## Appendix A

### Java Glossary

<table>
<thead>
<tr>
<th>1. Package</th>
<th>50. Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. JavaStandardLibraryPackage</td>
<td>51. Unchecked</td>
</tr>
<tr>
<td>3. java.lang</td>
<td>52. RuntimeException</td>
</tr>
<tr>
<td>4. java.util</td>
<td>53. Error</td>
</tr>
<tr>
<td>5. Networking</td>
<td>54. Checked</td>
</tr>
<tr>
<td>6. RMI</td>
<td>55. IOException</td>
</tr>
<tr>
<td>7. Web Services</td>
<td>56. SQLException</td>
</tr>
<tr>
<td>8. CORBA</td>
<td>57. Tools</td>
</tr>
<tr>
<td>9. Sockets</td>
<td>58. IDE</td>
</tr>
<tr>
<td>10. IO</td>
<td>59. IntelliJ IDEA</td>
</tr>
<tr>
<td>11. Input</td>
<td>60. Eclipse</td>
</tr>
<tr>
<td>12. Reader</td>
<td>61. MyEclipse</td>
</tr>
<tr>
<td>13. Streams</td>
<td>62. editors</td>
</tr>
<tr>
<td>14. Output</td>
<td>63. emacs</td>
</tr>
<tr>
<td>15. Writer</td>
<td>64. JDEE mode</td>
</tr>
<tr>
<td>16. Streams</td>
<td>65. java-mode</td>
</tr>
<tr>
<td>17. Collections</td>
<td>66. NetBeans</td>
</tr>
<tr>
<td>18. Map</td>
<td>67. Profiling Tools</td>
</tr>
<tr>
<td>19. SortedMap</td>
<td>68. JProfiler</td>
</tr>
<tr>
<td>20. TreeMap</td>
<td>69. VisualVM</td>
</tr>
<tr>
<td>21. HashMap</td>
<td>70. jhat</td>
</tr>
<tr>
<td>22. Old</td>
<td>71. Server side</td>
</tr>
<tr>
<td>23. Vector</td>
<td>72. Application servers</td>
</tr>
<tr>
<td>24. HashTable</td>
<td>73. Naming</td>
</tr>
<tr>
<td>25. Iterators</td>
<td>74. Web Services</td>
</tr>
<tr>
<td>26. Algorithms</td>
<td>75. JMS</td>
</tr>
<tr>
<td>27. Comparator interface</td>
<td>76. Java-based Servers</td>
</tr>
<tr>
<td>28. Embedded Database</td>
<td>77. JBoss</td>
</tr>
<tr>
<td>30. HSQL</td>
<td>79. Glasfish</td>
</tr>
<tr>
<td>31. Apache Derby</td>
<td>80. WebSphere</td>
</tr>
<tr>
<td>32. Object</td>
<td>81. Thick Client</td>
</tr>
<tr>
<td>33. JavaStandardLibraryObject</td>
<td>82. Swing</td>
</tr>
<tr>
<td>34. java.lang.System.err</td>
<td>83. Applet</td>
</tr>
<tr>
<td>35. java.lang.System.in</td>
<td>84. Servlet</td>
</tr>
<tr>
<td>36. java.lang.System.out</td>
<td>85. JSP</td>
</tr>
<tr>
<td>37. AbstractMethod</td>
<td>86. Multiple inheritance</td>
</tr>
<tr>
<td>38. FinalMethod</td>
<td>87. Closures</td>
</tr>
<tr>
<td>40. StaticMethod</td>
<td>89. Continuations</td>
</tr>
<tr>
<td>41. MainMethod</td>
<td>90. No pass-by-reference</td>
</tr>
<tr>
<td>42. concurrency</td>
<td>91. First-class method</td>
</tr>
<tr>
<td>43. Thread Class</td>
<td>92. Python</td>
</tr>
<tr>
<td>44. Runnable interface</td>
<td>93. .Net</td>
</tr>
<tr>
<td>45. Locks</td>
<td>94. Objective C</td>
</tr>
<tr>
<td>46. Atomic objects</td>
<td>95. Identifier</td>
</tr>
<tr>
<td>47. Task execution</td>
<td>96. Statement</td>
</tr>
<tr>
<td>48. Concurrent containers</td>
<td>97. Exception Handling</td>
</tr>
<tr>
<td>49. Synchronization utilities</td>
<td>98. Interface</td>
</tr>
</tbody>
</table>
Appendix B

Concept Map

Figure B.1. Java Programming Structure
Figure B.2. Expression
Figure B.3. Statement
Figure B.4. Client side
Figure B.5. Server Side
Figure B.6. Class
Figure B.7. Exceptions
Figure B.8. IO
Figure B.9. Limitations of Java
Figure B.10. Tools
Appendix C

Java Ontology Visualization using Jambalaya Tool

Figure C.1. Java Programming
Figure C.2. Java OOPs
Figure C.3. Class
Figure C.4. Java Language Program
Figure C.5. Exceptions
Figure C.6. Interface
Figure C.7. Java Programming Structure
Figure C.8. Access Specifier and Modifier
Figure C.9. Program Element Body and Specification
Figure C.10. Package and Method
Appendix D

Enclosed Reprint of Two Papers from the Seven Published Papers


Matching and Merging of Ontologies Using Conceptual Graphs

Gopinath Ganapathy, Ravi Lourdusamy

Abstract—Knowledge management applications need to determine whether two or more knowledge representations encode the same knowledge. Solving this matching problem is hard because representations may encode the same content but differ substantially in form. Previous approaches to this problem have used either syntactic measure or semantic knowledge to determine the distance between two representations. The aim of this article is to define matching of two representations. The knowledge encoded can be equivalently encoded as conceptual graphs. The algorithm for matching of two graphs is discussed in [11]. In this paper the generalized algorithm for matching of multiple conceptual graphs is developed. The fitness of the matched graph is further investigated. Further the algorithm is extended to merge m-conceptual graphs.

Index Terms—Binding, Taxonomy, Transformation, Fitness.

I. INTRODUCTION

A requirement common to many knowledge applications is to determine whether two or more knowledge representations, encoded using the same ontology, capture the same knowledge. The task of determining whether two or more representations encode the same knowledge is treated as a graph matching problem. The knowledge representation is encoded using conceptual graph. The representations capture the same knowledge if their corresponding graphs match. The multiple encoding of the same knowledge rarely match exactly, so a matcher must be flexible to avoid a high rate of false-negatives. However, a matcher that is too flexible can suffer from a high rate of false-positives. This problem has various causes, including

(i) The ontology is expressive enough to allow the same information to be encoded in different ways

(ii) The representations are built by different knowledge engineers (or computer programs), raising the likelihood they differ

(iii) The representations are large, increasing the opportunity for differences.

Previous solutions to this problem have produced two types of matchers. Syntactic matchers use only the graphical form of the representations, judging their similarity by the amount of common structures shared [1], [2] or the number of edit operations required to transform one graph into the other [3], [7], [8], [10]. Approaches that focus on the amount of shared common structures do not handle mismatches. Approaches that use edit operations can handle mismatches but are sensitive to the cost assigned to the edit operations and tuning these parameters optimally is problematic.

In contrast, semantic matchers use knowledge, stored in an ontology, of the terms referenced in the representations. Semantic matchers use this knowledge to determine the match of two representations. [3], [5], [6], [12]. The knowledge encoded can be equivalently encoded as conceptual graphs [9]. The algorithm for matching of two graphs is discussed in [11]. In this paper the generalized algorithm for matching of m conceptual graphs is developed. The fitness of the matched graph is further investigated. Further the algorithm is extended to merge m-conceptual graphs.

The article is organized as follows: Section 1 deals with an introduction. The terminologies involved are outlined in Section 2. In Section 3, the algorithm for matching m-conceptual graphs is illustrated. The algorithm for merging m-conceptual graphs is illustrated in Section 4 and in Section 5 conclusion is presented.

II. PRELIMINARIES

In this section, we present some definitions and preliminaries which will be useful for further discussion.

Definition 2.1: [6] Transitive and part ascendant transformations conform to a more general notion called ‘transfers through’. A relation r transfers through another relation r if

\[ X \xrightarrow{r} Y \xrightarrow{r'} Z \implies X \xrightarrow{r} Z \]  

(1)

Definition 2.2: [6] A triple is a 3-tuple of the form \((\text{head}, \text{relation}, \text{tail})\) where head and tail are concepts or instances (i.e., nodes in a conceptual graph) and relation is an edge in the graph. Every two nodes connected by an edge in a conceptual graph can be mechanically converted into a triple and hence a conceptual graph into a set of triples.

Definition 2.3: The \(n\) triples \(t_1 = (\text{head}_1, \text{relation}_1, \text{tail}_1), t_2 = (\text{head}_2, \text{relation}_2, \text{tail}_2), \ldots, t_n = (\text{head}_n, \text{relation}_n, \text{tail}_n)\) of graph \(G\) align if \(head_1 \geq head_2 \geq \ldots \geq head_n, \text{uneg}(relation_1) \geq \text{uneg}(relation_2) \geq \ldots \geq \text{uneg}(relation_n)\) and \(tail_1 \geq tail_2 \geq \ldots \geq tail_n\). The uneg(relation) unnegates relation if it is negated otherwise returns the relation.

Definition 2.4: For \(\ell = \{(t_{11}, t_{21}, \ldots, t_{m1}), (t_{12}, t_{22}, \ldots, t_{m2}), \ldots, (t_{1n_1}, t_{2n_1}, \ldots, t_{mn_1})\}\), a list of aligned triples of \(m\) graphs, the bindings for \(\ell\) is:

\[ b(\ell) = \{(\text{head}_{11}/\text{head}_{21}/\ldots/\text{head}_{m1}, \text{tail}_{11}/\text{tail}_{21}/\ldots/\text{tail}_{m1}), \]

\[ (\text{head}_{12}/\text{head}_{22}/\ldots/\text{head}_{m2}, \text{tail}_{12}/\text{tail}_{22}/\ldots/\text{tail}_{m2}), \ldots, (\text{head}_{1n_1}/\text{head}_{2n_1}/\ldots/\text{head}_{mn_1}, \text{tail}_{1n_1}/\text{tail}_{2n_1}/\ldots/\text{tail}_{mn_1})\} \]

III. MATCHING OF M-CONCEPTUAL GRAPHS

A. Binding of \(m\)-conceptual Graphs

Given \(m\) graphs \(G_1 = \{t_{11}, t_{12}, \ldots, t_{1n_1}\}, G_2 = \{t_{21}, t_{22}, \ldots, t_{2n_2}\}, \ldots, G_m = \{t_{mn_1}, t_{mn_2}, \ldots, t_{mn_m}\}\) where \(n_1, n_2, \ldots, n_m\) are the number of triples of \(G_1, G_2, \ldots, G_m\) respectively and a set of \(r\) transformations \(R\) where \(R = \{R_1, R_2, \ldots, R_r\}\). The aim of the algorithm is to find
a common subgraph of $G_1, G_2, \ldots$ and $G_m$ called $SG$.
Construct a list $M$ of all possible alignments between
the triples of $G_1, G_2, \ldots$ and $G_m$. Each element of $M$
is of the form $\ell = \{(t_{11}, t_{21}, \ldots, t_{m1}), (t_{12}, t_{22}, \ldots, t_{m2}),\ldots(t_{1m}, t_{2m}, \ldots, t_{mm})\}$. The generalized
algorithm for finding a match between $m$ representations, is presented
as Algorithm-1. The steps for finding a match between $m$
representations is illustrated with an example of three graphs,$G_1, G_2$ and $G_3$ generated by organization structure of three
hospitals shown in Figures - 1, 2 and 3. For reference, we
label each triple in $G_1$ with a unique number from 1 to 22,
each triple in $G_2$ with a unique upper case letter from A to
V and each triple in $G_3$ with a unique lowercase letter from
a to z and from aa to hh. We use subscripts to differentiate
terms that appear multiple times (e.g., Hospital..Model..1).

**Algorithm - 1: Outline of the generalized matching algorithm**

1) $M = NIL$ and $\ell = NIL$

FOR each triple $t_{i1}$ in $G_1$

FOR each triple $t_{2i}$ in $G_2$

$\rule{10cm}{0cm}$

FOR each triple $t_{mi}$ in $G_m$

IF $t_{i1}, t_{2i}, \ldots, t_{mi}$ are aligned

THEN add $(t_{i1}, t_{2i}, \ldots, t_{mi})$ to $\ell$.

ADD $\ell$ to $M$ and reset $\ell$ to NIL.

2) Use $M$ to construct a common subgraph of $G_1, G_2, \ldots, G_m$
called $SG$.

$SG = \{(t_{11}, \ldots, t_{m1}), (t_{12}, \ldots, t_{m2}), \ldots, (t_{1m}, \ldots, t_{mm})\}$

where $(t_{11}, \ldots, t_{m1})$ are the aligned triples of $G_1, G_2, \ldots, G_m$
respectively.

3) IF $SG$ is inconsistent THEN stop and return NIL.

4) FOR each rule $R_i$ in $R$, FOR each $j = 1, 2, \ldots, m$

FOR each $k = 1, 2, \ldots, m$

IF $R_i$ is applicable to $G_j$ with respect to $G_k$

THEN apply ($R_i; G_j, G_k$)

5) FOR each unaligned triple $t_{ji}$ in $G_j$

IF $t_{i1}, t_{2i}, \ldots, t_{mi}$ are aligned and

$b(\{(t_{i1}, t_{2i}, \ldots, t_{mi})\})$ is consistent with $b(SG)$,

THEN add $(t_{i1}, t_{2i}, \ldots, t_{mi})$ to $SG$ and break.

UNTIL $SG$ reaches quiescence go to step 4.

6) RETURN $SG$.

In step 1 the generalized algorithm compares each triple in
$G_1$ with each triple in $G_2, G_3, \ldots, G_m$ to find all possible
alignments. In our example $t_{i1}$ aligns with $t_{21}$ and $t_{31}$. 
Triple $t_{i1}$, however, does not align with triple $t_{22}$ and its
combinations because the relations differ. The initial match
is denoted by $M$. $M = \{\ell_1, \ell_2, \ldots, \ell_p\}$ where $p$ is the
total number of possible alignments. Each element of $M$ is a list
called $\ell$. For example $\{t_{11}, t_{21}, t_{31}\}$ is called $\ell_1$,
$\{t_{12}, t_{22}, t_{32}\}$ is called $\ell_2$, etc. In step 2 the generalized
algorithm uses $M$ to construct common subgraph of $G_1, G_2, \ldots, G_m$ called $SG$. The generalized algorithm begins by
selecting a member, $\ell_i$, of $M$ to serve as the seed of
the construction process (recall that $M = \{\ell_1, \ell_2, \ldots, \ell_p\}$)
and $\ell_i = \{(t_{i1}, t_{21}, t_{31}), (t_{i1}, t_{22}, t_{32}), \ldots, (t_{i1}, t_{2k}, t_{3m}),\}$, $i = 1, 2, \ldots, s$. This seed is selected
based on a heuristic scoring function

\[
h(\ell_i) = \frac{1}{K_1} \sum_{j=1}^{k} n(head_{1j}/head_{2j}/\ldots/head_{mj})
+n(tail_{1j}/tail_{2j}/\ldots/tail_{mj}) \tag{2}
\]

where $head_{1j}, head_{2j}, \ldots, head_{mj}$ and
tail_{1j}, tail_{2j}, \ldots, tail_{mj}$ are the bindings of
$t_{i1}, t_{2i}, \ldots, t_{mi}$ and $n(b)$ is the number of times
the bindings $b$ occurs in $binding(M)$. This function
$h$ is a heuristic that favors those $\ell_i$ in $M$ with
interconnectivity. Bindings that occur frequently indicate...
In steps 3-5, the generalized algorithm checks if SG is consistent. SG is inconsistent if it contains an aligned m-tuples of triples \((t_1, t_2, ..., t_m)\) where the relation of at least one \(t_{kj}\) is not negated. If SG contains such a \(m\)-tuples, then generalized algorithm stops and returns NIL. Otherwise, the generalized algorithm applies transformations to improve the match(i.e., steps 4 and 5). Steps 4 and 5 are repeated until SG reaches quiescence. In step 4, the generalized algorithm applies transformations to resolve mismatches among \(G_1, G_2, ..., G_m\). In step 5, the generalized algorithm will try to align additional triples among \(G_1, G_2, ..., G_m\). Step 5 is like step 1 except that it focuses on the unaligned triples. We have identified a set of transformations for the health care domain. These transformations are used to improve the matching of \(m\)-conceptual graphs in the domain.

Returning to our example, the triples of \(G_1, G_2\) and \(G_3\) are as follows:

\[
G_1 = \{(\text{Hospital}_1, \text{hasAct}, \text{Admit}_1), (\text{Hospital}_1, \text{hasAct}, \text{NewRenewReg}_1), (\text{Hospital}_1, \text{hasAct}, \text{AdmitPatient}_1), (\text{Patient}_1, \text{hasRole}, \text{DiagnosePatient}_1), (\text{DiagnosePatient}_1, \text{hasAct}, \text{Registration}_1), (\text{Registration}_1, \text{hasRole}, \text{Nurse}_1), (\text{Nurse}_1, \text{hasRole}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Stethoscope}_1, \text{isPlayedBy}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Nurse}_1, \text{hasAct}, \text{Registration}_1), (\text{Registration}_1, \text{hasAct}, \text{NewRenewReg}_1), (\text{NewRenewReg}_1, \text{hasRole}, \text{Nurse}_1), (\text{Nurse}_1, \text{hasRole}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Stethoscope}_1, \text{isPlayedBy}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Nurse}_1, \text{hasAct}, \text{Registration}_1), (\text{Registration}_1, \text{hasRole}, \text{DiagnosePatient}_1), (\text{DiagnosePatient}_1, \text{hasAct}, \text{Registration}_1), (\text{Registration}_1, \text{hasRole}, \text{DiagnosePatient}_1), (\text{DiagnosePatient}_1, \text{hasRole}, \text{Nurse}_1), (\text{AdmitPatient}_1, \text{hasRole}, \text{Nurse}_1), (\text{Nurse}_1, \text{hasRole}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Stethoscope}_1, \text{isPlayedBy}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Nurse}_1, \text{hasAct}, \text{Registration}_1), (\text{Registration}_1, \text{hasRole}, \text{DiagnosePatient}_1), (\text{DiagnosePatient}_1, \text{hasRole}, \text{Nurse}_1), (\text{AdmitPatient}_1, \text{hasRole}, \text{Nurse}_1), (\text{Nurse}_1, \text{hasRole}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Stethoscope}_1, \text{isPlayedBy}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Nurse}_1, \text{hasAct}, \text{Registration}_1), (\text{Registration}_1, \text{hasRole}, \text{DiagnosePatient}_1), (\text{DiagnosePatient}_1, \text{hasRole}, \text{Nurse}_1), (\text{AdmitPatient}_1, \text{hasRole}, \text{Nurse}_1), (\text{Nurse}_1, \text{hasRole}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Stethoscope}_1, \text{isPlayedBy}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Nurse}_1, \text{hasAct}, \text{Registration}_1), (\text{Registration}_1, \text{hasRole}, \text{DiagnosePatient}_1), (\text{DiagnosePatient}_1, \text{hasRole}, \text{Nurse}_1), (\text{AdmitPatient}_1, \text{hasRole}, \text{Nurse}_1), (\text{Nurse}_1, \text{hasRole}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Stethoscope}_1, \text{isPlayedBy}, \text{Physician}_1), (\text{Physician}_1, \text{plays}, \text{Stethoscope}_1), (\text{Nurse}_1, \text{hasAct}, \text{Registration}_1)
\]

In step 1, the algorithm compares each triple in \(G_1\) with each triple in \(G_2\) and \(G_3\) to find all possible alignments. Triple \(1\) of \(G_1\) aligns with \(A\) of \(G_2\) and \(a\) of \(G_3\). Triples of \(G_1\) aligns with \(B\) of \(G_2\) and \(b\) of \(G_3\) and so on. Hence the matched triples of \(G_1, G_2\) and \(G_3\) are as follows:

\[
\text{Binding}(M) = \{(1, a, A), (2, B, b), (3, C, c), (4, D, d), (5, E, e), (6, F, f), (7, G, g), (8, H, h), (9, I, i), (10, J, j), (11, K, k), (12, L, l), (13, M, m), (14, N, n), (15, O, o), (16, P, p)\} = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7, \ell_8, \ell_9, \ell_{10}, \ell_{11}, \ell_{12}, \ell_{13}, \ell_{14}, \ell_{15}, \ell_{16}\}
\]

In step 2, hscore is calculated as follows: \(hscore(\ell_1) = hscore(\ell_2) = hscore(\ell_3) = 10, hscore(\ell_4) = 18, hscore(\ell_5) = hscore(\ell_6) = 10, hscore(\ell_7) = 14\) where \(j = 7, 8, ..., 16\). Select \(\ell_4\) and remove it from \(M\) and hence the subgraph \(SG = \{4, D, d\}\):

\[
\text{Binding}(M) = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7, \ell_8, \ell_9, \ell_{10}, \ell_{11}, \ell_{12}, \ell_{13}, \ell_{14}, \ell_{15}, \ell_{16}\}
\]

The head of \(\ell_4\) is \(\text{Registration}_1\) and remove it from \(SG\) and hence the subgraph \(SG = \{4, D, d\}\):

\[
\text{Binding}(M) = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7, \ell_8, \ell_9, \ell_{10}, \ell_{11}, \ell_{12}, \ell_{13}, \ell_{14}, \ell_{15}, \ell_{16}\}
\]

The tail of \(\ell_4\) is \(\text{Registration}_1\) and remove it from \(SG\) and hence the subgraph \(SG = \{4, D, d\}\):

\[
\text{Binding}(M) = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7, \ell_8, \ell_9, \ell_{10}, \ell_{11}, \ell_{12}, \ell_{13}, \ell_{14}, \ell_{15}, \ell_{16}\}
\]

The result is the maximal subgraph of \(G_1, G_2\) and \(G_3\) which is shown in Figure 4.
In this section we present an algorithm to merge \( m \)-conceptual graphs after determining the matched graph. Let \( p_1, p_2, \ldots, p_m \) be the number of unmatched triples of the graphs \( G_1, G_2, \ldots, G_m \). Assume that the matched graph \( SG \) as \( G \) and consider each unmatched triple \( t_{ji} \) of a graph \( G_i \). Search \( G \) to determine the node containing the head of \( t_{ji} \). After determining such node, \( t_{ji} \) is added as a subnode of the searched graphs node. After adding all the unmatched triples in \( G \), the resultant graph is the merged graph of \( m \) Graphs \( G_1, G_2, \ldots, G_m \). Returning to our example the merged graph of \( G_1, G_2, \) and \( G_3 \) is shown in Figure 6.

\begin{algorithm}
(1) Set \( SG \) as \( G \)  
(2) FOR each unmatched triple \( t_{1ji} \) in \( G_1 \)  
   FOR each unmatched triple \( t_{2lj} \) in \( G_2 \)  
   ..........  
   FOR each unmatched triple \( t_{mki} \) in \( G_m \)  
   search \( G \) for the head of \( t_{1ji} \), in \( n_i \)  
\end{algorithm}
Determine the node where there is a match add $t_{ji}$ as a subnode for that searched node and return the resultant graph as $G$.

(3) RETURN $G$

V. CONCLUSION

Algorithms for matching and merging of $m$-conceptual graphs are developed and the theoretical aspects are discussed in detail. Further these algorithms can be implemented using any ontology management software to support its application to the health care domain.

REFERENCES

Towards Ontology Development for Teaching Programming Language

Gopinath Ganapathi, Ravi Lourdusamy, Veeraraghavan Rajaram

Abstract—Several research frameworks are proposed for the development of teaching ontologies. Ontology as a conceptual structure may work as a mind tool for effective teaching and a visual navigation interface to the learning objects. In this paper we have discussed an approach to the practical ontology development and presented the designed ontology for teaching JAVA programming.

Index Terms—Education Ontology, Mind Map, Concept map, Semantic Web Rule Language.

I INTRODUCTION

Knowledge representation and reasoning is an area of artificial intelligence whose fundamental goal is to represent knowledge in a manner that facilitates inferencing (i.e. drawing conclusions) from knowledge. It analyzes how to formally think and how to use a symbol system to represent a domain of discourse (that which can be talked about), along with functions that allow inference (formalized reasoning) about the objects. Generally speaking, some kind of logic is used both to supply formal semantics of how reasoning functions apply to symbols in the domain of discourse, as well as to supply operators such as quantifiers, model operators, etc. that, along with an interpretation theory, give meaning to the sentences in the logic.

Teachers as knowledge engineers are used to work with concept maps, mind maps, brain maps, semantic networks, frames [5],[9],[15] and other conceptual structures. The visual representation of the general domain concepts facilitates and supports students understanding of both semantic and syntactic knowledge. A teacher operates as a knowledge analyst by making the skeleton of the studied discipline visible and showing the domain’s conceptual structure. Ontology can be used to represent the domain’s conceptual structure. However, Ontology-based approaches to teaching are relatively new fertile research areas. They originated in the area of knowledge engineering [3], [6], [18], which were then transferred to ontology engineering [7], [8], [10].

Date of paper submission is March 6th 2011.

Gopinath Ganapathi is with the Department of Computer Science, Bharathidasan University, Trichy, India. (e-mail: gganapathy@gmail.com).

Ravi Lourdusamy is with Department of Computer Science, Sacred Heart College, Tirupattur, India. ( +91 9443280319; fax: +91 4179 225060; e-mail: raviatshe@yahoo.com).

Veeraragavan Rajaram is with Department of Computer Science, Sacred Heart College, Tirupattur, India. ( +91 9952598990 – +91 04179-244287 e-mail: ragavan.srv@gmail.com)

Knowledge Engineering traditionally emphasized and rapidly developed a range of techniques and tools including knowledge acquisition, conceptual structuring and representation models [1], [14]. Section1 deals with an introduction. In section 2 the related work are discussed to understand the background of the proposed research.

The theoretical issues of ontology engineering are discussed in section 3. An approach for developing teaching ontologies is presented in Section 4. Section 5 proposes the conclusion.

II. REVIEW RELATED WORKS

Several practical approaches for developing teaching ontologies are proposed. The research framework prepared by Tatiana Gavrilova pursues a methodology that will scaffold the process of Knowledge Structure and ontology design is discussed. Moreover, special stress is placed on visual design as a powerful mind tool. The process of developing a practical ontology from the domain of introductory C Programming is described [17]. The automatic mapping of ontology into java proposed by Aditya Kalyanpur creates a set of java classes using OWL ontology. The OWL ontology files created represents an instance of a single ontology class with its properties, class relationships and restriction-definitions maintained [2].

An approach to design and develop teaching ontologies is discussed by Tatiana Gavrilova. The teaching ontologies are used for teaching and learning C programming concepts [13].

JLOO (Java Learning Object Ontology) is a frame work, for organizing learning objects of Java course in an adaptive e-learning environment. The classification in JLOO is based on the Computing Curricula CC2001 of the ACM and IEEE/CS. Using the curriculum as a guideline; the ontology defines the atomic knowledge units (i.e. learning objects) for an introductory course of java programming. The most significant contributions of JLOO are: 1) Defining the atomic knowledge units of introductory courses of Java language, and the relationships among them, 2) Making the knowledge units of JLOO sharable and reusable, 3) Allowing different learning strategies of an e-learning environment to choose dynamically, using JLOO as a guideline, different learning paths, and 4) Making the realization of adaptive learning easy [11].

III. USING ONTOLOGICAL ENGINEERING FOR TEACHING PURPOSES
The theoretical issues of ontological engineering are discussed in this section by reviewing different definitions of ontology from literature circulated within the field.

A. Ontology Definitions

Ontology is a set of distinctions we make in understanding and viewing the world.

“An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary.” [12].

“An ontology is a formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use, are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group.” [16].

Ontology as a useful structuring tool may greatly enrich the teaching process, providing students an organizing axis to help them mentally mark their visions in the information hyper-space of the domain knowledge.

B. Ontology Development

Tatiana Gavrilova, [17] has proposed a 5-steps recipe for developing ontology:

Glossary development: The first step should be devoted to gathering all the information relevant to the described domain. The main goal of this step is selecting and verbalizing all the essential objects and concepts in the domain.

Laddering: Having all the essential objects and concepts of the domain in hand, the next step is to define the main levels of abstraction. It is also important to elucidate the type of ontology classification, such as taxonomy, partition, and genealogy. This is being done at this step since it affects the next stages of the design. Consequently, the high level hierarchies among the concepts should be revealed and the hierarchy should be represented visually on the defined levels.

Disintegration: The main goal of this step is to break high level concepts, built in the previous step, into a set of detailed ones where it is needed. This could be done via a top-down strategy trying to break the high level concept from the root of the previously built hierarchy.

Categorization: At this stage, detailed concepts are revealed in a structured hierarchy. A generalization is performed via bottom-up structuring strategy. This could be carried out by associating similar concepts to create meta-concepts from leaves of the aforementioned hierarchy.

Refinement: The final step is devoted to update the visual structure by excluding the excessiveness, synonymy, and contradictions.

IV. DEVELOPING PRACTICAL ONTOLOGY

In this section an attempt to develop ontology for JAVA programming language following the five step algorithm as discussed in Section 3.2 is defended.

A. Glossary Development

The first step in building ontology is collecting information in the domain and building a glossary of the terms of the domain. To build a glossary for teaching introductory JAVA programming course, the terms are generated from two different types of resources: closed-corpus material and open-corpus material.

The closed corpus materials are in the form of lecture notes that are precisely designed for the course. The open corpus materials include several online tutorials in JAVA programming. The terms were extracted from the lecture notes manually by carefully reviewing the lecture handout. The terms from open-corpus material were extracted automatically [4]. Consequently, the automatically extracted terms and manually extracted terms are combined to build a single glossary. The glossary for Java programming language is developed and it consists of 530 words.

B. Laddering: Building an Initial Mind Map Structure

The second step is to build an initial visual structure of the glossary terms. The main goal of this step is to create a set of preliminary concepts and the categorization of those terms into concepts. A mind map can be a useful visual structure. The mind map developed to design java teaching ontology is presented in Figure 1. Since the categorization is preliminary, some of the terms might not fit into any of this initial categorization. The categorization is done manually in this step. However the lecture notes employed were used to build glossary, and to build the initial categorization as well. When designing the ontology the lecture notes are equally compared with experts help.
C. Disintegration and Categorization: Building a Concept Map with more Precise Hierarchy

The next step is to build a visual structure by analyzing the glossary. First we employed the top-down design strategy to create meta-concepts such as “Class”, “Object”, and “IO”. Then using the bottom-up strategy we tried to fit the terms and concepts into the meta-concept. We have created the relationships between the concepts. (Figure 2)

A concept map is the most useful visual structure for representation of the results of this stage, since it gives the ability of defining the relationship in addition to building the hierarchy. The output of this step is a large and detailed map, which covers the course in a hierarchical way. However, since this ontology is designed for teaching purposes it is important to offer the overall picture and a general hierarchy as well.

D. Refinement

The general ontology is compared with a refined ontology to update and to get the next version of ontology (Figures 3 and 4). Hence it is an incremental approach. It’s not easy to build all relationship in depth of knowledge. To get the clarity on the ontology developed we have to remove unnecessary node and use the standard relationship that are easy to understand.

V. CONCLUSION

Our java ontology can be integrated with any E-learning platform for class room teaching purposes. The java ontology developed can be further enhanced by adding Semantic Web Rule Language (SWRL), to infer more knowledge.

Our research stresses the role of knowledge structuring for developing ontology rapidly, professionally and successfully. The visual paradigm which is used to represent and support the teaching process not only helps a professional trainer to concentrate on problem rather than on details, but also enables a trainee to process and understand great volume of information.

At a basic level of knowledge representation, within the context of everyday heuristics, it is easier for educationalists simply to draw the ontology using conventional “pen and pencil” techniques. However, for more complicated knowledge representations, it is necessary to master appropriate programming and the involved language, or to use well-known ontology editors.

This described approach can be applied to developing those teaching systems where general understanding is more important than factual details. Furthermore,
ontology design may be used as an assessment procedure for significant as opposed to exploratory learning. For both formative and summarizing assessment purposes, students can clearly indicate the extent as well as the nature of their knowledge and understanding through creating ontology and explaining the involved processes.

REFERENCES