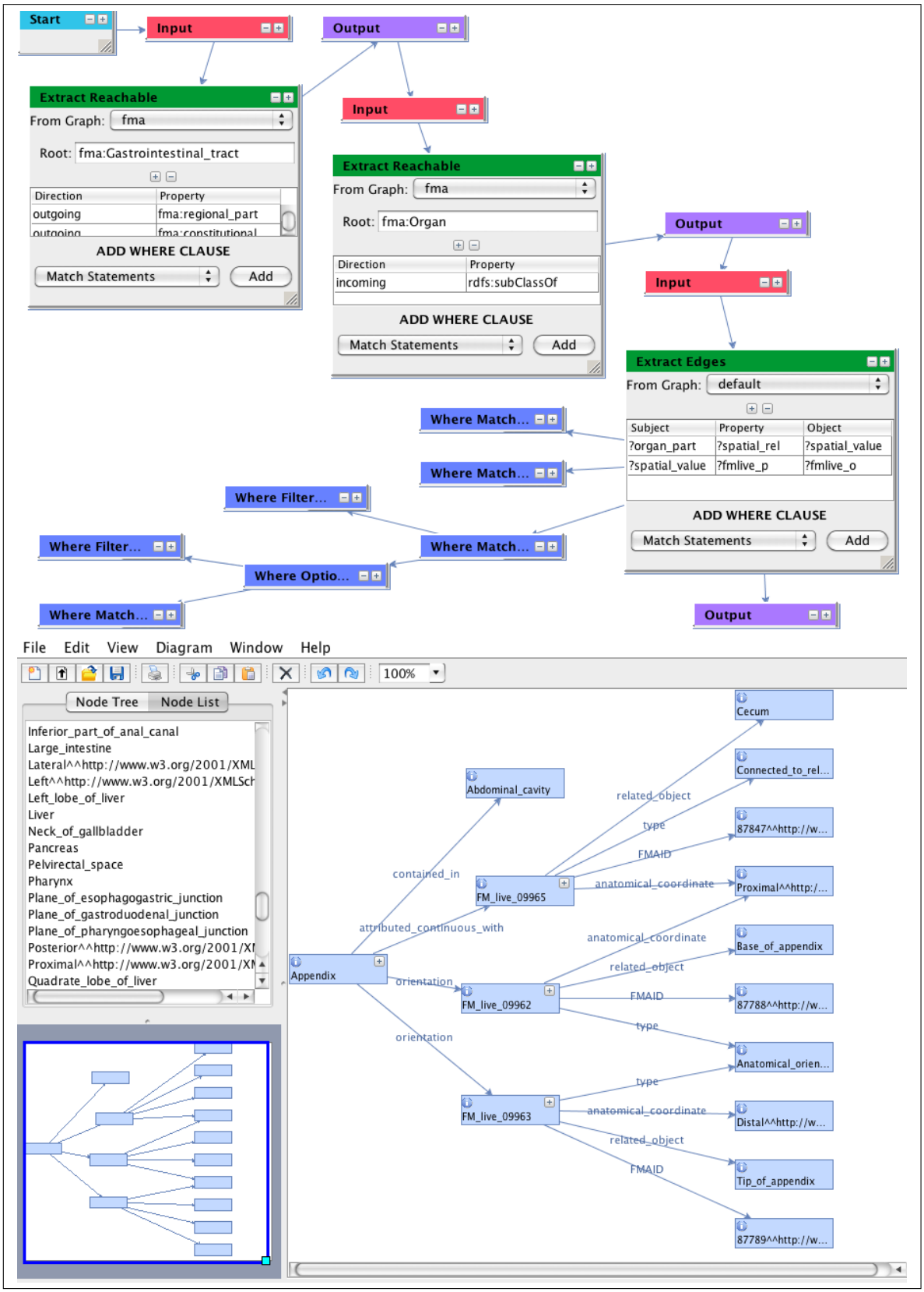


FIGURE 13. Graphical query workspace and results visualization for the organ spatial location query

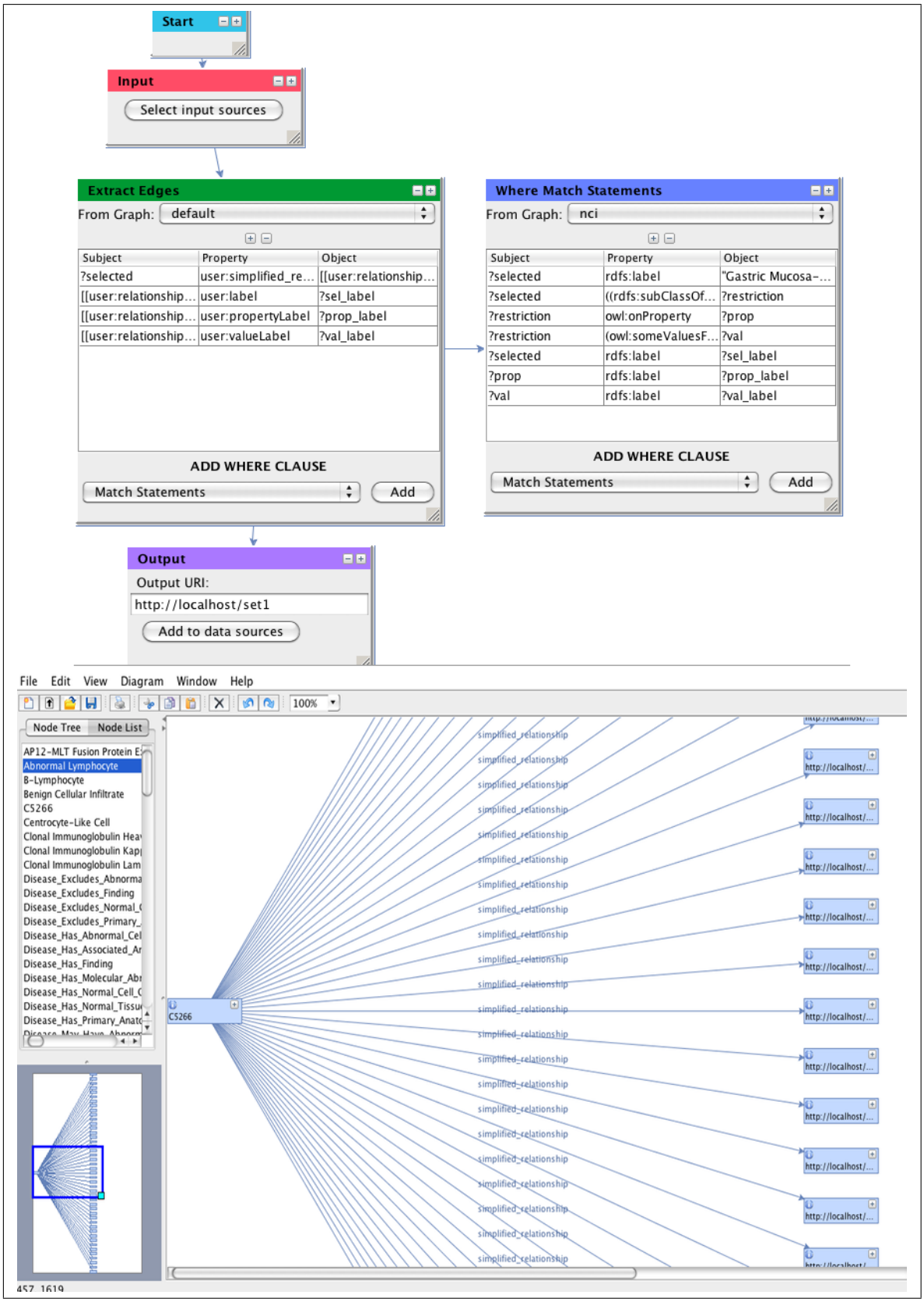


FIGURE 14. Graphical query workspace and results visualization for the NCIt simplification query

5.8.2. *Strategy.* The first step in this query is to find the node that corresponds to the *Gastric Mucosa-Associated Lymphoid Tissue Lymphoma*. We do this by finding which node has the *label* property *Gastric Mucosa-Associated Lymphoid Tissue Lymphoma*. We then find the set of nodes that identify OWL restrictions on this node and, for each of these restrictions, we find the properties that have been restricted and the allowed value sets for these properties. Since, in the NCIt, the nodes that represent the restricted properties and allowable values are not readable English descriptions, we also find the *label* properties for the node, restricted properties, and property values. The output of the query is a set of new triples that are created using the readable *label* properties of the node, the properties that have restrictions and their value sets.

Figure 14 shows the query-building workspace for the NCIt simplification query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 32.

5.9. **Blood contained in the heart.**

5.9.1. *Description.* From the FMA, generate a graph, to be used by physiology modelers, representing portions of blood contained in the heart. The results include heart parts and the blood portions they contain, where contains is a modied definition than that used by the FMA. In the FMA, only spaces are allowed in the domain of the contains property; in this use case, if a space contains blood, the structures that it is a part of should also be said to contain blood.

5.9.2. *Strategy.* We begin this query by starting at the *Heart* node and finding all of the *regional part* properties and *constitutional part* properties, and renaming these to simply *part* properties. For each of these *part* properties, we also find their *part* properties, and add these as *part* properties of the original *Heart* node. For example, if the *Heart* node has a part *Left heart*, and *Left heart* has a part *Left ventricle*, we add *Left ventricle* as a part of *Heart*. We continue this process recursively for all the parts of parts of the *Heart* until we reach the leaves of the graph. Now we need to find the parts of the *Heart* that contain blood. As mentioned above, in the FMA, only spaces are allowed to have the *contains* property, but we want to modify this so that heart parts which are not spaces may also have the *contains* property. To do this, we look for those heart parts which have the *contains* property and, for each of these, add the *contains* property to all of the nodes higher up in the tree of *Heart* parts. For example, if the *Heart* has the part *Left ventricle*, and the *Left ventricle* contains *blood*, we add the triple *Heart* contains *blood*.

Figure 15 shows the query-building workspace for the blood contained in heart query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 33.

5.10. **Biosimulation model editor.**

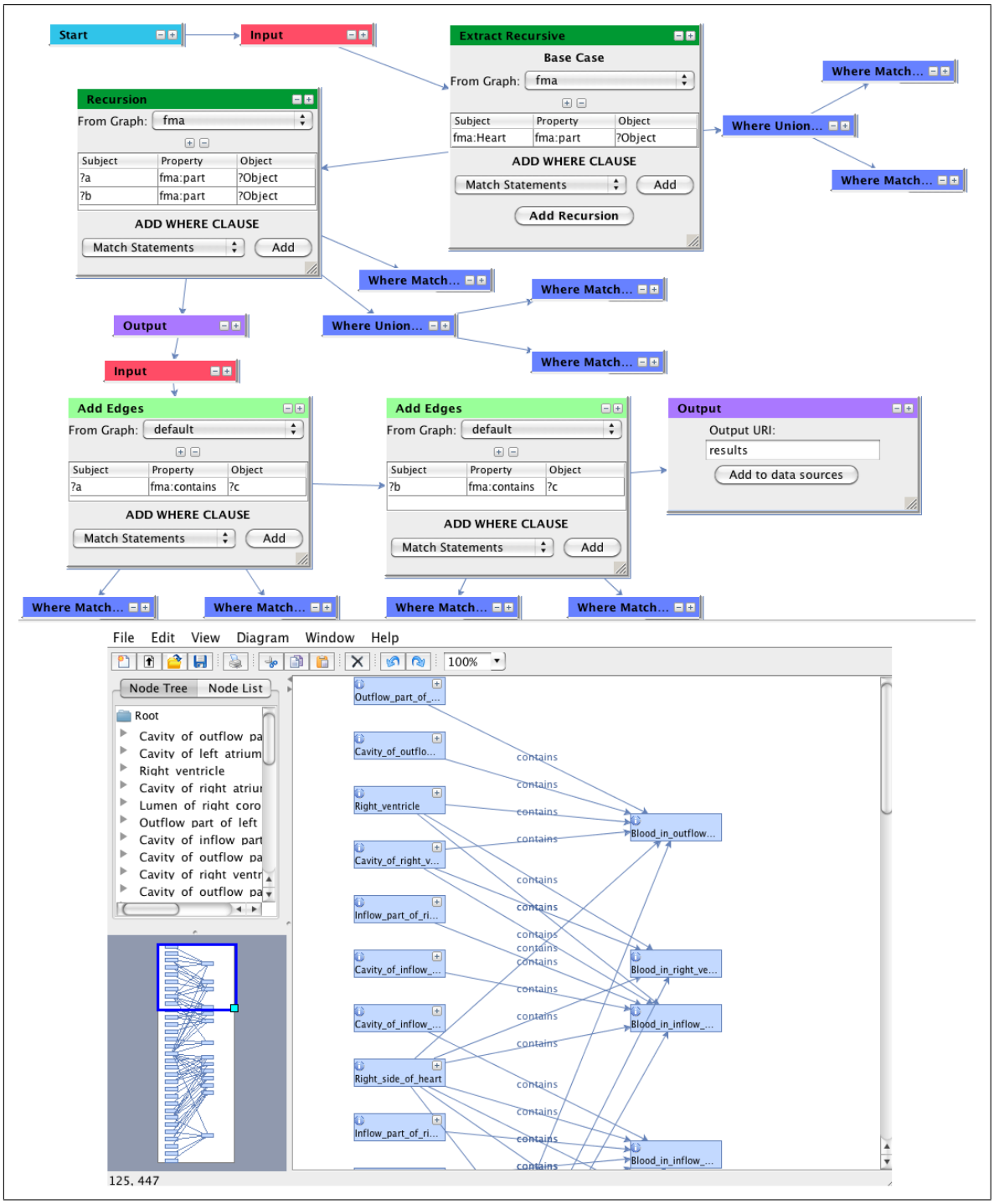


FIGURE 15. Graphical query workspace and results visualization for the blood contained in the heart query

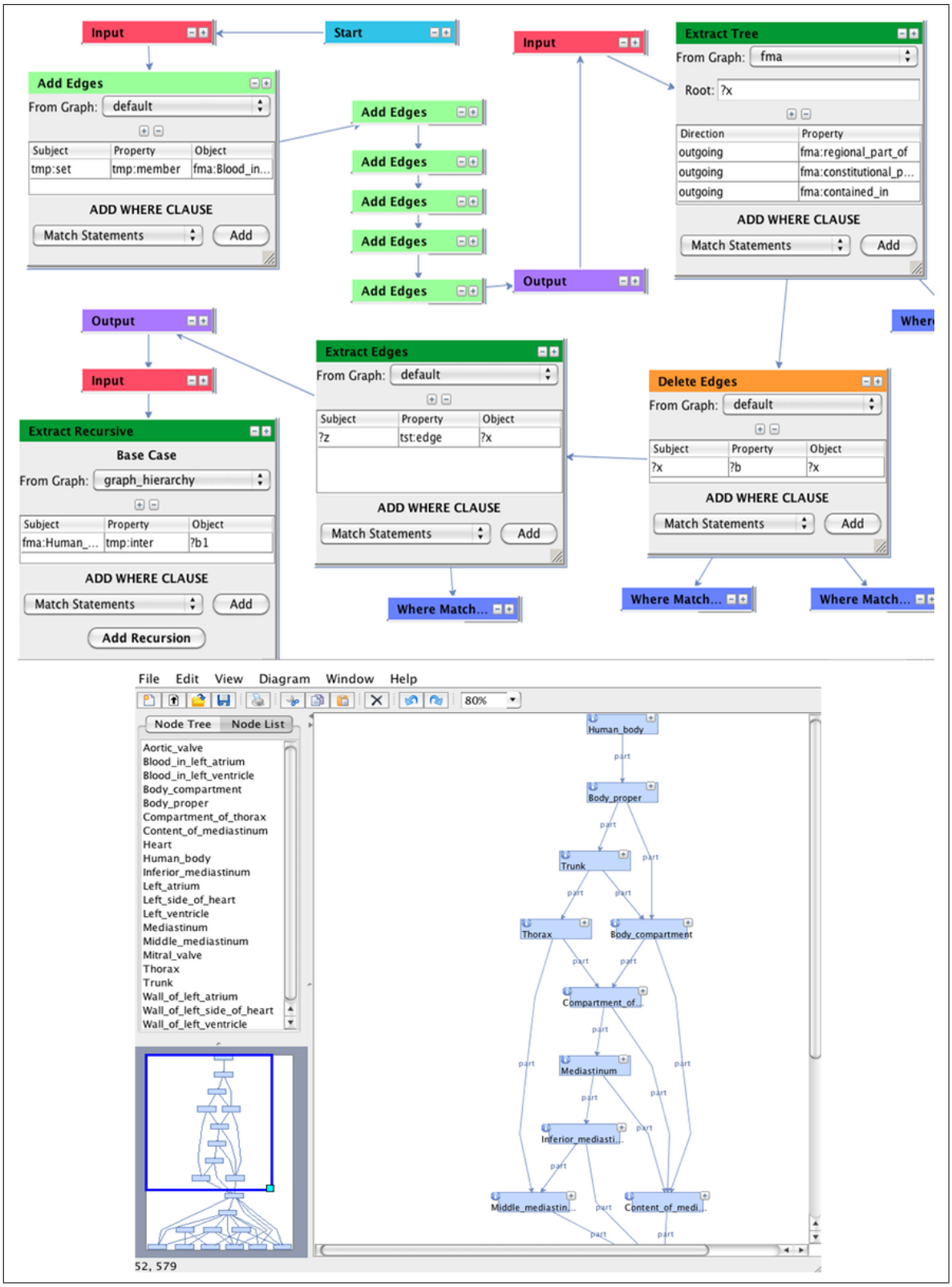


FIGURE 16. Graphical query workspace and results visualization for the biosimulation model editor query

5.10.1. *Description.* From the FMA, identify the relationship graph between a set of concepts; the graph is displayed by a biosimulation model editing tool. The application displays the relationships between the concepts as a restructured, pared-down hierarchy; the only nodes that should remain in the hierarchy are those under which there is a divergence between concepts. The result changes as new concepts are added and removed from the set of interest. As the user selects concepts within the hierarchy, the application queries additional properties from the FMA and displays it.

5.10.2. *Strategy.* We begin this query by creating a new graph that contains only six concepts: *Blood in left ventricle*, *Mitral valve*, *Wall of left ventricle*, *Aortic valve*, *Blood in left atrium* and *Wall of left atrium*. Then, for each of these six concepts, we extract from the FMA the tree of nodes created by following the *regional part of* property, the *constitutional part of* property and the *contained in* property. This will produce six trees of nodes starting from the leaves of the FMA and progressing to the six original concepts. Since we are not interested in the property names, we replace them with the simple property name *edge*, and then reverse the directions of all the edges, so that the trees of nodes now progress from the root node to the leaf nodes. We now wish to collapse the trees of nodes and only keep those nodes under which there is a divergence between concepts. To do this, we start at the *Human body* node in our extracted trees and get all of the child nodes connected to the *Human body* node. For each child node, there are three possible cases: the node has one child, more than one child, or zero children. If it has one child, then there is no divergence between the concepts and so we do not include this node in the output, but we do recursively examine it's child nodes. If it has more than one child (and these child nodes are not the same) then there is a divergence between the concepts so we keep them for inclusion in the output and recursively examine their children. If the node has zero children, it is a leaf node and so we include it in the output.

Figure 16 shows the query-building workspace for the biosimulation model editor query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 34.

5.11. **Blood fluid properties.**

5.11.1. *Description.* Combine information from two independent ontologies (FMA and Ontology of Physics for Biology (OPB)) to create new information for a biosimulation model editing application. Properties of fluids defined in the FMA should be combined with the kinetic properties of fluids defined in the OPB. For example, concepts like arterial blood and capillary blood in the FMA can be combined with flow, pressure, viscosity in the OPB, resulting in new concepts like arterial blood flow, capillary blood viscosity, etc. The result contains the newly created resources and their properties which can be used to annotate computational models of physiology.

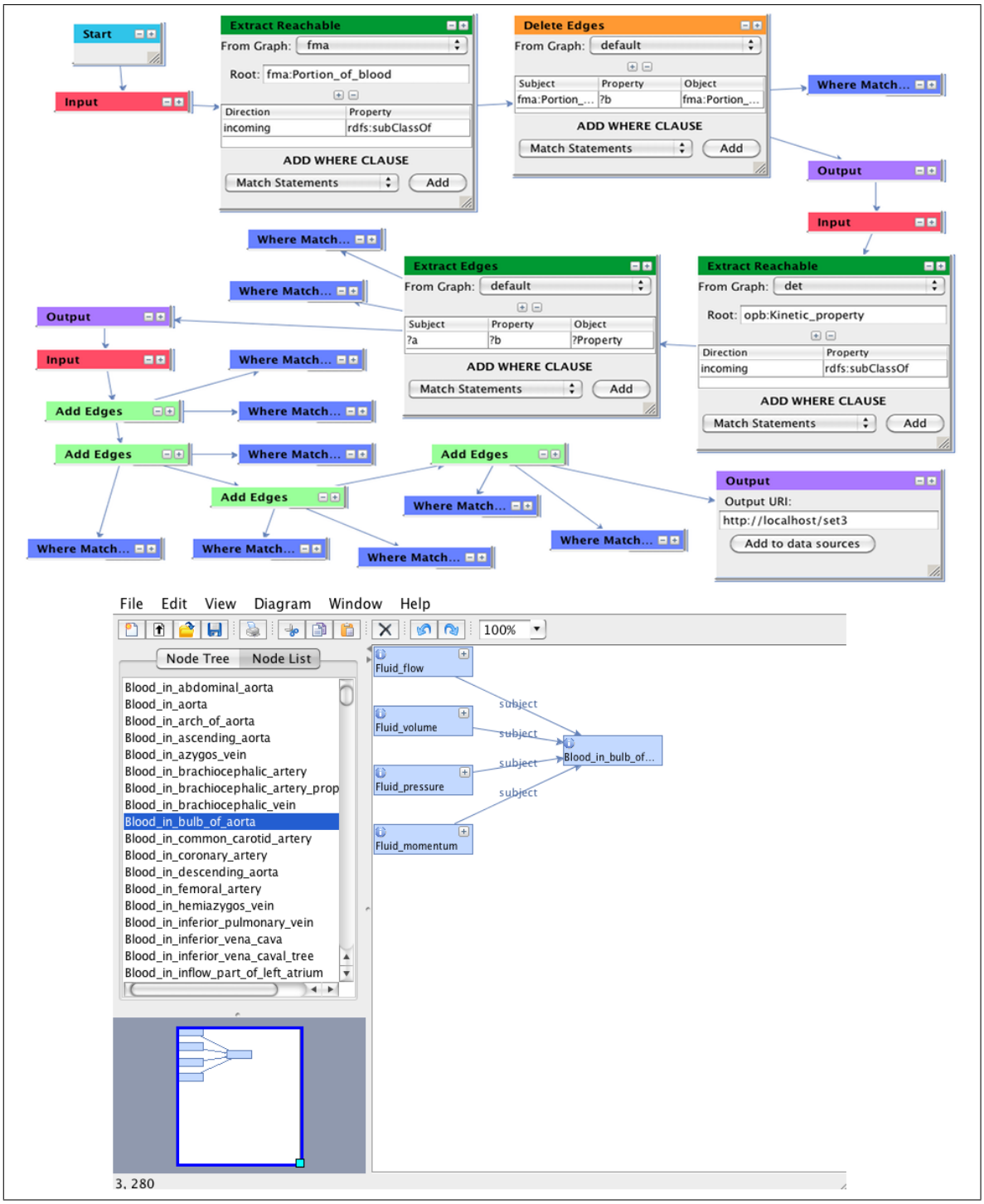


FIGURE 17. Graphical query workspace and results visualization for the blood fluid properties query

5.11.2. *Strategy.* The first step in this query is to extract the set of nodes from the FMA that reach *Portion of blood* through the *subclass of* hierarchy. Since this operation may potentially produce nodes with edges to themselves, for example, *Portion of blood* is a *subclass of Portion of blood*, we delete these. The next step is to extract the set of nodes from the OPB that are both *subclasses of* the *Kinetic property* node and also have the domain class *Fluid domain*. Finally, we create new nodes which combine the extracted FMA nodes with the extracted OPB nodes. For each new node, we add four child nodes, a combined FMA/OPB statement node, a *Portion of blood* node, a *has property* node and a node with the relevant OPB *Kinetic property*.

Figure 17 shows the query-building workspace for the blood fluid properties query along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 35.

5.12. **Radiologist liver ontology.**

5.12.1. *Description.* From the FMA, generate a sub-ontology to be used by an application for annotating medical images of the liver. The sub-ontology contains all of the visible parts of the liver and their associated superclass hierarchy; no property other than the superclass relationship should be included. Modify the structure of the subclass hierarchy to remove the concepts *Cavitated organ* and *Solid organ*.

5.12.2. *Strategy.* We start by extracting the trees of nodes from the FMA that are derived from the *Organ* node and the *Cardinal organ part* node, and union these trees. This will produce the set of structures that are large enough to be visible on a radiologist's image. The next step is to extract all of the parts of the *Liver* node, including *regional part*, *constitutional part* and *systemic part*. Now we compute a join between the liver parts and the visible organ structures. If a node is part of the *Liver* and is also an organ structure, we extract it along with the relevant part relationship. Additionally, since the *Liver* node may not have itself as one of its parts, but is visible on a radiology image, we make sure to add the *Liver* node to the set of extracted edges. Now, in order to ensure that the output of the view is still a valid ontology, we extract the *subclass of* hierarchy for all of the liver parts that are also organ structures. Finally, we simply replace the concepts *Cavitated organ* and *Solid organ* with *Organ*.

Figure 18 shows the query-building workspace for the radiologist liver ontology along with a visualization of the query results. The corresponding automatically generated IML query may be found in Appendix B, Figure 36.

6. Evaluation and Discussion

Thus far, we have primarily evaluated VIQUEN through an analysis of the complexity and accuracy of the queries that it can compose. Of the eight original view definition use

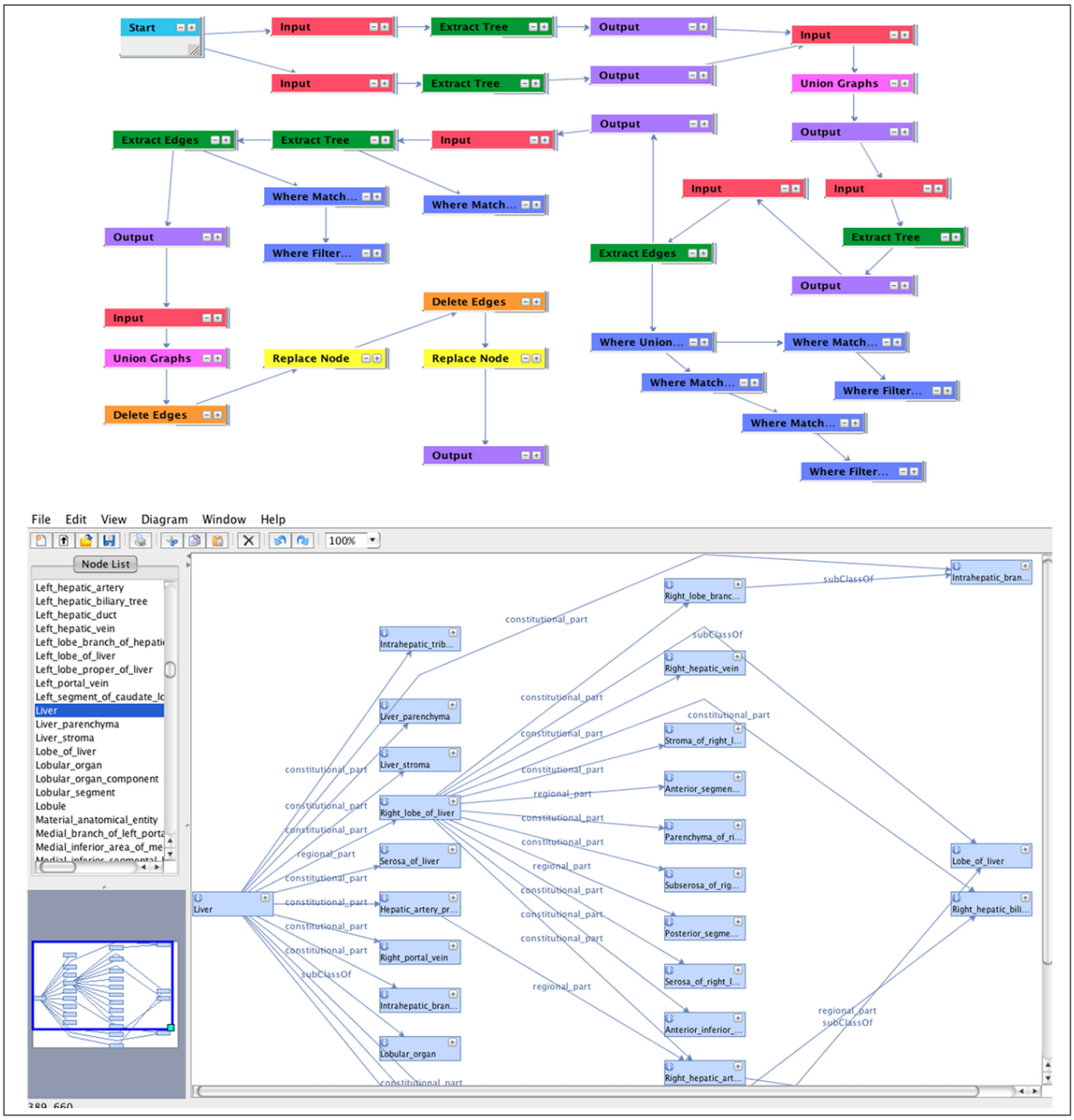


FIGURE 18. Graphical query workspace and results visualization for the radiologist liver ontology

case queries presented in [29], we have successfully managed to express seven of them. We did not attempt to create the eighth query, the NeuroFMA ontology, since we lacked the original handwritten IML query that would be required in order to evaluate the query

produced by VIQUEN. For the seven view definition queries we did compose, we used a modified version of RDFSync [18] to compare VIQUEN's RDF output with the output generated by both the original handwritten IML query and the original vSPARQL query presented in [29], and found them to be semantically equivalent. This clearly indicates that VIQUEN is capable of successfully expressing sophisticated, real-world semantic web queries.

Another of the strengths of our system is that it may be used as an educative tool for people who wish to learn more about the IML query language as well as the RDF data sets available on the semantic web. As discussed in the query-building environment section of this paper, VIQUEN makes it clear to the user which parameters are necessary for each high-level query operation. It also provides the user with an opportunity to examine the generated IML query and see how those parameters fit together to formulate the query. By adjusting the query parameters and examining the changes to the generated query, the user can quickly learn how to formulate queries in IML. Furthermore, if the user wishes to formulate a query but does not know the exact relationships that exist in the data set, they can compose a smaller query to obtain this information. For example, in the FMA, if a user wishes to find all of the *part* relationships of the *Liver* concept, but does not know the exact names of the *part* relationships, she may compose a simple query to find all of the relationships that exist for the *Liver* concept, and by visualizing these will see that there are three relationships which refer to parts: *regional part*, *constitutional part* and *systemic part*. Using these three relationships, she can now compose a query that retrieves all the different parts of the *Liver*. Furthermore, if she subsequently wants to obtain all of the parts of any other anatomical structure, she may now use the same three different *part* relationships.

In many cases, a single query can be expressed in several different ways and with varying levels of complexity. The exact query specification will depend on how familiar the user is with the data set and the query language, and on how much thought has gone into defining the query. It is important to note that while graphical systems like VIQUEN can make it easier for people to express queries in efficient ways, they cannot ensure that they do so. In the same way that sophisticated Integrated Development Environments (IDE's) cannot prevent software developers from writing bad code, the visual query engine cannot prevent end-users from writing bad queries.

However, through our work on VIQUEN, we have realized that an interesting relationship exists between query complexity and ease of query formulation. We believe that VIQUEN will be a valuable tool for non-expert users who wish to be able to express relatively simple queries and explore the concepts contained in the data sets. However, if a query is extremely complex, expressing it successfully will require significant knowledge of the structure of the data sets being queried as well as how to relate the concepts in the data sets to one another. In these cases, it is likely that the query will need to be formulated by an expert, and that even if tools like VIQUEN exist to aid non-expert users in expressing such queries, they will not want to spend the time and effort to do so, and will request that an expert

compose the query for them. However, if expert users are the ones composing complex queries, they will already have significant knowledge of semantic web query languages, and may find it easier to write the query by hand rather than using a visual query system like VIQUEN. Thus, we hypothesize that graphical query tools like VIQUEN may end up being most useful in helping non-expert users to compose more simple queries, rather than in formulating highly complex queries. We recognize that further research is essential in order to determine exactly who the end-users of tools such as VIQUEN are, and the type of queries that they wish to be able to express.

We acknowledge that VIQUEN currently has several limitations. Firstly, we have not, at the present time, fully evaluated the system with respect to a particular user group. Controlled empirical analyses and a full usability study are required in order to fully evaluate the system with respect to user satisfaction and task performance criteria. This is discussed in greater detail in the next section of this paper.

In addition to this, in order to compose meaningful queries, users are required to know the structure of the ontologies relatively well. In contrast to working with relational databases, which have a predefined schema, different concepts in RDF data sets may have different sets of properties associated with them. For example, in the FMA, the *Brain* concept may have several *regional part* properties, but not every concept in the FMA will have *regional part* properties. Thus, in order to formulate queries, users must have an understanding of which properties a particular concept is likely to have. VIQUEN attempts to help the user in this regard by providing drop down menus of common concepts and properties. However, sophisticated semantic web queries may utilize many large data sets, each of which may have millions of different concepts and properties. The provision of an ontology browser which enables users to select concepts out of different ontologies would be a useful addition to the system. However, locating a particular concept in the browser would still be difficult if the user did not already have good knowledge of the ontology and its structure.

Furthermore, since VIQUEN has been based on the semantic web query language IML, it has inherited some of the complexities associated with composing sophisticated semantic web queries. Specifically, the system requires users to have an understanding of how to do variable bindings using *Where* clauses. For example, in the *soft palate* query mentioned in the previous section of this paper, we first extract all of the concepts that are reachable from the *muscle organ* concept. When, in the next subquery block, we wish to refer to these reachable concepts, we must identify them using a variable which is defined in the *Where* clause of the subquery. In this case we may choose to name the variable *?muscles*. Now, when we refer to the *?muscles* variable in another part of the subquery, such as in finding the *nerve supply* for the *?muscles*, the system will bind *?muscles* variable to the relevant set of nodes that are reachable from *muscle organ*. The concept of variable bindings may be complicated for novice users to understand. Methods for how to make this part of the query formulation easier, possibly by automating the process of defining *Where* clauses, would be an interesting area for future research.

7. Future Work

The goal of our work has been to explore methods which make it easier for non-technical users to write semantic web queries and visualize the results of these queries, and has highlighted the need for further research in a number of areas. Firstly, as mentioned in the evaluation section of this paper, controlled empirical analyses and a full usability study are required in order to fully evaluate the system with respect to task performance and user satisfaction criteria. We anticipate that such a study would involve identifying a set of potential users and guiding them through several tutorials designed to show them how to use the system. We would then develop a number of specific tasks that would be completed by each user, including query specification, execution and navigation of the resulting RDF graph. We would evaluate user performance on each of these tasks as well as gather qualitative feedback on the overall user experience.

There are also a number of potential features that, if implemented, would improve the system. VIQUEN currently allows users to select concepts either by typing in the concept name, or by selecting it from a list of common concepts provided in the form of drop-down menus. Since a query may potentially make use of a number of data sets, and each data set may have millions of concepts, providing a comprehensive list of potential concepts in a down-down menu would not be user-friendly. The implementation of an ontology browser that would allow users to navigate different ontologies and select concepts would be a better solution. Additionally, some method of searching within the ontology browser would be required, since users may not know the exact location of a particular concept.

The visualization component of the system currently processes and holds the entire results graph in memory. However, if the query is very large, it may not be possible to keep the entire graph in memory at one time and, in these cases, VIQUEN will fail. Further research into methods for lazy evaluation of queries is necessary in order to be able to process and visualize portions of the results graph dynamically and on demand. Furthermore, while VIQUEN displays the generated IML query to the user, it does not permit editing of the generated query. If the user wishes to edit the query, she must return to the query-building environment to do so. Allowing the user to edit the IML query, and propagating these changes back to the graphical query would be an extremely useful addition to the system. Additionally, VIQUEN has been built on top of the IML query language and queries over RDF data sets. As such, it operates on the graph level representation of the data set and does not take advantage of the semantics of the data. The system would be more powerful if it could be modified so as to have the option of taking the semantics of the data into account.

Finally, we believe that further research into the usability of all the semantic web query languages is crucial in order to allow the widespread adoption of the rich data sets and query languages available on the semantic web. We hypothesize that the users of semantic web query languages fall into several different classes, ranging from novice to expert, and that the types of queries that each class of users wants to express will vary significantly in

complexity. Furthermore, it is likely that there will be users who are highly knowledgeable in the biomedical domain, and therefore wish to compose complex queries, but who lack the technical expertise to do so. There will also be technically competent users, who are capable of writing complex queries, but who lack the requisite biomedical knowledge of the underlying data sets. A study which aims to identify these different classes of users, along with the types of query that they are interested in writing, would be invaluable to the semantic web community and would allow the development of tools tailored to meet the needs of a particular user group.

8. Conclusion

We have presented VIQUEN, a graphical tool for semantic query construction, execution and visualization that is based on the IML data flow graph transformation language for manipulating RDF data. We have evaluated VIQUEN through the complexity of the queries that can be expressed using the system, including a number of real view definition queries. However, a full usability study is required in order to fully evaluate the system with respect to user satisfaction and task performance. Our work on VIQUEN represents an important step forward in making the sophisticated and expressive RDF query languages available for the semantic web more accessible to non-technical users. It also highlights the need for further research to determine who the end-users of such query languages are, the type of queries that they wish to be able to express, and additional methods which could support users throughout the process of query formulation.

9. Acknowledgements

This work was funded by NIH grant HL087706.

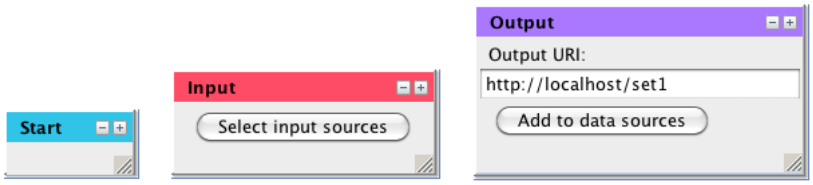
References

[1] Reactome a curated knowledgebase of biological pathways. http://www.reactome.org.
[2] G Alder. Jgraph. http://www.jgraph.com/.
[3] J Borsje and H Embregts. Graphical query composition and natural language processing in an rdf visualization interface. Bachelor Thesis, Erasmus School of Economics and Business Economics, Rotterdam, Erasmus University, 2006.
[4] T Catarci, M.F Costabile, S Levialdi, and C Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8:215–260, 1997.
[5] T Catarci, P Dongilli, T D Mascio, E Franconi, G Santucci, and S Tessaris. An ontology based visual tool for query formulation support. In *16th European Conference on Artificial Intelligence*, 2004.
[6] D L Cook, J L V Mejino, M L Neal, and J H Gennari. Bridging biological ontologies and biosimulation: The ontology of physics for biology. In *Proceedings, American Medical Informatics Association Fall Symposium*, pages 136–140, 2008.
[7] L Deligiannidis, K Kochut, and A Sheth. Rdf data exploration and visualization. In *Proceedings of the first workshop on CyberInfrastructure 2007*, pages 39–46. ACM Press, 2007.
[8] L T Detwiler. Query manager. http://axon.biostr.washington.edu:8080/QueryManager/QueryManager.html, 2010.

[9] L T Detwiler, D Suciu, and J F Brinkley. Regular paths in sparql: Querying the nci thesaurus. In *Proceedings, American Medical Informatics Association Fall Symposium*, pages 161–165, 2008.
[10] A Fadhil and V Haarslev. Ontovql: A graphical query language for owl ontologies. In *International Workshop on Description Logics (DL-2007)*, 2007.
[11] S M Falconer, C Callendar, and M Storey. Flexviz: Visualizing biomedical ontologies on the web.
[12] Jena A Semantic Web Framework for Java. http://jena.sourceforge.net.
[13] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
[14] S Goyal and R Westenthaler. Rdf gravity (rdf graph visualization tool). http://semweb.salzburgresearch.at/apps/rdf-gravity/, 2004.
[15] B Hu, S Dasmahapatra, P Lewis, and N Shadbolt. Ontology-based medical image annotation with description logics. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, pages 77–82, 2003.
[16] IsaViz. A visual authoring tool for rdf. http://www.w3.org/2001/11/IsaViz/, 2001-2006.
[17] G Marquet, O Dameron, S Saikali, J Mosser, and A Burgun. Grading glioma tumors using owl-dl and nci thesaurus. In *AMIA Annual Symposium Proceedings*, pages 508–512, 2007.
[18] C Morbidoni, G Tummarello, O Erling, and R Bachmann-Gmr. Rdfsync: Efficient remote synchronization of rdf models. In *Proceedings, International Semantic Web Conference*, pages 537–551, 2007.
[19] NCBO. Bioportal. http://bioportal.bioontology.org/.
[20] NCIthesaurus. http://nciterms.nci.nih.gov.
[21] N F Noy and D L Rubin. Translating the foundational model of anatomy into owl. In *Web Semantics: Science, Services and Agents on the World Wide Web*, pages 133–136, 2008.
[22] OpenLink. isparql. http://demo.openlinksw.com/isparql/.
[23] Resource Description Framework (RDF). http://www.w3.org/RDF/.
[24] RDQL. A query language for rdf. http://www.w3.org/Submission/RDQL/, 2004.
[25] C Rosse and JVL Mejino. A reference ontology for biomedical informatics: the foundational model of anatomy. *Journal of Biomedical Informatics*, pages 478–500, 2003.
[26] P Shannon, A Markiel, O Ozier, N S Baliga, J T Wang, D Ramage, N Amin, B Schwikowski, and Ideker T. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Research*, pages 2498–2504, 2003.
[27] L G Shapiro, E Chung, L T Detwiler, J L V Mejino, A V Agoncillo, J F Brinkley, and C Rosse. Processes and problems in the formative evaluation of an interface to the foundational model of anatomy knowledge base. *Journal of the American Medical Informatics Association*, 1:35–46, 2005.
[28] M Shaw, L T Detwiler, N Noy, J Brinkley, and D Suciu. Intermediate language. http://trac.biostr.washington.edu/trac/wiki/IntermediateLanguage/.
[29] M Shaw, L T Detwiler, N Noy, J Brinkley, and D Suciu. vsparql: A view definition language for the semantic web. *Journal of Biomedical Informatics*, 2010.
[30] SIMILE. Welkin. http://simile.mit.edu/welkin/, 2004-2005.
[31] Andrea Splendiani. Semantic browsing of pathway ontologies and biological networks with rdfscape (working paper). In *Managing and Mining Genome Information: Frontiers in Bioinformatics*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
[32] M Storey, N F Noy, M Musen, C Best, R Fergerson, and N Ernst. Jambalaya: an interactive environment for exploring ontologies. In *Proceedings of the 7th international conference on Intelligent user interfaces*, pages 239–239. ACM, 2002.
[33] J Wang, J Lu, Y Zhang, Z Miao, and B Zhou. Integrating heterogeneous data sources using ontology. *Journal of Software*, 4(8):843–850, 2009.

Appendix A. VIQUEN query-building operations

The VIQUEN query-building workspace has been designed to take advantage of the data flow graph transformation style of IML. Each high-level query operation is represented in its own visual node. Nodes of the same type are the same color for easy identification. The visual nodes are then chained together, using directed edges, to compose the entire query.



(a) Start operation (b) Input operation (c) Output operation

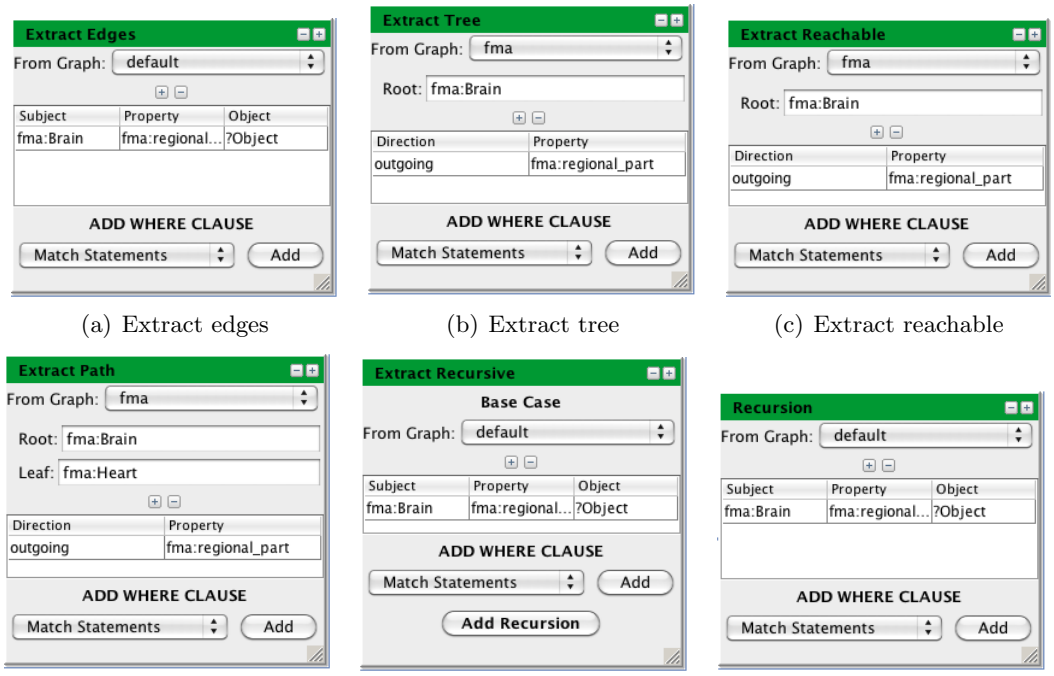
Figure 19. Basic (a) Start, (b) Input and (c) Output operations

A.1. **Start operation.** A query must begin with a Start operation, Figure 19(a), which indicates the point from which the system will start to compile the query. By positioning the Start operation appropriately, different chunks of the query may be executed individually before combining them into a larger query. After the Start operation, the query is defined by adding one or more subquery blocks to the workspace.

A.2. **Input operation.** Each subquery block begins with an Input operation, Figure 19(b), which defines the data sources to be used as input to the query. Clicking on the "Select input sources" button will bring up a list of available data sources which may be selected for inclusion in the query.

A.3. **Output operation.** A subquery block must end with an Output operation, Figure 19(c), which specifies the output graph for the block. This output graph may easily be added to the list of potential input data sources by clicking on the "Add to data sources" button and specifying a name for the output graph.

A.4. **Extract operations.** These are a set of operations provided specifically for extracting information from an RDF graph. The operations include extract edges, extract tree, extract reachable, extract path, and extract recursive.



(a) Extract edges (b) Extract tree (c) Extract reachable

(d) Extract path (e) Extract recursive: base case (f) Extract recursive: recursion

FIGURE 20. Extract operations: (a) Extract edges, **??** Extract tree, (c) Extract reachable, (d) Extract path, (e) Extract recursive (base case), and (f) Extract recursive (recursion)

A.4.1. *Extract Edges.* This operation, shown in Figure 20(a), specifies a subset of RDF triples that should be extracted from the data set indicated in the *From Graph* field. The table defines the triple pattern to be found. Triples may be added and removed from the table using the buttons provided. If the specified triple pattern is found in the input graph, the triples are added to the output of the operation. Variables may be specified in the table using a *?* symbol, for example *?Object*. The variables are then bound to sets of RDF triples by adding a *Where* clause.

A.4.2. *Extract Tree.* This operation, shown in Figure 20(b), allows a user to indicate a root node and a set of properties that should be recursively followed to extract a subgraph of the input. This operation will maintain the structure of the extracted subgraph. *From Graph* indicates the graph that should be used to construct the tree. *Root* indicates the node in the graph to be used as the root of the tree. The table specifies the properties or edges that should be followed, and the direction in which to follow them: outgoing (from the root node), incoming (to the root node) or both.

A.4.3. *Extract Reachable.* This operation, shown in Figure 20(c), allows a user to indicate a root node and a set of properties that should be recursively followed to identify the set of nodes that can be reached by traversing those properties. The input fields are identical to those of the *Extract tree* operation described above. However, unlike *Extract Tree*, this operation does not maintain the structure of the extracted subgraph, but rather produces a flat list of the nodes that can be reached from the specified root node.

A.4.4. *Extract Path.* This operation, shown in Figure 20(d), allows a user to specify a root node, a leaf node and a list of properties, and returns the subgraph containing the path from the root to the leaf by recursively traversing the list of properties.

A.4.5. *Extract Recursive.* This operation provides general recursion mechanism allowing users to precisely specify the edges that they want to follow and the output that should be produced. The operation is broken down into two parts: a set of base cases, shown in Figure 20(e), and a set of recursive cases, shown in Figure 20(f). Each base case specifies a graph and a triple pattern that should be used to locate the base case RDF triples. The operation evaluates the base case and the results are added to the output *recursive* graph using set union. A recursive case may access both the incoming set of RDF graphs and the result set being produced, referenced by the graph name *recursive*. The operation evaluates each case and new results are added to the *recursive* graph using set union, and then the recursive cases are evaluated again and the results also added to *recursive*. This iteration process continues until a stable state is reached (no new results are added to the *recursive* graph.)

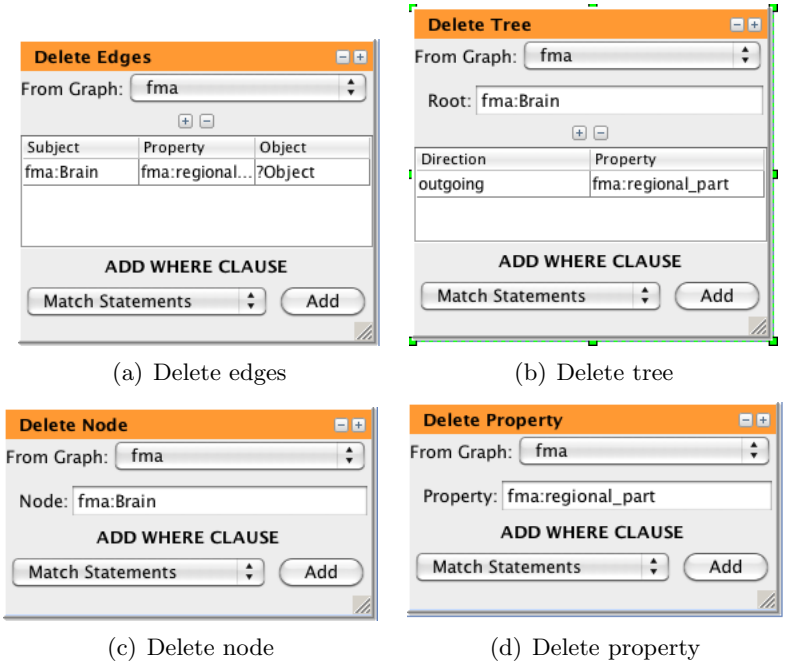
A.5. **Delete operations.** These are a set of operations for removing information from an RDF graph. Delete operations include delete edges, delete tree, delete node and delete property.

A.5.1. *Delete Edges.* This operation, shown in Figure 21(a), is syntactically the same as *Extract edges* but instead specifies a pattern for RDF triples that should be removed from the data set indicated in the *From Graph* field.

A.5.2. *Delete Tree.* This operation, shown in Figure 21(b), is syntactically the same as *Extract tree* but, rather than extracting a subtree, it removes the subtree from the graph.

A.5.3. *Delete Node.* This operation, shown in Figure 21(c), allows a user to indicate a specific node that should be deleted from the graph specified in the *From Graph* field. The *Node* field indicates the node to be deleted. All edges that contain the node either as a subject or as an object are deleted from the graph.

A.5.4. *Delete Property.* This operation, shown in Figure 21(d), allows a user to indicate a specific property that should be deleted from the graph specified in the *From Graph* field. All edges with the property specified in the *Property* field are deleted from the graph.



(a) Delete edges (b) Delete tree

(c) Delete node (d) Delete property

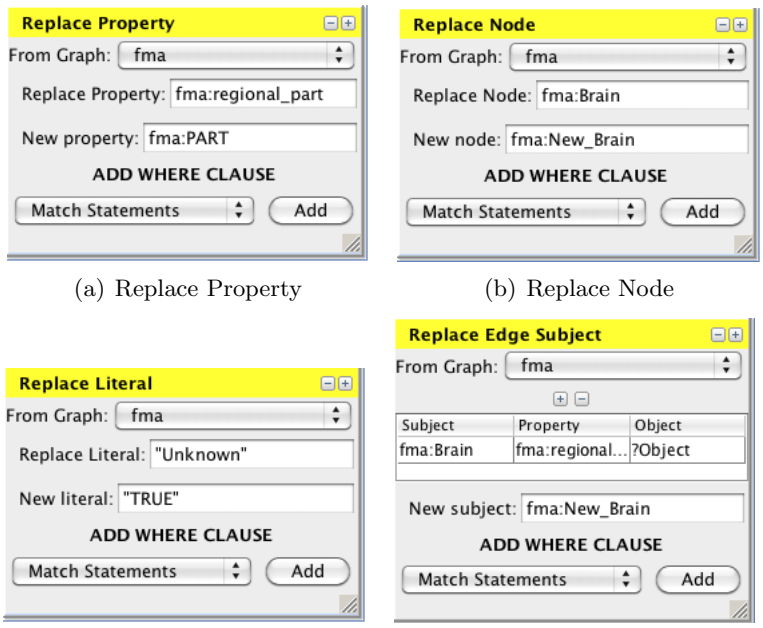
FIGURE 21. Delete operations: (a) Delete edges, (b) Delete tree, (c) Delete node and (d) Delete property

A.6. **Replace operations.** These are a set of operations for replacing information in an RDF graph. Replace operations include replace property, replace node, replace literal, and replace edge subject, replace edge property, replace edge object and replace edge literal).

A.6.1. *Replace Property.* This operation, shown in Figure 22(a), is used to replace all instances of a property occurring in the graph specified in the *From Graph* field. The *Replace Property* field indicates the name of the property that will be replaced, while the *New property* field indicates the name of the new property that will be the replacement. All other edges in the graph are unchanged and exist in the operation's output graph.

A.6.2. *Replace Node.* This operation, shown in Figure 22(b), is similar to *Replace property* but rather replaces a node that occurs in the graph specified in the *From Graph* field. The *Replace Node* field indicates the name of the node that will be replaced, while the *New Node* field indicates the name of the new node that will be the replacement. All other edges in the graph are unchanged and exist in the operation's output graph.

A.6.3. *Replace Literal.* This operation, shown in Figure 22(c), is similar to *Replace property* and *Replace node* but rather replaces a literal that occurs in the graph specified in the



(a) Replace Property (b) Replace Node

(c) Replace Literal (d) Replace Edge Subject

FIGURE 22. Replace operations: **??** Replace property, (b) Replace node, (c) Replace literal and (d) Replace Edge Subject

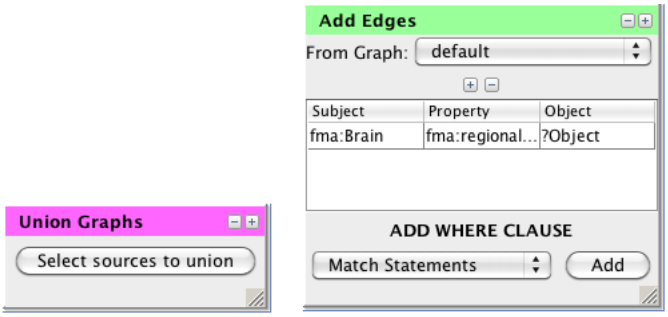
From Graph field. The *Replace Literal* field indicates the name of the literal that will be replaced, while the *New Literal* field indicates the name of the new literal that will be the replacement. All other edges in the graph are unchanged and exist in the operation's output graph.

A.6.4. *Replace Edge Subject, Property, Object and Literal.* The *Replace Edge Subject* operation, shown in Figure 22(d), is used to replace the subject of an RDF triple. The *From Graph* field indicates that graph in which to find the specified edges, while the table contains the triple pattern specifying the edges whose subjects will be replaced. The *New Subject* field indicates the replacement subject for the specified edges. All other edges in the input graph are unchanged and exist in the operation's output graph.

The *Replace Edge Property*, *Replace Edge Object* and *Replace Edge Literal* operations work in exactly the same way as the *Replace Edge Subject* operation, but rather replace the property, object or literal of the specified edges.

A.7. **Union operation.** This operation, shown in Figure 23(a), is used to combine information from two or more RDF graphs. Clicking on the *Select sources to union* button will

open a list of available data sources which may be selected for inclusion in the union operation. The operation produces the result of combining all of the selected graphs. Duplicate edges in multiple RDF graphs will only appear once in the result graph.



(a) Union operation (b) Add edges operation

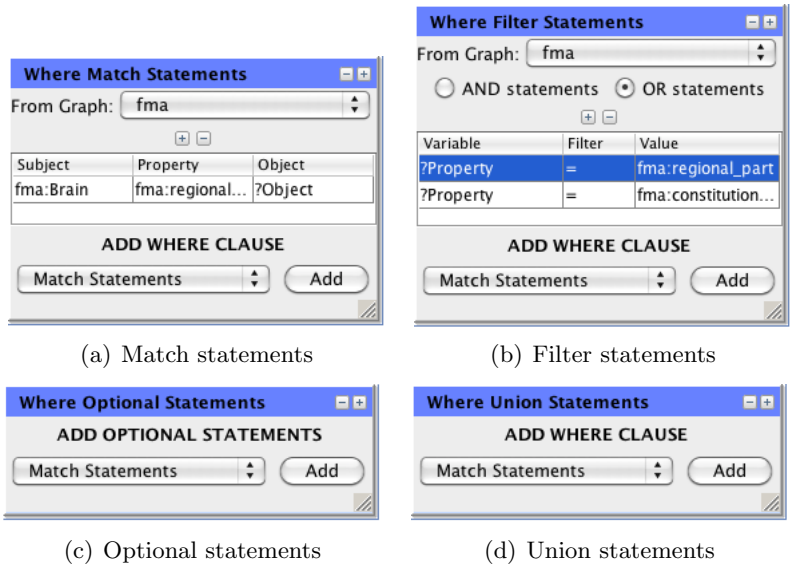
FIGURE 23. Additional operations: (a) Union operation and (b) Add operation

A.8. **Add Edges operation.** This operation, shown in Figure 23(b), is used to create new edges that will be added to the RDF graph specified in the *From Graph* field. All the edges in this graph are combined with the new edges in the output graph. The table indicates the triple pattern to be added.

A.9. **Where clause operations.** *Where* operations are used to bind sets of RDF triples to unknown variables. Variables are specified using a *?* symbol, for example *?property*. There are four different types of *Where* clause operations: *match statements*, *union statements*, *filter statements* and *optional statements*.

A.9.1. *Where match statements.* This operation, shown in Figure 24(a), allows a user to specify an RDF triple pattern containing variables to be matched to sets of RDF statements located in the graph specified in the *From Graph* field. The table defines the triple pattern to be found. If the triple pattern matches sets of triples in the specified graph, these triples are bound to the relevant variable and added as output for the operation.

A.9.2. *Where filter statements.* This operation, shown in Figure 24(b), tests values within a the graph. The *From Graph* field indicates the graph that will be used to evaluate the constraints. The *AND statements* and *OR statements* specify the logic used to combine multiple constraints. The table contains a list of constraints. The constraints must evaluate to TRUE in order for the pattern to match.



(a) Match statements (b) Filter statements

(c) Optional statements (d) Union statements

FIGURE 24. Where clause operations: (a) Match statements, (b) Filter Statements, (c) Optional statements and (d) Union statements

A.9.3. *Where optional statements.* This operation, shown in Figure 24(c), allows specified triple patterns to be made optional. Additional *Where* operations added from the *Optional Statements* operation will be added to the output if they exist, but will not make the query fail if they do not exist.

A.9.4. *Where union statements.* This operation, shown in Figure 24(d), allows multiple different triple patterns to be matched to the same variables. The operations to be included in the union are added from the *Union Statements* operation.

Appendix B. Generated IML for sample queries.

This appendix contains the automatically generated IML query for each of the sample queries presented in Section 5 of the paper.

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_EDGES { ?organ fma:contained_in fma:Abdominal_cavity } GRAPH <http://sig.biostr.washington.edu/fma3.0>
WHERE { GRAPH <http://sig.biostr.washington.edu/fma3.0> { ?organ fma:contained_in fma:Abdominal_cavity .  } . }
}
OUTPUT <http://localhost/set1>
```

Figure 25. Generated IML query for the abdominal cavity query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_TREE { fma:Left_frontal_lobe [ forward (fma:regional_part)  ]  } GRAPH <http://sig.biostr.washington.edu/fma3.0>
}
OUTPUT <http://localhost/set1>
```

Figure 26. Generated IML query for the left frontal lobe query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_TREE { fma:Lung [ forward (fma:regional_part) , forward (fma:constitutional_part) , forward (fma:systemic_part)  ]  }
GRAPH <http://sig.biostr.washington.edu/fma3.0>
REPLACE_PROPERTY fma:regional_part fma:part
REPLACE_PROPERTY fma:constitutional_part fma:part
REPLACE_PROPERTY fma:systemic_part fma:part
}
OUTPUT <http://localhost/set1>
```

Figure 27. Generated IML query for the lung parts query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX skull_parts: <http://localhost/set1#>
PREFIX face_parts: <http://localhost/set2#>
PREFIX parts: <http://localhost/set3#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_TREE { fma:Skull [ forward (fma:regional_part) , forward (fma:constitutional_part)  ]  }
GRAPH <http://sig.biostr.washington.edu/fma3.0>
}
OUTPUT <http://localhost/set1>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_TREE { fma:Face [ forward (fma:regional_part) , forward (fma:constitutional_part)  ]  }
GRAPH <http://sig.biostr.washington.edu/fma3.0>
}
OUTPUT <http://localhost/set2>

INPUT <http://localhost/set1>, <http://localhost/set2>
{
UNION_GRAPHS <http://localhost/set1>, <http://localhost/set2>
}
OUTPUT <http://localhost/set3>

INPUT <http://sig.biostr.washington.edu/fma3.0>, <http://localhost/set3>
{
EXTRACT_EDGES { ?Subject ?Property ?Object . ?Object rdfs:label ?label . ?Object fma:FMAID ?id }
WHERE { GRAPH <http://localhost/set3> { ?Subject ?Property ?Object .  } .
GRAPH <http://sig.biostr.washington.edu/fma3.0> { ?Object rdfs:label ?label . ?Object fma:FMAID ?id .  } . }
}
OUTPUT <http://localhost/set4>
```

FIGURE 28. Generated IML query for the craniofacial query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_EDGES { ?muscle fma:nerve_supply ?nerves } WHERE {
GRAPH <http://sig.biostr.washington.edu/fma3.0>
{ fma:Soft_palate fma:constitutional_part* ?muscle . ?muscle fma:nerve_supply ?nerves . ?muscle rdfs:subClassOf* fma:Muscle_organ .  } . }
}
OUTPUT <http://localhost/set1>
```

FIGURE 29. Generated IML query for the soft palate query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX reactome: <http://purl.org/science/ontology/reactome/>
PREFIX cell_cycle: <http://localhost/set1#>

INPUT <http://purl.org/science/ontology/reactome>
{
EXTRACT_TREE { ?a [ backward (reactome:componentOf)  ]  } GRAPH <http://purl.org/science/ontology/reactome>
WHERE { GRAPH <http://purl.org/science/ontology/reactome> { ?a rdfs:label "mitotic cell cycle" .  } . }
}
OUTPUT <http://localhost/set1>

INPUT <http://purl.org/science/ontology/reactome>, <http://localhost/set1>
{
EXTRACT_EDGES { ?a ?b ?c . ?a rdfs:label ?a_label . ?c rdfs:label ?c_label } GRAPH <http://purl.org/science/ontology/reactome>
WHERE { GRAPH <http://localhost/set1> { ?a ?b ?c .  } . ?a rdfs:label ?a_label . ?c rdfs:label ?c_label . }
}
OUTPUT <http://localhost/set2>
```

FIGURE 30. Generated IML query for the mitotic cell cycle query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX gi_parts: <gi_parts#>
PREFIX organ_classes: <organ_classes#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_REACHABLE { fma:Gastrointestinal_tract [ forward (fma:regional_part) , forward (fma:constitutional_part) , forward (fma:systemic_part)  ]  }
GRAPH <http://sig.biostr.washington.edu/fma3.0>
}
OUTPUT <gi_parts>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_REACHABLE { fma:Organ [ backward (rdfs:subClassOf)  ]  } GRAPH <http://sig.biostr.washington.edu/fma3.0>
}
OUTPUT <organ_classes>

INPUT <http://sig.biostr.washington.edu/fma3.0>, <gi_parts>, <organ_classes>
{
EXTRACT_EDGES { ?organ_part ?spatial_rel ?spatial_value . ?spatial_value ?fmlive_p ?fmlive_o }
WHERE { GRAPH <gi_parts> { ?a ?b ?organ_part .  } . GRAPH <organ_classes> { ?c ?d ?organ_part .  } .
GRAPH <http://sig.biostr.washington.edu/fma3.0> { ?organ_part ?spatial_rel ?spatial_value .
FILTER (  ( ( ?spatial_rel ) =( fma:orientation )  )  ||  ( ( ?spatial_rel ) =( fma:continuous_with )  )  ||
( ( ?spatial_rel ) =( fma:continuous_with_distally )  )  ||  ( ( ?spatial_rel ) =( fma:continuous_with_proximally )  )  ||
( ( ?spatial_rel ) =( fma:attributed_continuous_with )  )  ||  ( ( ?spatial_rel ) =( fma:contained_in )  )  ) .
OPTIONAL { FILTER (  ( ( ?spatial_rel ) =( fma:orientation )  )  ||  ( ( ?spatial_rel ) =( fma:attributed_continuous_with )  )  ) .
?spatial_value ?fmlive_p ?fmlive_o . } } . }
}
OUTPUT <results>
```

FIGURE 31. Generated IML query for the organ spatial location query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX nci: <http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl#>
PREFIX user: <http://localhost/userFriendlyNCIT#>
PREFIX gleen: <java:edu.washington.sig.gleen.>

INPUT <http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl>
{
EXTRACT_EDGES { ?selected user:simplified_relationship [[user:relationship(?sel_label, ?prop_label, $val_label)]] .
[[user:relationship(?sel_label,?prop_label,?val_label)]] user:label ?sel_label .
[[user:relationship(?sel_label,?prop_label,?val_label)]] user:propertyLabel ?prop_label .
[[user:relationship(?sel_label,?prop_label,?val_label)]] user:valueLabel ?val_label }
WHERE { GRAPH <http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl>
{ ?selected rdfs:label "Gastric Mucosa-Associated Lymphoid Tissue Lymphoma" .
?selected ((rdfs:subClassOf|owl:equivalentClass)/(owl:intersectionOf/rdf:rest*/rdf:first)?)+ ?restriction .
?restriction owl:onProperty ?prop . ?restriction (owl:someValuesFrom|owl:allValuesFrom) ?val .
?selected rdfs:label ?sel_label . ?prop rdfs:label ?prop_label . ?val rdfs:label ?val_label .  } . }
}
OUTPUT <http://localhost/set1>
```

FIGURE 32. Generated IML query for the NCIt simplification query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX heart_parts: <http://localhost/set1#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_RECURSIVE { { fma:Heart fma:part ?Object } GRAPH <http://sig.biostr.washington.edu/fma3.0>
WHERE { { fma:Heart fma:regional_part ?Object .  }  UNION { fma:Heart fma:constitutional_part ?Object . } } }

{ { ?a fma:part ?Object . ?b fma:part ?Object } GRAPH <http://sig.biostr.washington.edu/fma3.0>
WHERE { GRAPH <recursive> { ?a fma:part ?b .  } . { ?b fma:regional_part ?Object .  }
UNION { ?b fma:constitutional_part ?Object . } } }
}
OUTPUT <http://localhost/set1>

INPUT <http://sig.biostr.washington.edu/fma3.0>, <http://localhost/set1>
{
ADD_EDGE < ?a fma:contains ?c > WHERE { GRAPH <http://localhost/set1> { ?a fma:part ?b .  } .
GRAPH <http://sig.biostr.washington.edu/fma3.0> { ?b fma:contains ?c .  } . }
ADD_EDGE < ?b fma:contains ?c > WHERE { GRAPH <http://localhost/set1> { ?a fma:part ?b .  } .
GRAPH <http://sig.biostr.washington.edu/fma3.0> { ?b fma:contains ?c .  } . }
}
OUTPUT <results>
```

FIGURE 33. Generated IML query for the blood contained in the heart query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX reactome: <http://purl.org/science/ontology/reactome/>
PREFIX tst: <http://localhost/tst#>
PREFIX tmp: <http://localhost/tmp#>
PREFIX view: <http://localhost/view#>
PREFIX input_list: <input_list#>
PREFIX graph_hierarchy: <graph_hierarchy#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
ADD_EDGE < tmp:set tmp:member fma:Blood_in_left_ventricle > ADD_EDGE < tmp:set tmp:member fma:Mitral_valve >
ADD_EDGE < tmp:set tmp:member fma:Wall_of_left_ventricle > ADD_EDGE < tmp:set tmp:member fma:Aortic_valve >
ADD_EDGE < tmp:set tmp:member fma:Blood_in_left_atrium > ADD_EDGE < tmp:set tmp:member fma:Wall_of_left_atrium >
}
OUTPUT <input_list>

INPUT <http://sig.biostr.washington.edu/fma3.0>, <input_list>
{
EXTRACT_TREE { ?x [ forward (fma:regional_part_of) , forward (fma:constitutional_part_of) , forward (fma:contained_in)  ]  }
GRAPH <http://sig.biostr.washington.edu/fma3.0> WHERE { GRAPH <input_list> { tmp:set tmp:member ?x .  } . }

DELETE_EDGE < ?x ?b ?x > WHERE { GRAPH <input_list> { ?c ?d ?x .  } . ?x ?b ?x . }
EXTRACT_EDGES { ?z tst:edge ?x } WHERE {?x ?y ?z . }
}
OUTPUT <graph_hierarchy>

INPUT <http://sig.biostr.washington.edu/fma3.0>, <graph_hierarchy>
{
EXTRACT_RECURSIVE { { fma:Human_body tmp:inter ?b1 } GRAPH <graph_hierarchy> WHERE {fma:Human_body tst:edge ?b1 . } }

{ { ?a1 view:part ?b1 . ?b1 tmp:inter ?c1 . ?b1 tmp:inter ?c2 } GRAPH <graph_hierarchy>
WHERE { GRAPH <recursive> { ?a1 tmp:inter ?b1 .  } . ?b1 tst:edge ?c1 . ?b1 tst:edge ?c2 . FILTER  ( ( ?c1 ) !=( ?c2 )  ) } }

{ { ?a1 tmp:inter ?d1 } GRAPH <graph_hierarchy>
WHERE { GRAPH <recursive> { ?a1 tmp:inter ?b2 .  } . ?b2 tst:edge ?d1 . OPTIONAL { ?b2 tst:edge ?d2 .
FILTER  ( ( ?d1 ) !=( ?d2 )  ) } FILTER  ( !bound( ?d2 )  ) } }

{ { ?a1 view:part ?b3 } GRAPH <graph_hierarchy>
WHERE { GRAPH <recursive> { ?a1 tmp:inter ?b3 .  } . OPTIONAL { ?b3 tst:edge ?e1 . } FILTER  ( !bound( ?e1 )  ) } }

EXTRACT_EDGES { ?a view:part ?c } WHERE { ?a view:part ?c . }
}
OUTPUT <restructure>
```

FIGURE 34. Generated IML query for the biosimulation model editor query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX holder: <http://localhost/holder#>
PREFIX VSparQL: <java:vsparql.ext.#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX opb: <http://sig.biostr.washington.edu/OPB-01.owl#>
PREFIX annot_view: <http://sig.biostr.washington.edu/annot_v1.0#>
PREFIX det: <http://sig.biostr.washington.edu/~detwiler/OntViews/OPB/OPB-01.owl/>
PREFIX portion_of_blood: <http://localhost/set1#>
PREFIX kinetic_properties: <http://localhost/set2#>

INPUT <http://sig.biostr.washington.edu/fma3.0>
{
EXTRACT_REACHABLE { fma:Portion_of_blood [ backward (rdfs:subClassOf)  ]  }
GRAPH <http://sig.biostr.washington.edu/fma3.0>
DELETE_EDGE < fma:Portion_of_blood ?b fma:Portion_of_blood >
WHERE { fma:Portion_of_blood ?b fma:Portion_of_blood . }
}
OUTPUT <http://localhost/set1>

INPUT <http://sig.biostr.washington.edu/~detwiler/OntViews/OPB/OPB-01.owl>
{
EXTRACT_REACHABLE { opb:Kinetic_property [ backward (rdfs:subClassOf)  ]  }
GRAPH <http://sig.biostr.washington.edu/~detwiler/OntViews/OPB/OPB-01.owl>
EXTRACT_EDGES { ?a ?b ?Property }
WHERE { ?a ?b ?Property . GRAPH <http://sig.biostr.washington.edu/~detwiler/OntViews/OPB/OPB-01.owl>
{ ?Property opb:Physical_domain_class opb:Fluid_domain .  } . }
}
OUTPUT <http://localhost/set2>

INPUT <http://sig.biostr.washington.edu/~detwiler/OntViews/OPB/OPB-01.owl>, <http://localhost/set1>, <http://localhost/set2>
{
ADD_EDGE < [[annot_view:annotation(?pob, ?property)]] rdf:type rdf:Statement >
WHERE { GRAPH <http://localhost/set1> { ?a ?b ?pob .  } . GRAPH <http://localhost/set2> { ?c ?d ?property .  } . }
ADD_EDGE < [[annot_view:annotation(?pob, ?property)]] rdf:subject ?pob >
WHERE { GRAPH <http://localhost/set1> { ?a ?b ?pob .  } . GRAPH <http://localhost/set2> { ?c ?d ?property .  } . }
ADD_EDGE < [[annot_view:annotation(?pob, ?property)]] rdf:object ?property >
WHERE { GRAPH <http://localhost/set1> { ?a ?b ?pob .  } . GRAPH <http://localhost/set2> { ?c ?d ?property .  } . }
ADD_EDGE < [[annot_view:annotation(?pob, ?property)]] rdf:predicate opb:hasProperty >
WHERE { GRAPH <http://localhost/set1> { ?a ?b ?pob .  } . GRAPH <http://localhost/set2> { ?c ?d ?property .  } . }
}
OUTPUT <http://localhost/set3>
```

FIGURE 35. Generated IML query for the blood fluid properties query

```
PREFIX fma: <http://sig.biostr.washington.edu/fma3.0#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX organ_subclass: <organ_subclass#>
PREFIX cardinal_organ_part_subclass: <cardinal_organ_part_subclass#>
PREFIX subclass: <subclass#>
PREFIX liver_parts: <liver_parts#>
PREFIX liver_parts_organs: <liver_parts_organs#>
PREFIX liver_parts_organ_superclasses: <liver_parts_organ_superclasses#>

INPUT <http://sig.biostr.washington.edu/fma3.0> {
EXTRACT_TREE { fma:Organ [ backward (rdfs:subClassOf)  ]  } GRAPH <http://sig.biostr.washington.edu/fma3.0>
} OUTPUT <organ_subclass>

INPUT <http://sig.biostr.washington.edu/fma3.0> {
EXTRACT_TREE { fma:Cardinal_organ_part [ backward (rdfs:subClassOf)  ]  } GRAPH <http://sig.biostr.washington.edu/fma3.0>
} OUTPUT <cardinal_organ_part_subclass>

INPUT <organ_subclass>, <cardinal_organ_part_subclass> {
UNION_GRAPHS <organ_subclass>, <cardinal_organ_part_subclass>
} OUTPUT <subclass>

INPUT <http://sig.biostr.washington.edu/fma3.0> {
EXTRACT_TREE { fma:Liver [ forward (fma:regional_part) , forward (fma:constitutional_part) , forward (fma:systemic_part)  ]  }
GRAPH <http://sig.biostr.washington.edu/fma3.0>
} OUTPUT <liver_parts>

INPUT <subclass>, <liver_parts>
{
EXTRACT_EDGES { ?j ?part_var ?k . ?k rdfs:subClassOf ?lk . fma:Liver rdfs:subClassOf ?lj }
WHERE { { GRAPH <liver_parts> { ?j ?part_var ?k . GRAPH <subclass> { ?k ?sco ?lk . FILTER  ( !isBlank( ?lk )  ) } .  } . }
UNION { GRAPH <subclass> { fma:Liver ?sco ?lj . FILTER  ( !isBlank( ?lj )  ) } . } }
}
OUTPUT <liver_parts_organs>

INPUT <http://sig.biostr.washington.edu/fma3.0>, <liver_parts_organs>
{
EXTRACT_TREE { ?l [ forward (rdfs:subClassOf)  ]  } GRAPH <http://sig.biostr.washington.edu/fma3.0>
WHERE { GRAPH <liver_parts_organs> { ?k rdfs:subClassOf ?l .  } . }
EXTRACT_EDGES { ?a ?b ?c } WHERE { ?a ?b ?c . FILTER (  ( !isBlank( ?a )  )  &&  ( !isBlank( ?c )  )  ) . }
}
OUTPUT <liver_parts_organ_superclasses>

INPUT <liver_parts_organs>, <liver_parts_organ_superclasses>
{
UNION_GRAPHS <liver_parts_organs>, <liver_parts_organ_superclasses>
DELETE_EDGE < fma:Solid_organ rdfs:subClassOf fma:Organ >
REPLACE_NODE fma:Solid_organ fma:Organ
DELETE_EDGE < fma:Cavitated_organ rdfs:subClassOf fma:Organ >
REPLACE_NODE fma:Cavitated_organ fma:Organ
}
OUTPUT <http://localhost/set1>
```

FIGURE 36. Generated IML query for the radiologist liver ontology